Flambé Documentation

Release 0.4.9

ASAPP Inc.

Dec 30, 2020

Getting Started

1 Installation **5**
- 1.1 Via `pip` . . . 5
- 1.2 From source . . . 5

2 Quickstart **7**
- 2.1 Executing Flambé . . . 7
- 2.2 A Simple Experiment . . . 8
- 2.3 Monitoring the Experiment . . . 9
- 2.4 Artifacts . . . 10
- 2.5 Recap . . . 11
- 2.6 Try it yourself! . . . 11
- 2.7 Next Steps . . . 12

3 Motivation **13**
- 3.1 Why Flambé? . . . 13
- 3.2 Core values . . . 13

4 Components **15**
- 4.1 Loading and Dumping from YAML . . . 15
- 4.2 Saving and Loading State . . . 16
- 4.3 Delayed Initialization . . . 16
- 4.4 Core Components . . . 17
- 4.5 Custom Component . . . 18

5 Runnables **21**
- 5.1 Providing secrets . . . 22
- 5.2 Automatic extensions installation . . . 22
- 5.3 Other flags . . . 23

6 Experiments **25**
- 6.1 Pipeline . . . 25
- 6.2 Linking . . . 26
- 6.3 Search Options . . . 26
- 6.4 Reducing . . . 28
- 6.5 Resources (Additional Files and Folders) . . . 28
- 6.6 Scheduling and Reducing Strategies . . . 29
- 6.7 General Logging . . . 30

6.8 Tensorboard Logging . . . 30
6.9 Script Usage . . . 31
6.10 Checkpointint and Saving . . . 31
6.11 Resuming . . . 32
6.12 Debugging . . . 33
6.13 Adding Custom State . . . 33

7 Report Site **35**
7.1 How to launch the report site? . . . 35

8 Extensions **43**
8.1 Building Extensions . . . 43
8.2 Using Extensions . . . 44

9 Clusters **47**
9.1 Overall remote architecture . . . 47
9.2 Launching a cluster . . . 48
9.3 Setting the cluster up . . . 49
9.4 Submitting Jobs to a Cluster . . . 49
9.5 Using AWS . . . 50
9.6 Intelligent versioning . . . 51
9.7 Cluster Runnables . . . 51
9.8 Remote Experiments . . . 52

10 Builders **53**
10.1 Motivation . . . 53
10.2 How to use a builder . . . 54
10.3 Future Work . . . 55

11 Security **57**
11.1 Secrets . . . 57
11.2 Clusters . . . 58
11.3 Extensions . . . 58

12 Advanced **59**
12.1 Developer Mode . . . 59
12.2 Cache Git-based Extensions . . . 59
12.3 Debugging . . . 59
12.4 Custom YAML Tags . . . 60

13 Converting a script to Flambé **63**
13.1 Wrapping your script in a pip installable . . . 63
13.2 Writing a config file . . . 64

14 Using Custom Code in Flambé **65**
14.1 Writing your custom code . . . 65
14.2 Setting up your extension . . . 65
14.3 Using your extension . . . 66

15 Writing a multistage pipeline: BERT Fine-tuning + Distillation **69**
15.1 First step: BERT fine-tuning . . . 69
15.2 Second step: Knowledge distillation . . . 71
15.3 Full configuration . . . 72

16 Creating a cluster with existing instances **75**
16.1 Instances in a cloud service provider . . . 75

16.2 Instances in the private LAN . . . 77
16.3 More information . . . 77

17 Creating a cluster using Amazon Web Services (AWS) 79
17.1 Setting up your AWS account . . . 79
17.2 Creating a `AWSCluster` . . . 90
17.3 Reusing a `AWSCluster` . . . 92

18 `flambe.dataset` 95
18.1 Submodules . . . 95
18.2 Package Contents . . . 99

19 `flambe.cluster` 103
19.1 Subpackages . . . 103
19.2 Submodules . . . 119
19.3 Package Contents . . . 131

20 `flambe.compile` 143
20.1 Submodules . . . 143
20.2 Package Contents . . . 161

21 `flambe.experiment` 173
21.1 Subpackages . . . 173
21.2 Submodules . . . 174
21.3 Package Contents . . . 182

22 `flambe.export` 187
22.1 Submodules . . . 187
22.2 Package Contents . . . 188

23 `flambe.field` 191
23.1 Submodules . . . 191
23.2 Package Contents . . . 194

24 `flambe.learn` 197
24.1 Submodules . . . 197
24.2 Package Contents . . . 201

25 `flambe.logging` 205
25.1 Subpackages . . . 205
25.2 Submodules . . . 207
25.3 Package Contents . . . 215

26 `flambe.metric` 223
26.1 Subpackages . . . 223
26.2 Submodules . . . 227
26.3 Package Contents . . . 228

27 `flambe.model` 233
27.1 Submodules . . . 233
27.2 Package Contents . . . 233

28 `flambe.nlp` 235
28.1 Subpackages . . . 235

29 flambe.nn **249**
29.1 Subpackages . . . 249
29.2 Submodules . . . 254
29.3 Package Contents . . . 269

30 flambe.runnable **281**
30.1 Submodules . . . 281
30.2 Package Contents . . . 287

31 flambe.runner **293**
31.1 Submodules . . . 293

32 flambe.sampler **295**
32.1 Submodules . . . 295
32.2 Package Contents . . . 298

33 flambe.tokenizer **301**
33.1 Submodules . . . 301
33.2 Package Contents . . . 304

34 flambe.vision **307**
34.1 Subpackages . . . 307

Python Module Index **311**

Index **313**

Welcome to Flambé, a PyTorch-based library that allows users to:

- Run complex experiments with multiple training and processing stages.
- Search over an arbitrary number of parameters and reduce to the best trials.
- Run experiments remotely over many workers, including full AWS integration.
- Easily share experiment configurations, results and model weights with others.

Visit the github repo: https://github.com/asappresearch/flambe

A simple Text Classification experiment

```
!Experiment

name: sst-text-classification

pipeline:

  # stage 0 - Load the Stanford Sentiment Treebank dataset and run preprocessing
  dataset: !SSTDataset
    transform:
      text: !TextField
      label: !LabelField

  # Stage 1 - Define a model
  model: !TextClassifier
      embedder: !Embedder
        embedding: !torch.Embedding  # automatically use pytorch classes
          num_embeddings: !@ dataset.text.vocab_size
          embedding_dim: 300
        embedding_dropout: 0.3
        encoder: !PooledRNNEncoder
          input_size: 300
          n_layers: !g [2, 3, 4]
          hidden_size: 128
          rnn_type: sru
          dropout: 0.3
      output_layer: !SoftmaxLayer
          input_size: !@ model[embedder][encoder].rnn.hidden_size
          output_size: !@ dataset.label.vocab_size

  # Stage 2 - Train the model on the dataset
  train: !Trainer
    dataset: !@ dataset
    model: !@ model
    train_sampler: !BaseSampler
    val_sampler: !BaseSampler
    loss_fn: !torch.NLLLoss
    metric_fn: !Accuracy
    optimizer: !torch.Adam
      params: !@ train[model].trainable_params
    max_steps: 10
    iter_per_step: 100

  # Stage 3 - Eval on the test set
  eval: !Evaluator
    dataset: !@ dataset
    model: !@ train.model
```

(continues on next page)

(continued from previous page)

```
    metric_fn: !Accuracy
    eval_sampler: !BaseSampler

# Define how to schedule variants
schedulers:
  train: !ray.HyperBandScheduler
```

The experiment can be executed by running:

```
flambe experiment.yaml
```

Tip: All objects in the `pipeline` are subclasses of *`Component`*, which are automatically registered to be used with YAML. Custom *`Component`* implementations must implement `run()` to add custom behavior when being executed.

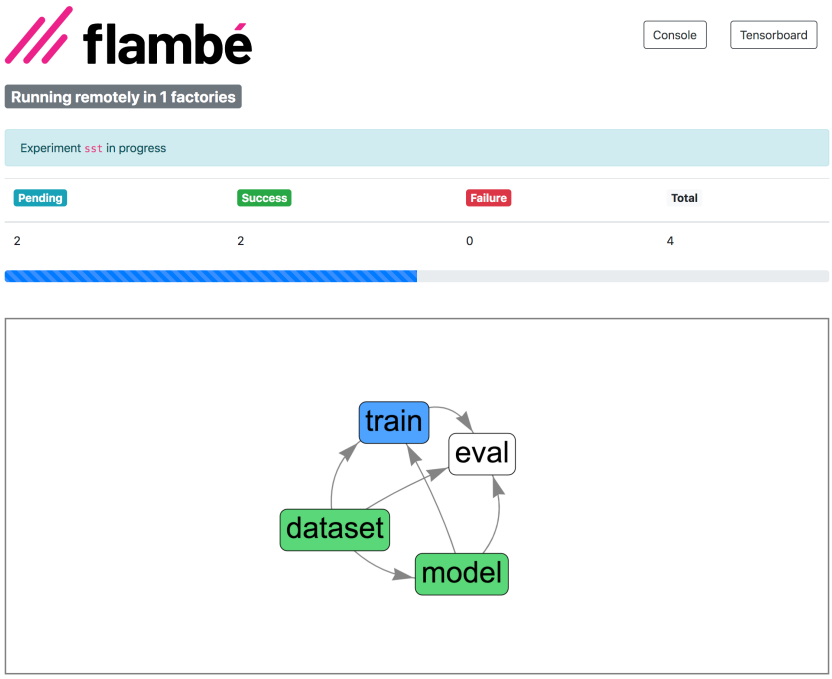
By defining a cluster:

```
!AWSCluster

name: my-cluster  # Make sure to name your cluster

factories_num: 2 # Number of factories to spin up, there is always just 1 orchestrator

factories_type: g3.4xlarge
orchestrator_type: t3.large

key: '/path/to/ssh/key'

...
```

Then the same experiment can be run remotely:

```
flambe experiment.yaml --cluster cluster.yaml
```

Progress can be monitored via the Report Site (with full integration with Tensorboard):



Flambé

Getting Started

Check out our *Installation Guide* and *Quickstart* sections to get up and running with Flambé in just a few minutes!

CHAPTER 1

Installation

1.1 Via `pip`

You can install the latest **stable** version of flambé as follows:

```
pip install flambe
```

1.2 From source

For the lastest version you can install from source:

```
git clone git@github.com:asappresearch/flambe.git
cd flambe
pip install .
```

Hint: We recommend installing flambé in an isolated virtual environment

CHAPTER 2

Quickstart

Flambé runs processes that are described using YAML files. When executing, Flambé will automatically convert these processes into Python objects and it will start executing them based on their behavior.

One of the processes that Flambé is able to run is an *Experiment*:

Listing 1: simple-exp.yaml

```yaml
!Experiment

name: sst

pipeline:

  # stage 0 - Load the dataset object SSTDataset and run preprocessing
  dataset: !SSTDataset
    transform:
      text: !TextField  # Another class that helps preprocess the data
      label: !LabelField
```

This *Experiment* just loads the Stanford Sentiment Treebank dataset which we will use later.

Important: Note that all the keywords following `!` are just Python classes (`Experiment`, `SSTDataset`, `TextField`, `LabelField`) whose keyword parameters are passed to the `__init__` method.

2.1 Executing Flambé

Flambé can execute the previously defined `Experiment` by running:

```
flambe simple-exp.yaml
```

Because of the way `Experiments` work, flambé will start executing the `pipeline` sequentially. Once done, you should see the generated artifacts in `flambe-output/output__sst/`. Obviously, these artifacts are useless at this point. Let's add a Text Classifier model and train it with this same dataset:

See also:

For a better understanding of *`Experiment`* read the *Experiments* section.

2.2 A Simple Experiment

Lets add a second stage to the `pipeline` to declare a text classifier. We can use Flambé's *`TextClassifier`*:

```
!Experiment

name: sst
pipeline:

  # stage 0 - Load the dataset object SSTDataset and run preprocessing
  [...]  # Same as before


  # stage 1 - Define the model
  model: !TextClassifier
    embedder: !Embedder
      embedding: !torch.Embedding
        num_embeddings: !@ dataset.text.vocab_size
        embedding_dim: 300
      encoder: !PooledRNNEncoder
        input_size: 300
        rnn_type: lstm
        n_layers: !g [2, 3, 4]
        hidden_size: 256
    output_layer: !SoftmaxLayer
      input_size: !@ model[embedder][encoder].rnn.hidden_size
      output_size: !@ dataset.label.vocab_size
```

By using `!@` you can link to attributes of previously defined objects. Note that we take `num_embeddings` value from the dataset's vocabulary size that it is stored in its `text` attribute. These are called `Links` (read more about them in *Linking*).

Links always start from the top-level stage in the pipeline, and can even be self-referential, as the second link references the model definition it is a part of:

```
input_size: !@ model[embedder][encoder].rnn.hidden_size
```

Note that the path starts from `model` and the brackets access the embedder and then the encoder in the config file. You can then use dot notation to access the runtime instance attributes of the target object, the encoder in this example.

Always refer to the documentation of the object you're linking to in order to understand what attributes it actually has when the link will be resolved.

Important: You can only link to non-parent objects above the position of the link in the config file, because later objects, and parents of the link, will not be initialized at the time the link is resolved.

Important: Flambé supports native hyperparameter search!

```
n_layers: !g [2, 3, 4]
```

Above we define 3 variants of the model, each containing different amount of `n_layers` in the `encoder`.

Now that we have the dataset and the model, we can add a training process. Flambé provides a powerful and flexible implementation called *`Trainer`*:

```
!Experiment

name: sst
pipeline:

  # stage 0 - Load the dataset object SSTDataset and run preprocessing
  [...]  # Same as before


  # stage 1 - Define the model
  [...]  # Same as before

  # stage 2 - train the model on the dataset
  train: !Trainer
    dataset: !@ dataset
    train_sampler: !BaseSampler
      batch_size: 64
    val_sampler: !BaseSampler
    model: !@ model
    loss_fn: !torch.NLLLoss  # Use existing PyTorch negative log likelihood
    metric_fn: !Accuracy  # Used for validation set evaluation
    optimizer: !torch.Adam
      params: !@ train[model].trainable_params
    max_steps: 20
    iter_per_step: 50
```

Tip: Flambé provides full integration with Pytorch object by using `torch` prefix. In this example, objects like `NLLLoss` and `Adam` are directly used in the configuration file!

Tip: Additionally we setup some `Tune` classes for use with hyperparameter search and scheduling. They can be accessed via `!ray.ClassName` tags. More on hyperparameter search and scheduling in *Experiments*.

2.3 Monitoring the Experiment

Flambé provides a powerful UI called the **Report Site** to monitor progress in real time. It has full integration with Tensorboard.

When executing the experiment (see *Executing Flambé*), flambé will show instructions on how to launch the Report Site.

See also:

Read more about monitoring in *Report Site* section.

2.4 Artifacts

By default, artifacts will be located in `flambe-ouput/` (relative the the current work directory). This behaviour can be overriden by providing a `save_path` parameter to the `Experiment`.

```
flambe-output/output__sst
├── dataset
│   └── 0_2019-07-23_XXXXXX
│       └── checkpoint
│           └── checkpoint.flambe
│               ├── label
│               └── text
├── model
│   ├── n_layers=2_2019-07-23_XXXXXX
│   │   └── checkpoint
│   │       └── checkpoint.flambe
│   │           ├── embedder
│   │           │   ├── embedding
│   │           │   └── encoder
│   │           └── output_layer
│   ├── n_layers=3_2019-07-23_XXXXXX
│   │   └── ...
│   └── n_layers=4_2019-07-23_XXXXXX
│       └── ...
└── trainer
    ├── n_layers=2_2019-07-23_XXXXXX
    │   └── checkpoint
    │       └── checkpoint.flambe
    │           ├── model
    │           │   ├── embedder
    │           │   │   └── ...
    │           │   └── output_layer
    │           └── dataset
    │               └── ...
    ├── n_layers=3_2019-07-23_XXXXXX
    │   └── ...
    └── n_layers=4_2019-07-23_XXXXXX
        └── ...
```

Note that the output is 100% hierarchical. This means that each component is isolated and reusable by itself.

`load()` is a powerful utility to load previously saved objects.

```
import flambe

path = "flambe-output/output__sst/train/n_layers=4_.../.../model/embedder/encoder/"
encoder = flambe.load(path)
```

Important: The output folder also reflects the variants that were speficied in the config file. There is one folder for each variant in `model` and in `trainer`. **The `trainer` inherits the variants from the previous components, in this case the `model`.** For more information on variant inheritance, go to *Search Options*.

2.5 Recap

You should be familiar now with the following concepts

- `Experiments` can be represented in a YAML format where a `pipeline` can be specified, containing different components that will be executed sequentially.
- Objects are referenced using `!` + the class name. Flambé will compile this structure into a Python object.
- Flambé supports natively searching over hyperparameters with tags like `!g` (to perform Grid Search).
- References between components are done using `!@` links.
- The Report Site can be used to monitor the `Experiment` execution, with full integration with Tensorboard.

2.6 Try it yourself!

Here is the full config we used in this tutorial:

Listing 2: simple-exp.yaml

```yaml
!Experiment

name: sst
pipeline:

  # stage 0 - Load the dataset object SSTDataset and run preprocessing
  dataset: !SSTDataset
    transform:
      text: !TextField  # Another class that helps preprocess the data
      label: !LabelField


  # stage 1 - Define the model
  model: !TextClassifier
    embedder: !Embedder
      embedding: !torch.Embedding
        num_embeddings: !@ dataset.text.vocab_size
        embedding_dim: 300
      encoder: !PooledRNNEncoder
        input_size: 300
        rnn_type: lstm
        n_layers: !g [2, 3, 4]
        hidden_size: 256
    output_layer: !SoftmaxLayer
      input_size: !@ model[embedder][encoder].rnn.hidden_size
      output_size: !@ dataset.label.vocab_size

  # stage 2 - train the model on the dataset
  train: !Trainer
    dataset: !@ dataset
    train_sampler: !BaseSampler
      batch_size: 64
    val_sampler: !BaseSampler
    model: !@ model
    loss_fn: !torch.NLLLoss  # Use existing PyTorch negative log likelihood
    metric_fn: !Accuracy  # Used for validation set evaluation
```

(continues on next page)

(continued from previous page)

```
    optimizer: !torch.Adam
      params: !@ train[model].trainable_params
    max_steps: 20
    iter_per_step: 50
```

We encourage you to execute the experiment and to start getting familiar with the artifacts and the report site.

2.7 Next Steps

- *Components*: `SSTDataset`, `Trainer` and `TextClassifier` are examples of *`Component`*. These objects are the core of the experiment's `pipeline`.
- *Runnables*: flambé supports running multiple processes, not just `Experiments`. These objects must implement *`Runnable`*.
- *Clusters*: learn how to create clusters and run remote experiments.
- *Extensions*: flambé provides a simple and easy mechanism to declare custom *`Runnable`* and *`Component`*.
- *Scheduling and Reducing Strategies*: besides grid search, you might also want to try out more sophisticated hyperparameter search algorithms and resource allocation strategies like Hyperband.

CHAPTER 3

Motivation

Flambé's primary objective is to **speed up all of the research lifecycle** including model prototyping, hyperparameter optimization and execution on a cluster.

3.1 Why Flambé?

1. Running machine learning experiments takes a lot of continuous and tedious effort.
2. Standardizing data preprocessing and weights sharing across the community or within a team is difficult.

We've found that while there are several new libraries offering a selection of reliable model implementations, there isn't a great library that couples these modules with an experimentation framework. Since experimentation (especially hyper-parameter search, deployment on remote machines, and data loading and preprocessing) is one of the most important and time-consuming aspects of ML research we decided to build Flambé.

An important component of Flambé is Ray, an open source distributed ML library. Ray has some of the necessary infrastructure to build experiments at scale; coupled with Flambé you could be tuning many variants of your already existing models on a large cluster in minutes! Flambé's crucial contribution is to facilitate rapid iteration and experimentation where tools like Ray and AllenNLP alone require large development costs to integrate.

The most important contribution of Flambé is to improve the user experience involved in doing research, including the various phases of experimentation we outlined at the very beginning of this page. To do this well, we try to adhere to the following values:

3.2 Core values

- **Practicality**: customize functionality in code, and iterate over settings and hyperparameters in config files.
- **Modularity & Composability**: rapidly repurpose existing code for hyper-parameter optimization and new use-cases
- **Reproducability**: reproducible experiments, by anyone, at any time.

CHAPTER 4

Components

The most important class in Flambé is *Component* which implements loading from YAML (using `!ClassName` notation) and saving state.

4.1 Loading and Dumping from YAML

A *Component* can be created from a YAML config representation, as seen the *Quickstart* example.

Lets take the previously used *TextClassifier* component:

Listing 1: model.yaml

```
!TextClassifier
embedder: !Embedder
  embedding: !torch.Embedding
    num_embeddings: 200
    embedding_dim: 300
  encoder: !PooledRNNEncoder
    input_size: 300
    rnn_type: lstm
    n_layers: 3
    hidden_size: 256
output_layer: !SoftmaxLayer
  input_size: 256
  output_size: 10
```

Loading and dumping objects can be done using `flambe.compile.yaml` module.

```
from flambe.compile import yaml

# Loading from YAML into a Schema
text_classifier_schema = yaml.load(open("model.yaml"))
text_classifier = text_classifier_schema()  # Compile the Schema
```

(continues on next page)

(continued from previous page)

```

# Dumping object
yaml.dump(text_classifier, open("new_model.yaml", "w"))
```

Important: `Components` compile to an intermediate state called *`Schema`* when calling `yaml.load()`. This partial representation can be compiled into the final object by calling `obj()` (ie executing `__call__`), as shown in the example above. For more information about this, go to *Delayed Initialization*.

See also:

For more examples of the YAML representation of an object look at understanding-configuration_label

4.2 Saving and Loading State

While YAML represents the "architecture" or how to create an instance of some class, it does not capture the state. For state, `Components` rely on a recursive *`get_state()`* and *`load_state()`* methods that work similarly to PyTorch's `nn.Module.state_dict` and `nn.Module.load_state_dict`:

```
from flambe.compile import yaml

# Loading from YAML into a Schema
text_classifier_schema = yaml.load(open("model.yaml"))
text_classifier = text_classifier_schema()  # Compile the Schema

state = text_classifier.get_state()

from flambe.nlp.classification import TextClassifier

another_text_classifier = TextClassifier(...)
another_text_classifier.load_state(state)
```

Semantic Versioning

In order to identify and describe changes in class definitions, flambé supports opt-in semantic class versioning. (If you're not familiar with semantic versioning see this link).

Each class has a class property `_flambe_version` to prevent conflics when loading previously saved states. Initially, all versions are set to `0.0.0`, indicating that class versioning should not be used. Once you increment the version, Flambé will then start comparing the saved class version with the version on the class at load-time.

See also:

See *Adding Custom State* for more information about *`get_state()`* and *`load_state()`*.

4.3 Delayed Initialization

When you load `Components` from YAML they are not initialized into objects immediately. Instead, they are precompiled into a *`Schema`* that you can think of as a blueprint for how to create the object later. This mechanism allows `Components` to use links and grid search options.

If you load a schema directly from YAML you can compile it into an instance by calling the schema:

```yaml
from flambe.compile import yaml

schema = yaml.load('path/to/file.yaml')
obj = schema()
```

4.4 Core Components

Dataset This object holds the training, validation and test data. Its only requirement is to have the three properties: `train`, `dev` and `test`, each pointing to a list of examples. For convenience we provide a `TabularDataset` implementation of the interface, which can load any `csv` or `tsv` type format.

```yaml
from flambe.dataset import TabularDataset
import numpy as np

# Random dataset
train = np.random.random((2, 100))
val = np.random.random((2, 10))
test = np.random.random((2, 10))

dataset = TabularDataset(train, val, test)
```

Field A field takes raw examples and produces a `torch.Tensor` (or tuple of `torch.Tensor`). We provide useful fields such as `TextField`, or `LabelField` which perform tokenization and numericalization.

```yaml
from flambe.field import TextField
from flambe.tokenizer import WordTokenizer

import numpy as np

# Random dataset
data = np.array(['Flambe is awesome', 'This framework rocks!'])
text_field = TextField(WordTokenizer())

# Setup the entire dataset to build vocab.
text_field.setup(data)
text_field.vocab_size  # Returns to 9

text_field.process("Flambe rocks")  # Returns tensor([6, 1])
```

Sampler A sampler produces batches of data, as an interator. We provide a simple `BaseSampler` implementation, which takes a dataset as input, as well as the batch size, and produces batches of data. Each batch is a tuple of tensors, padded to the maximum length along each dimension.

```yaml
from flambe.sampler import BaseSampler
from flambe.dataset import TabularDataset
import numpy as np

dataset = TabularDataset(np.random.random((2, 10)))

sampler = BaseSampler(batch_size=4)
for batch in sampler.sample(dataset):
    # Do something with batch
```

Module This object is the main model component interface. It must implement the `forward` method as PyTorch's `nn.Module` requires.

We also provide additional machine learning components in the `nn` submodule, such as `Encoder` with many different implementations of these interfaces.

Trainer A *Trainer* takes as input the training and dev samplers, as well as a model and an optimizer. By default, the object keeps track of the last and best models, and each call to run is considered to be an arbitrary of training iterations, and a single evaluation pass over the validation set. It implements the *metric()* method, which points to the best metric observed so far.

Evaluator An *Evaluator* evaluates a given nn`Module over a *Dataset* and computes given metrics.

Script A *Script* integrate a pre-written script with Flambé.

Important: For more detailed information about this `Components`, please refer to their documentation.

4.5 Custom Component

Custom `Components` should implement the *run()* method. This method performs a single computation step, and returns a boolean, indicating whether the `Component` is done executing (`True` iff there is more work to do).

```
class MyClass(Component):

    def __init__(self, a, b):
        super().__init__()
        ...

    def run(self) -> bool:
        ...
        return continue_flag
```

Tip: We recommend always extending from an implementation of `Component` rather than implementing the plain interface. For example, if implementing an autoencoder, inherit from `Module` or if implementing cross validation training, inherit from `Trainer`.

If you would like to include custom state in the state returned by *get_state()* method see the *Adding Custom State* section and the *Component* package reference.

Then in YAML you could do the following:

```
!MyClass
  a: val1
  b: val2

# or using the registrable_factory
```

Flambé also provides a way of registering factory methods to be used in YAML:

```
class MyClass(Component):

    ...

    @registrable_factory
    @classmethod
    def special_factory(cls, x, y):
```

(continues on next page)

(continued from previous page)

```
        a, b = do_something(x, y)
        return cls(a, b)
```

Now you can do:

```
!MyClass.special_factory
  x: val1
  y: val2
```

For information on how to add your custom `Component` in the YAML files, go to *Extensions*

CHAPTER 5

Runnables

Runnables are top level objects that flambé is able to run. *Experiment* is an example of a *Runnable*.

Implementing Runnables is as easy as implementing the Runnable interface. Esentially, it requires a single method *run()*.

Hint: A `Runnable` object can be executed by flambé:

```
flambe runnable.yaml
```

For example, let's imagine we want to implement an **S3Pusher** that takes an `Experiment` output folder and uploads the content to a specific S3 bucket:

```
from flambe.runnable import Runnable

class S3Pusher(Runnable):

    def __init__(self, experiment_path: str, bucket_name: str) -> None:
        super().__init__()
        self.experiment_path = experiment_path
        self.bucket_name = bucket_name

    def run(self, **kwargs) -> None:
        """Upload a local folder to a S3 bucket"""

        # Code to upload to S3 bucket
        S3_client = boto3.client("s3")
        for root, dirs, files in os.walk(self.experiment_path):
            for f in files:
                s3C.upload_file(os.path.join(root, f), self.bucketname, f)
```

This class definition can now be included in an extension (read more about extensions in *Extensions*) and used as a top level object in a YAML file.

```
ext: /path/to/S3Pusher/extension:
---

!ext.S3Pusher
    experiment_path: /path/to/my/experiment
    bucket_name: my-bucket
```

Then, simply execute:

```
flambe s3pusher.yaml
```

5.1 Providing secrets

All `Runnables` have access to secret information that the users can share via an ini file.

By executing the `Runnable` with a `--secrets` parameter, then the `Runnable` can access the secrets through the *`config`* attribute:

Let's say that our `S3Pusher` needs to access the `AWS_SECRET_TOKEN`. Then:

```
from flambe.runnable import Runnable

class S3Pusher(Runnable):

    def __init__(self, experiment_path: str, bucket_name: str) -> None:
        # Same as before
        ...

    def run(self, **kwargs) -> None:
        """Upload a local folder to a S3 bucket"""

        # Code to upload to S3 bucket
        S3_client = boto3.client("s3", token=self.config['AWS']['AWS_SECRET_TOKEN'])
        for root, dirs, files in os.walk(self.experiment_path):
            for file in files:
                s3C.upload_file(os.path.join(root, file), self.bucketname, file)
```

Then if `secrets.ini` contains:

```
[AWS]
AWS_SECRET_TOKEN = ABCDEFGHI123456789
```

We can execute:

```
flambe s3pusher.yaml --secrets secret.ini
```

5.2 Automatic extensions installation

Important: To understand this section you should be familiar with extensions. For information about extensions, go to *Extensions*.

When executing a *`Runnable`*, it's possible that extensions are being involved. For example:

```
ext: /path/to/extension
other_ext: http://github.com/user/some_extension
---

!ext.CustomRunnable
    ...
    param: !other_ext.CustomComponent
```

Flambé provides a `-i / --install-extensions` flag to automatically "pip" installs the extensions:

```
flambe custom_runnable.yaml -i
```

By default, this is **not** activated and the user needs to install the extensions beforehand.

> **Warning: Installing extensions automatically could possibly update libraries in the your environment because of a version reequirement.** Flambé will output all libraries that are being updated.

5.3 Other flags

When executing flambé CLI passing a YAML file, this additional flags can be provided (among others):

`--verbose / -v` All logs will be displayed in the console.

`--force` `Runnables` may chose to accept this flag in its *`run()`* method to provide some overriding policies.

CHAPTER 6

Experiments

The top level object in every configuration file must be a *Runnable*, the most common and useful being the *Experiment* class which facilitates executing a ML pipeline.

The Experiment's most important parameter is the pipeline, where users can define a DAG of Components describing how dataset, models, training procedures, etc interact between them.

> **Attention:** For a full specification of Experiment, see *Experiment*

The implementation of *Experiment* and its *pipeline'* uses Ray's Tune under the hood.

6.1 Pipeline

A pipeline is defined as a list of Components that will be executed **sequentially**. Each *Component* is identified by a **key that can be used for later linking**.

Let's assume that we want to define an experiment that consists on:

1. Pick dataset A and preprocess it.
2. Train model A on dataset A.
3. Preprocess dataset B.
4. Finetune the model trained in 2. on dataset B.
5. Evaluate the fine tuned model on dataset A testset.

All these stages can be represented by a sequential pipeline in a simple and readable way:

```
pipeline:
    dataset_A: !SomeDataset
      ...
```

(continues on next page)

(continued from previous page)

```
    model_A: !SomeModel
       ...

    trainer: !Trainer
       model: !@ model_A
       dataset: !@ dataset_A
       ...

    dataset_B: !Trainer
       ...

    fine_tunning: !Trainer
       model: !@ trainer.model
       dataset: !@ dataset_B
       ...

    eval: !Evaluator
       model: !@ trainer.model
       dataset: !@ dataset_A
```

Note how this represents a DAG where the nodes are the `Components` and the edges are the links to attributes of previouly defined `Components`.

6.2 Linking

As seen before in *Quickstart*, stagegs in the `pipeline` are connected using `Links`.

Links can be used anywhere in the `pipeline` to refer to earlier components or any of their attributes.

During the compilation that is described above in *Delayed Initialization* we actually resolve the links to their intended value, but cache the original link representation so that we can dump back to YAML with the original links later.

6.3 Search Options

`Experiment` supports declaring multiple variants in the `pipeline` by making use of the search tags:

```
!Experiment
...
pipeline:
    ...
    model: !TextClassifier
       ...
       n_layers: !g [2, 3, 4]

    ...
```

The value `!g [2, 3, 4]` indicates that each of the values should be tried. Flambé will create internally 3 variants of the model.

You can specify grid search options search for any parameter in your config, without changing your code to accept a new type of input! (in this case `n_layers` still receives an `int`)

Tip: You can also search over `Components` or even links:

```
!Experiment
...

pipeline:
  dataset: !SomeDataset
    transform:
      text: !g
      - !SomeTextField {{}}  # Double braces needed here
      - !SomeOtherTextField {{}}
```

Types of search options

!g Previously shown. It grids over all its values

```
param: !g [1, 2, 3]  # grids over 1, 2 and 3.
param: !g [0.001, 0.01]  # grids over 0.001 and 0.01
```

!s Yields k values from a range (low, high). If both `low` and `high` are int values, then `!s` will yield int values. Otherwise, it will yield float values.

```
param: !s [1, 10, 5]  # yiels 5 int values from 1 to 10
param: !s [1.5, 2.2, 5]  # yiels 5 float values from 1.5 to 2.2
param: !s [1.5, 2.2, 5, 2]  # yiels 5 float values from 1.5 to 2.2, rounded to 2␣
↪decimals
```

Combining Search tags

Search over different attributes at the same time will have a combinatorial effect.

For example:

```
!Experiment
...
pipeline:
    ...
    model: !TextClassifier
        ...
        n_layers: !g [2, 3, 4]
        hidden_size: !g [128, 256]
```

This will produce 6 variants (3 `n_layers` values times 2 `hidden_size` values)

Variants inheritance

Attention:

Any object that links to an attribute of an object that describes multiple variants will inherit those variants.

```
!Experiment
...
pipeline:
    ...
    model: !TextClassifier
        n_layers: !g [2, 3, 4]
        hidden_size: !g [128, 256]
        ...
    trainer: !Trainer
        model: !@ model
        lr: !g [0.01, 0.001]
        ...
    evaluator: !Evaluator
        model: !@ trainer.model
```

The `trainer` will have 12 variants (6 from `model` times 2 for the `lr`). `eval` will run for 12 variants as it links to `trainer`.

6.4 Reducing

`Experiment` provides a `reduce` mechanism so that variants don't flow down the `pipeline`. **reduce** is declared at the `Experiment` level and it can specify the number of variants to reduce to for each `Component`.

```
!Experiment
...
pipeline:
    ...
    model: !TextClassifier
       n_layers: !g [2, 3, 4]
       hidden_size: !g [128, 256]
    trainer: !Trainer
       model: !@ model
       lr: !g [0.01, 0.001]

    evaluator: !Evaluator
       ...
       model: !@ trainer.model

 reduce:
   trainer: 2
```

Flambé will then pick **the best 2 variants before finishing executing ''trainer''**. This means `eval` will receive the best 2 variants only.

6.5 Resources (Additional Files and Folders)

The `resources` argument lets users specify files that can be used in the *`Experiment`* (usually local datasets, embeddings or other files).

For example:

```
!Experiment
...

resources:
    data: path/to/train.csv
    embeddings: s3://mybucket/embeddings.bin
...
```

In case a resource is a remote URL, then flambé will download the file fow you (relying on the user local permissions)

Attention: Currently S3 and HTTP hosted resources are supported.

`resources` can be referenced in the pipeline via linking:

```
!Experiment
...

resources:
    ...
    embeddings: path/to/embeddings.txt

pipeline:
    ...
      some_field: !@ embeddings
```

Resources in remote experiment

When running remote experiments, all resources will be rsynced into the instances so that they are available in the cluster **unless a "!cluster" tag is specified**.

The `!cluster` tag is useful when the cluster needs to handle the resources. The local process will just ignore those tagged resources.

For example:

```
!Experiment
...

resources:
    data: !cluster path/to/train.csv  # This file is already in all instances of the␣
↪cluster
...
```

When running this example in a cluster, then no `rsync` will be involved as flambé assumes the resource path `path/to/train.csv` exists in all instances of the cluster.

Tip: You can also specify remote URL with the `!cluster` tag:

```
!Experiment
...

resources:
    data: !cluster s3://bucket/data.csv
...
```

In this case the cluster will download the data instead of the local process (if it has permissions to do so)

Attention: The `!cluster` tag is only useful in remote experiments. If the user is running local experiments, using `!cluster` will fail.

6.6 Scheduling and Reducing Strategies

When running a search over hyperparameters, you may want to run a more sophisticated scheduler. Using Tune, you can already use algorithms such as HyperBand, and soon more complex search algorithms like HyperOpt will be available.

```
schedulers:
    b1: !ray.HyperBandScheduler

pipeline:
    b0: !ext.TCProcessor
        dataset: !ext.SSTDataset
    b1: !Trainer
        train_sampler: !BatchSampler
            data: !@ b0.train
            batch_size: !g [32, 64, 128]
        model: ...
    b2: !Evaluator
        model: !@ b1.model
```

6.7 General Logging

We adopted the standard library's logging module for logging:

```
import logging
logger = logging.getLogger(__name__)
...
logger.info("Some info here")
...
logger.error("Something went wrong here...")
```

The best part of the logging paradigm is that you can instantly start logging in any file in your code without passing any data or arguments through your object hierarchy.

Important: By default, only log statements at or above the `INFO` log level will be shown in the console. The rest of the logs will be saved in `~/.flambe/logs` (more on this in *Debugging*)

In order to show all logs in the console, you can use the `--vebose` flag when running flambé:

```
flambe my_config_file.yaml --verbose
```

6.8 Tensorboard Logging

Flambé provides full integration with Tensorboard. Users can easily have data routed to Tensorboard through the logging interface:

```
from flambe import log
...
loss = ... # some calculation here
log('train loss', loss, step)
```

Where the first parameter is the tag which Tensorboard uses to name the value. The logging system will automatically detect the type and make sure it goes to the right Tensorboard function. See *flambe.logging.log()* in the package reference.

Flambé provides also logging special types of data:

- *flambe.logging.log_image()* for images

- *flambe.logging.log_histogram()* for distributions and histograms
- flambe.logging.log_pr_curves() for displaying PR curves
- *flambe.logging.log_text()* for displaying text

See the *logging* for more information on how to use this logging methods.

6.9 Script Usage

If you're using the *flambe.learn.Script* object to wrap an existing piece of code with a command-line based interface, all of the logging information above still applies to you!

See more on Scripts in *Converting a script to Flambé*.

6.10 Checkpointint and Saving

As *Quickstart* explains, flambé saves an *Experiment* in a hierarchical way so that Components can be accessed independant to each other. Specifically, our save files are a directory by default, and include information about the class name, version, source code, and YAML config, in addition to the state that PyTorch normally saves, and any custom state that the implementer of the class may have included.

For example, if you initialize and use the following object as a part of your Experiment:

```
!TextClassifier
embedder: !Embedder
  embedding: !torch.Embedding
    input_size: !@ b0.text.vocab_size
    embedding_size: 300
  encoder: !PooledRNNEncoder
    input_size: 300
    rnn_type: lstm
    n_layers: 2
    hidden_size: 256
output_layer: !SoftmaxLayer
  input_size: !@ b1[model][encoder][encoder].rnn.hidden_size
  output_size: !@ b0.label.vocab_size
```

Then the save directory would look like the following:

```
save_path
├── state.pt
├── config.yaml
├── version.txt
├── source.py
├── embedder
│   ├── state.pt
│   ├── config.yaml
│   ├── version.txt
│   ├── source.py
│   ├── embedding
│   │   ├── state.pt
│   │   ├── config.yaml
│   │   ├── version.txt
│   │   └── source.py
```

(continues on next page)

(continued from previous page)

```
│   └── encoder
│       ├── state.pt
│       ├── config.yaml
│       ├── version.txt
│       └── source.py
└── output_layer
    ├── state.pt
    ├── config.yaml
    ├── version.txt
    └── source.py
```

Note that each subdirectory is self-contained: if it's possible to load that object on its own, you can load from just that subdirectory.

Important: As seen before, each variant of a *Component* will have it's separate output folder.

Note: Flambé will save in this format automatically after each Component of the pipeline executes *run()*. As there are objects that execute *run()* multiple times (for example, *Trainer*), each time the state will be overriden by the latest one (checkpointing).

6.11 Resuming

Experiment has a way of resuming perviously run experiments:

```
!Experiment
resume: trainer
...
pipeline:
    ...
    model: !TextClassifier
        ...
        n_layers: !g [2, 3, 4]
        hidden_size: !g [128, 256]

    trainer: !Trainer
        ...
        model: !@ model
        lr: !g [0.01, 0.001]

    other_trainer: !Trainer
        ...
        model: !@ trainer.model
```

By providing a Component keyname (or a list of them) that belong to the pipeline, then **flambé will resume AFTER all the given blocks, i.e. it would not execute those blocks and continue the experiment after them.**

6.12 Debugging

`Experiment` has a debugging option that is only available in local executions (not remotely). This is activated by adding `debug: True` at the top level of the YAML.

When debugging is on, a debugger will appear before executing `run` on each `Component`.

Warning: Debugging is not enabled when running remote experiments.

6.13 Adding Custom State

Users can add other data to the state that is saved in the save directory. If you just want to have some additional instance attributes added, you can register them at the end of the `__init__` method:

```
class MyModel(flambe.nn.Module):

    def __init__(self, x, ...):
        super().__init__(...)
        ...
        self.x = x,
        self.y = None
        self.register_attrs('x', 'y')
```

This will cause the `get_state` method to start including *x* and *y* in the state dict for instances of `MyModel`, and when you load state into instances of `MyModel` it will know to update these attributes.

If you want more flexibility to manipulate the state_dict or add computed properties you can override the *`_state()`* and *`_load_state()`* methods.

CHAPTER 7

Report Site

The report site is a *control center* website where you are able to:

- View the progress of the experiment
- See live logs
- Download all models [remote]
- Access tensorboard [remote]

Attention: [remote] means that these features are only available in remote experiments.

In local experiments they are not needed.

Flambé's report site comes included in the **Flambé** package. No further setup is required.

Important: The Report Site is currently compatible with `Experiment`.

7.1 How to launch the report site?

Local experiments

When running local experiments, the console output will give you the exact command to run in order to run the report site with the appropriate parameters.

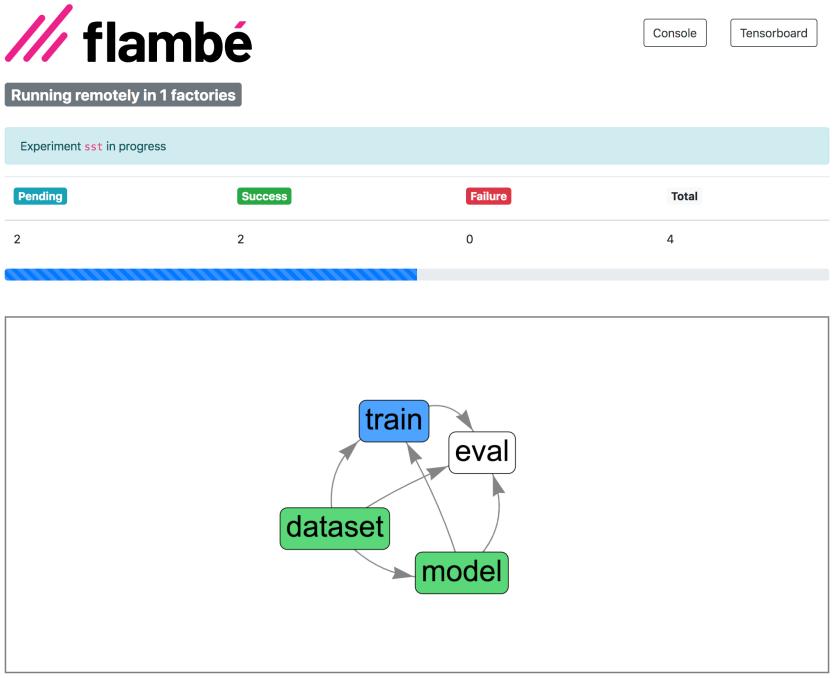
Remote experiments

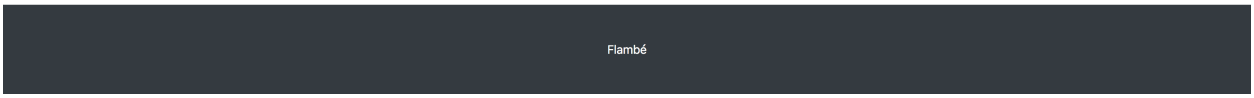
When running remote experiments, the report site will automatically start and you can find the URL in the console output. Remember that it may take a few moments for everything to start up particularly in a remote experiment.

Important: Users are responsible for making ports 49586 (Report Site) and 49556 (Tensorboard)

Screenshots

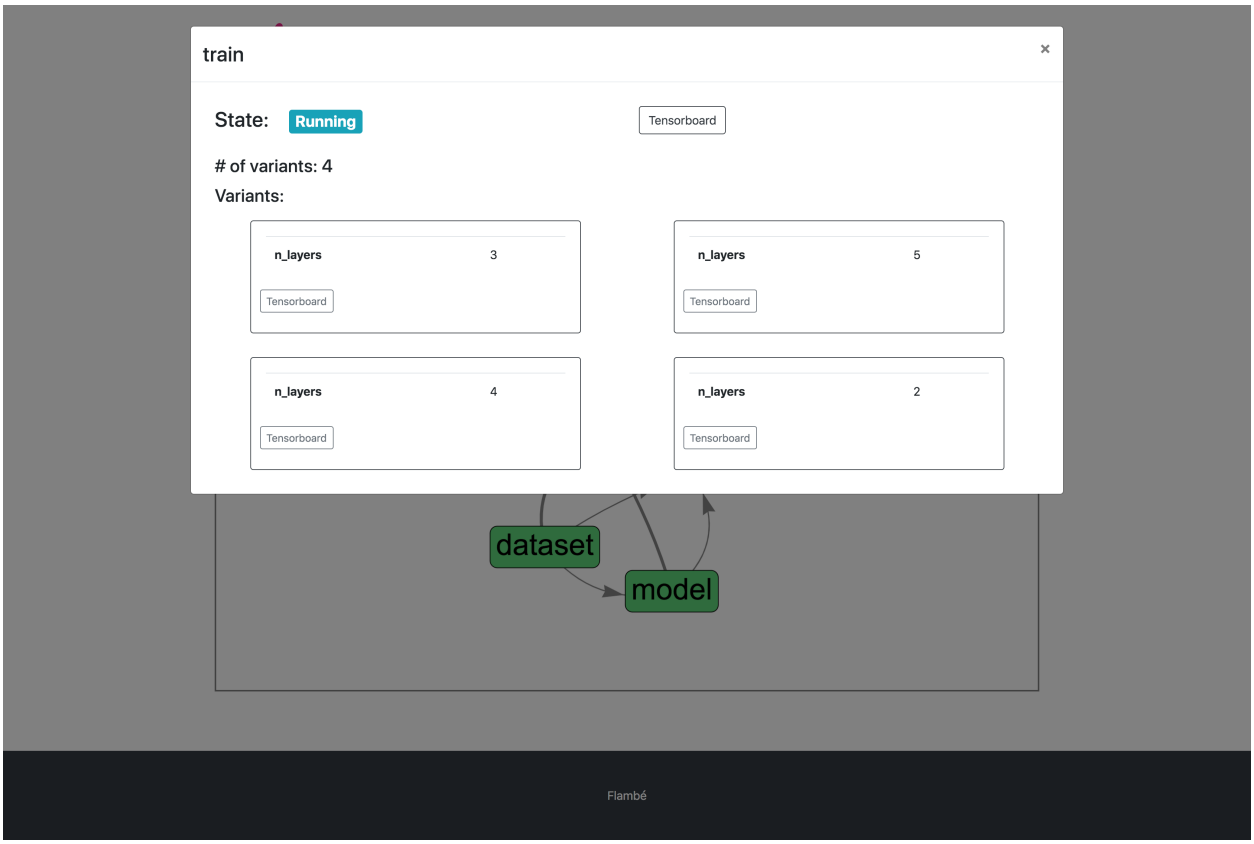
The Report Site displays a real time DAG of the experiment's pipeline.



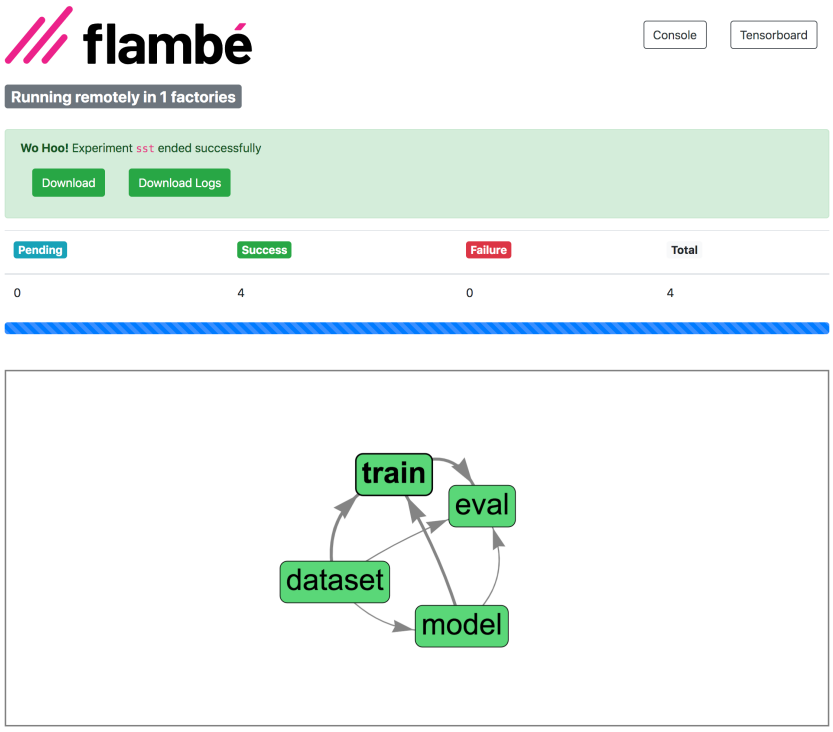


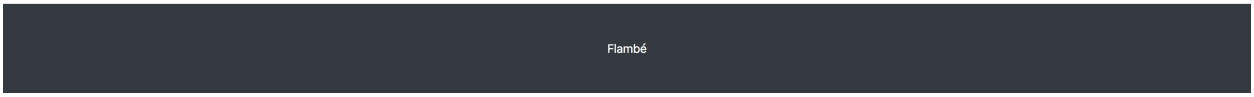
You can access more information by clicking on a block of the DAG.

Hint: The Report Site provides links to Tensorboard with the correct filters already applied. For example, you can access Tensorboard only for the `train` block or even for a specific variant.

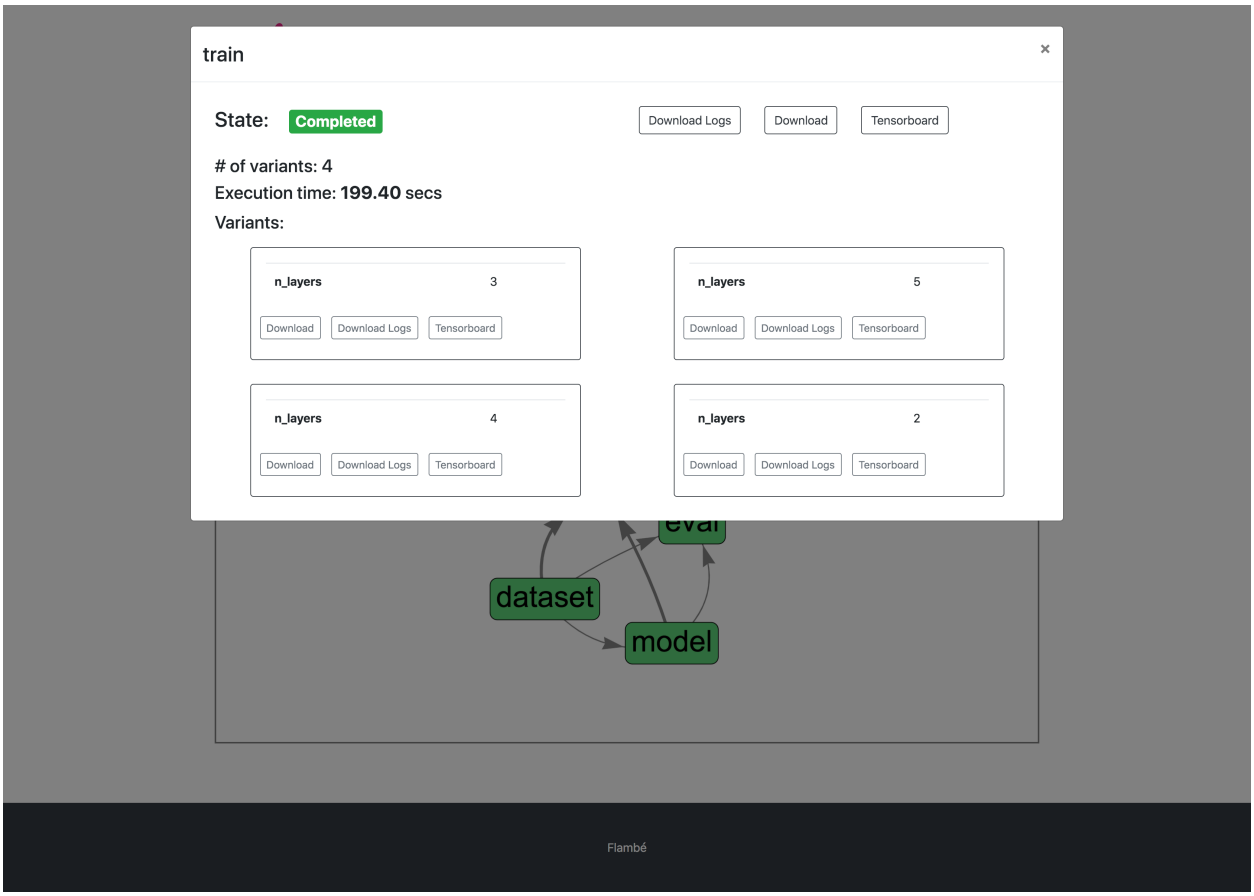


Once the `Experiment` is over, you should see that all blocks are green. In addition, you will be able to download the artifacts (or just the logs).

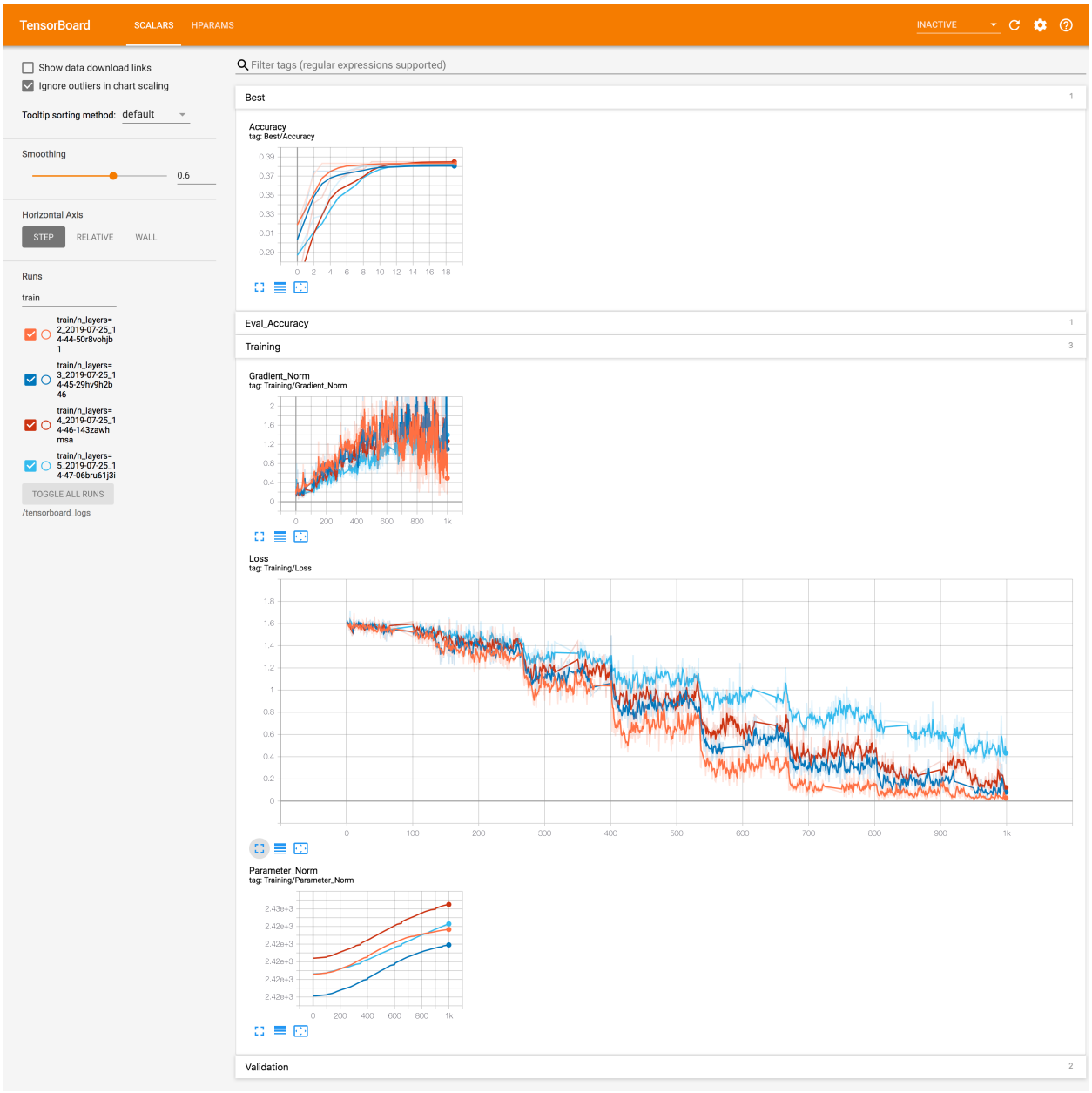




Download can be also done at block or variant level.



The Report Site gives you full integration with Tensorboard:



And it also includes a live console for debugging.


/$$$$$$$ /$$ /$$ /$$ /$
| $$_____/| $$ | $$ /$$
| $$ | $$ /$$$$$$ /$$$$$$/$$$$ | $$$$$$$ /$$$$$$
| $$$$$ | $$ |____ $$| $$_ $$_ $$| $$__ $$ /$$__ $$
| $$__/ | $$ /$$$$$$$| $$ \ $$ \ $$| $$ \ $$| $$$$$$$$
| $$ | $$ /$$__ $$| $$ | $$ | $$| $$ | $$| $$_____/
| $$ | $$| $$$$$$$| $$ | $$ | $$| $$$$$$$/| $$$$$$$
|__/ |__/ _______/|__/ |__/ |__/|_______/ _______/
VERSION: 0.3.0
14:44:34 | Registering tag: !torch.ModuleDict
14:44:34 | Registering tag: !torch.ModuleDict.load_from_path
14:44:35 | Registering tag: !torch.LogSoftmax
14:44:35 | Registering tag: !torch.LogSoftmax.load_from_path
14:44:35 | Registering tag: !torch._Loss
14:44:35 | Registering tag: !torch._Loss.load_from_path
14:44:35 | Registering tag: !torch._MaxPoolNd
14:44:35 | Registering tag: !torch._MaxPoolNd.load_from_path
14:44:35 | Registering tag: !torch.Conv1d
14:44:35 | Registering tag: !torch.Conv1d.load_from_path
14:44:35 | Registering tag: !torch._ReflectionPadNd
14:44:35 | Registering tag: !torch._ReflectionPadNd.load_from_path
14:44:35 | Registering tag: !torch.SELU
14:44:35 | Registering tag: !torch.SELU.load_from_path
14:44:35 | Registering tag: !torch.PairwiseDistance
14:44:35 | Registering tag: !torch.PairwiseDistance.load_from_path
14:44:35 | Registering tag: !torch.RNNCell
14:44:35 | Registering tag: !torch.RNNCell.load_from_path
14:44:35 | Registering tag: !torch.Container
14:44:35 | Registering tag: !torch.Container.load_from_path
14:44:35 | Registering tag: !torch.EmbeddingBag
14:44:35 | Registering tag: !torch.EmbeddingBag.load_from_path
14:44:35 | Registering tag: !torch.Unfold

CHAPTER 8

Extensions

Flambé comes with many built-in *Component* and *Runnable* implementations. (for example *Experiment*, *Trainer*, *TabularDataset* and *BaseSampler*). However, users will almost always need to write new code and include that in their executions. This is done via our **"extensions"** mechanism.

Tip: At this point you should be familiar with the concepts of *Components* and *Runnables*.

Important: The same "extensions" mechanism serves for importing custom `Components` and `Runnables`.

8.1 Building Extensions

An extension is a pip installable package that contains valid Flambé objects as top level imports. These objects could be `Runnables` or `Components`. See here for more instructions on Python packages ready to be pip-installed.

Let's assume we want to define a custom trainer called `MyCustomTrainer` that has a special behavior not implemented by the base `Trainer`. The **extension** could have the following structure:

```
extension
├── setup.py
└── my_ext
    ├── my_trainer.py  # Here lives the definition of MyCustomTrainer
    └── __init__.py
```

Listing 1: extension/setup.py

```
from setuptools import setup, find_packages

setup(
    name='my_extension-pkg-name',
```

(continues on next page)

(continued from previous page)

```
    version='1.0.0',
    packages=find_packages(),  # This will install my_ext package
    install_requires=['extra_dependency==1.2.3'],
)
```

Listing 2: extension/my_ext/__init__.py

```
from my_ext.my_trainer import MyCustomTrainer

__all__ = ['MyCustomTrainer']
```

Listing 3: extension/my_ext/my_trainer.py

```
from flambe.learn import Trainer

class MyCustomTrainer(Trainer):

    ...

    def run(self):
          # Do something special here
```

Attention: If the extension was correctly built you should be able to `pip install` it and execute `from my_ext import import MyCustomTrainer`, which means that this object is at the top level import.

8.2 Using Extensions

You are able to use any extension in any YAML config by specifying it in the `extensions` section which precedes the rest of the YAML:

```
my_extension: /path/to/extension
---
!Experiment
... # use my_extension.MyCustomTrainer and other objects here
```

Each extension is declared using a `key: value` format.

Important: The `key` should be the top-level module name (not the package name).

The `value` can be:

- a local path pointing to the extension's folder (like in the above example)
- a remote GitHub repo folder URLs.
- a PyPI package (alongside its version)

For example:

```
my_extension: /path/to/extension
my_other_extension: https://github.com/user/my_other_extension
```

(continues on next page)

(continued from previous page)

```
another_extension: py-extensions==1.0
---
!Experiment

... # use my_extension.MyCustomTrainer and other objects here
```

Once an extension was added to the `extensions` section, all the extension's objects become available using the module name as a prefix:

```
my_extension: /path/to/extension
my_other_extension: https://github.com/user/my_other_extension
---
pipeline:
    ...

    some_stage: !my_extension.MyCustomTrainer
       ...

    other_stage: !my_other_extension.AnotherCustomObject
       ...
```

Important: Remember to use the **module name** as a prefix

Hint: **We support branches in GitHub extension repositories!** Just use `https://github.com/user/repo/tree/<BRANCH_NAME>/path/to/extension`.

Tip: Using extensions is similar to Python `import` statements. At the top of the file, you declare the non-builtin structures that you wish to use later.

Python	Flambe YAML

`from my_extension import MyCustomTrainer`
 	
`...`
`MyCustomTrainer(...)`
	`my_extension: /path/to/extensions`
	`---`
	`...`
	`!my_extension.MyCustomTrainer`
	`  ...`

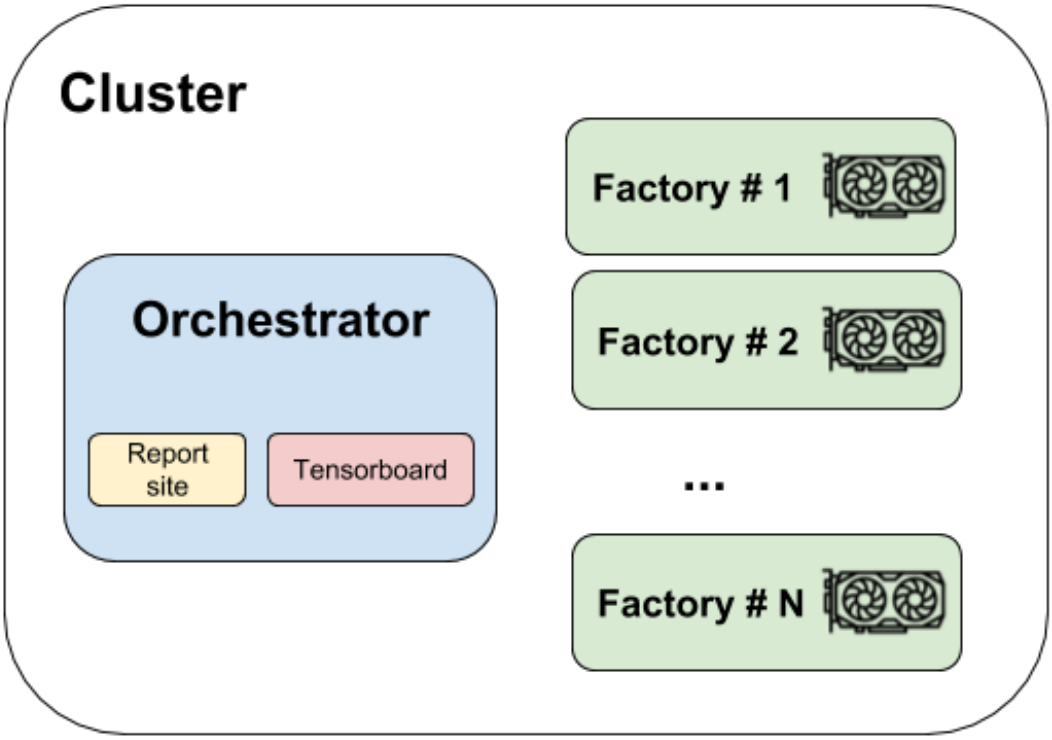
CHAPTER 9

Clusters

Flambé supports running remote `Runnables` where jobs can be distributed across a cluster of workers.

9.1 Overall remote architecture

Flambé will create the following cluster when running a *`Cluster`*:



Orchestrator

The Orchestrator is the main machine in the cluster. The Orchestrator might host websites, run docker containers, etc. It can also collect artifacts like checkpoints or logs.

Attention: This machine doesn't need to contain a GPU as it does not perform heavy computations.

Factories

The factories are instances that are capable of doing heavy computational work and likely need to have GPU resources (for example, if you're running an *Experiment* with PyTorch and CUDA).

Important: Orchestrator and Factories have private SSH connection with a pair of keys that are create and distributed specially for the specific `Cluster`. More information about this *here*.

9.2 Launching a cluster

Cluster is a special type of *Runnable* implementation that handles clusters of machines (e.g. AWS instances) that are capable of running distributed jobs. As with any *Runnable*, you can run a cluster by executing flambé with the YAML config as an argument:

```
flambe cluster.yaml
```

Attention: *Cluster* is an abstract class because it depends on the cloud service provider, so users will need to use one of the provided implementations or create a custom one by overriding the abstract methods.

Important: We currently provide a full cluster implementation for AWS; see *Using AWS*

9.3 Setting the cluster up

All implementations of *Cluster* support setting `setup_cmds`, which are a list of `bash` commands that will run on all instances after creating the cluster:

```
!XXXCluster

name: my_cluster

...

setup_cmds:
  - sshfs user@host:/path/to/remote /path/to/local/mount/point  # Mount a remote␣
↪filesystem
  - pip config set index_url https://my-custom-pypi.com  # Configure PyPI
```

Note that all commands will run sequentially in all the hosts of the cluster.

Tip: This could be useful for mounting volumes, configuring tools or install binaries.

Attention: If you need more complex setup you can also create your own base images for the hosts. *AWSCluster* supports specifying AMIs for both the Orchestrator and factories.

9.4 Submitting Jobs to a Cluster

A cluster is able to run any *ClusterRunnable* implementation, for example *Experiment* (more information ‘ in *Cluster Runnables*).

Given an *experiment.yaml* config file, running it remotely is as easy as:

```
flambe experiment.yaml --cluster cluster.yaml [--force]
```

Flambé will take care of preparing the cluster to run the `ClusterRunnable` (in this case an `Experiment`).

Important: `--force` option is necessary when an existing execution is taking place in the same cluster and the user wants to override it.

Important: There is no need to run `flambe cluster.yaml` before running a `ClusterRunnable` in it. **If it's the first time using the cluster, flambé will create it for you!**

9.5 Using AWS

We provide full AWS integration using the *AWSCluster* implementation. When using this cluster, flambé will take care of:

- Building the cluster
- Preparing all instances (e.g. installing the version of flambé that matches what the user has locally)
- Automatically shutting the cluster down (if specified)

How to use AWSCluster?

A *AWSCluster* is like any flambé *Runnable* and therefore it can be specified in a YAML format:

```
!AWSCluster

name: name-of-the-cluster # Pick a unique identifier for the cluster

factories_num: 1  # The amount of factories

factories_type: g3.4xlarge  # The type of factories. GPU instances are recommended.
orchestrator_type: t3.large # The type of the orchestrator (GPU is not necessary).

orchestrator_timeout: -1  # # -1 means the orchestrator will have to be killed␣
↪manually (recommended)
factories_timeout: -1 # Factories timeout after being unused for these many hours

creator: user@company
key_name: aws-key-name

tags:  # Extra tags to add to all instances
    company: my-company

key: /path/to/ssh/key

subnet_id: subnet-abcdef
volume_size: 100. # GBs of disk space for all instances

security_group: sg-0987654321
```

For a full description, see *flambe.cluster.AWSCluster*.

Automatic shutdown

This *AWSCluster* implementation provides a way of automatically shutting down all instances that have been created:

```
!AWSCluster

# rest of manager config
```

(continues on next page)

(continued from previous page)

```
orchestrator_timeout: 5
factories_timeout: 0
```

These parameters specify **how many hours the resources will persist with low CPU consumption.**

In the above example, the Orchestrator will be terminated after 5 hours of low CPU usage. The Factories will be terminated as soon as CPU usage goes down.

Use `-1` to keep the resources alive permanently, or until you manually stop them.

See also:

For a full example of a configuration file for a Cluster, go here.

9.6 Intelligent versioning

When running `ClusterRunnables` remotely, the correct version of Flambé will be installed automatically, i.e. the version being used locally. For example, if the user has `flambe==1.2` installed locally, then all instances (orchestrator and factories) will be using version `1.2`!

Attention: This is also valid in developer mode. More on developer mode in *Debugging*.

9.7 Cluster Runnables

A *`ClusterRunnable`* is a special implementation of a *`Runnable`* that is able to execute on a flambé cluster.

The *`Experiment`* object, for example, is a *`ClusterRunnable`*.

Users are able to create custom `ClusterRunnables` by implementing its interface (which extends from the `Runnable` interface as well).

This new interface requires an additional implementation for the *`setup()`* method:

```
from flambe.runnable import ClusterRunnable

class MyClusterRunnable(ClusterRunnable):

   def setup(self, cluster: Cluster,
             extensions: Dict[str, str],
             force: bool, **kwargs) -> None:
          # code to setup the cluster
```

The *`setup()`* method should prepare the cluster (which is received as a parameter) to run the *`Runnable`* remotely. This usually involves creating folders, downloading resources, running docker containers, etc.

Important: All *`Cluster`* implementations provides basic functionality that allow directory creation, running bash commands, rsyncing folders, running docker containers and much more. See its documentation for more information about this.

Hint: It's highly likely that you will need to change some instance attributes in the object in the *setup()* method. For doing this, you should use *set_serializable_attr()* to ensure that the attribute change is serializable.

How to run a ClusterRunnable

For running a *ClusterRunnable* remotely, you will need to provide a cluster configuration:

```
flambe cluster_runnable.yaml --cluster cluster.yaml
```

Because of being *Runnable*, it can still be executed locally:

```
flambe cluster_runnable.yaml
```

9.8 Remote Experiments

Users can get the most of performance by running `Experiments` in a `Cluster`.

In remote `Experiments`, a **ray cluster** will be created connecting all instances in the cluster. The Orchestrator will host Tensorboard and the Report Site (the URL will be provided in the console) and the Factories will do the heavy work executing the `pipeline`.

Additionally, when running remote `Experiments`, flambé will take care of uploading the local resources that were specified, making them available to all instances.

CHAPTER 10

Builders

A *Builder* is a simple *Runnable* that can be used to create any *Component* post- *Experiment*, and export it to a local or remote location.

`Builders` decouple the inference logic with the experimentation logic, allowing users to iterate through inference contracts independently without needing to rerun an *Experiment*.

Hint: A *Builder* should be used to build inference engines that rely on previous experiments' artifacts.

10.1 Motivation

Let's assume that a user wants to train a binary classifier using an *Experiment*:

```
ext: /path/to/my/extensions
---
!Experiment
..
pipeline:
    ...
    model: !ext.MyBinaryClassifier
```

Now, the user needs to implement an inference object `ClassifierEngine` that has a method `predict` that performs the forward pass on the trained `model`:

```
from flambe.nn import Module
from flambe.compile import Component

class ClassifierEngine(Component):

   def __init__(self, model: Module):
      self.model = model
```

(continues on next page)

(continued from previous page)

```

   def predict(self, **kwargs):
      # Custom code (for example, requests to APIs)
      p = self.model(feature)
      return {"POSITIVE": p, "NEGATIVE": 1-p}
```

By implementing *Component*, the user can use a *Builder* to build this object:

```
ext: /path/to/my/extensions
---
!Builder

storage: s3
destination: my-bucket

..
component: !ClassifierEngine
    ...
    model: !ext.MyBinaryClassifier.load_from_path:
      path: /path/to/saved/modeel
```

The inference object will be saved in `s3://my-bucket`. Then the user can:

```
import flambe

inference_engine = flambe.load("s3://my-bucket")
inference_engine.predict(...)
# >> {"POSITIVE": 0.9, "NEGATIVE": 0.1}
```

Important: Note that the inference logic is decoupled from the *Experiment*. If in the future the inference logic changes, there is no need of rerunning it.

Note: Why not just implement a plain Python class and use *flambe.compile.serialization.load()* to get the model? Because of being a *Component*, this object will have all the features *Component* has (YAML serialization, versioning, compatibility with other *Runnable* implementations, among others).

10.2 How to use a builder

Usage is really simple. The most important parameters for a *Builder* are the *Component* and the destination:

```
!Builder

storage: [ local | s3 ]
destination: path/to/location

..
component: !MyComponent
    params1: value1
    params2: value2
    ...
    paramsN: valueN
```

Important: For a full list of parameters, go to *Builder*.

Hint: If storage is **"s3"**, then the destination can be an S3 bucket folder. Flambé will take care of uploading the built artifacts.

10.3 Future Work

The goal is to develop builders for different technologies. For example, a `DockerBuilder` that is able to build a Docker container based on a *Component*.

CHAPTER 11

Security

When creating clusters, sending information to instances or even pulling extensions from GitHub, flambé needs to deal with user's protected data.

Important: Flambé will always rely on the default local configuration users have for all services flambé uses. This means that when using services like github or AWS, flambé will rely on the standard authentication mechanisms each service requires.

There is no need for users to configure special auth mechanisms for flambé.

11.1 Secrets

As explained in *Providing secrets*, users have the posibility of providing secrets information to the *Runnable* objects that will be executed.

This is done via an ini file. For example:

```
[SERVICE]
SECRET_TOKEN = ABCDEFGHI123456789


[OTHER]
PASSWORD = 0987654321
```

When calling:

```
flambe runnable.yaml --secrets secrets.ini [--cluster cluster.yaml]
```

Then the *Runnable* will have access to the secrets through its attribute *config*.

11.2 Clusters

Some important items related to Security when dealing with clusters:

- Flambé will use SSH for all communications with the instances. It will use **only** the key specified in the config.
- All resources that need to be uploaded to the instances are done via **rsync with the given key**.
- A copy of the secrets file will be sent to all instances using **rsync with the given key**.
- **The key provided in the config is never uploaded to the instances.**
- When loading the cluster, flambé will distribute a special key pair (created exclusively for the cluster) to all instances. This key will be used for internal communication (ie the communication betweeen the instances)

Important: All flow between the local process and the instances is done in a secure way using SSH protocols.

Attention: Flambé won't configure any instances security policies (eg firewalls, security groups, etc). The user is responsible for configuring this to ensure clusters work correctly. This involves:

- Allowing private communication between the hosts created in the same subnet.
- Opening port 22 for SSH connection in all hosts.
- Opening ports 49556 and 49558 for the Report Site (in case of running an *`Experiment`*).

11.2.1 AWS

When using *`AWSCluster`* for creating an AWS EC2 cluster, flambé will rely on the local configuration for authentication (it uses `boto3` under the hood). Users are going to be able to create clusters as long as they follow the AWS standards for storing credentials/tokens (for example, having `~/.aws/credentials` file or having `AWS_*` environment variables defined).

11.3 Extensions

As explained in *Automatic extensions installation*, flambé will install the extensions when `-i` is specified. For all extensions that are git based URLs (from GitHub or BitBucket for example), then **flambé will try to clone/pull them using the local configuration**. This means that if for example a user wants to use an extensions from its private GitHub account, then it needs to have local configuration that allows pulling from this GitHub account. Flambé will not provide any special SSH keys to authenticate with these services.

Hint: git URLs for extensions support both HTTPS/SSH protocols:

```
extensions: ssh://git@github.com:user/repo.git
other_extension: https://github.com/user/repo/tree/my_branch/extensions
---
!Runnable
...
```

CHAPTER 12

Advanced

12.1 Developer Mode

By using `pip install -e .`, you enable the developer mode, which allows you to use Flambé in editable mode.

By installing in developer mode (see starting-install-dev_label) Flambé will automatically use the current code in your local copy of the repo that you installed, including remote experiments.

12.2 Cache Git-based Extensions

As explained in *Automatic extensions installation*, flambé will install the extensions when `-i` is specified.

For all extensions that are git based URLs (from GitHub or BitBucket for example), flambé will clone the repositories into `~/.flambe/extensions/` folder the first time those extensions are being used. After this, every time one of those extensions is being used flambé will pull instead of cloning again.

This allows `Runnables` to install extensions much faster in case they are heavy sized git repos.

> **Attention:** Flambé will warn once the size of `~/.flambe/extensions/` get bigger than 100MB.

12.3 Debugging

Flambé will save all logs in `~/.flambe/logs` folder using a rotating mechanism.

> **Hint:** The latest logs will be available in `~/.flambe/logs/log.log`.

12.4 Custom YAML Tags

12.4.1 Aliases

Sometimes the best name for a class isn't the best or most convenient name to use in a YAML config file. We provide an *alias()* class decorator that can give your class alternative aliases for use in the config.

Usage

```
from flambe.compile import alias

@alias('cool_tag')
class MyClass(...):
    ...
```

Then start using your class as `!cool_tag` instead of `MyClass` in the config. Both options will still work though. This combines seemlessly with extensions namespaces; if your extension's module name is "ext" then the new alias will be `!ext.cool_tag`.

12.4.2 Registrables

While you will normally subclass *Component* to use some class in a YAML configuration file, there may be situations where you don't want all the functionality described in :ref:'understanding-component_label' such as delayed initialization, and recursive compilation. For these situations you can instead subclass the *Registrable* class which only defines the necessary functionality for loading and dumping into YAML. You will have to implement your own *from_yaml()* and *to_yaml()* methods.

Example

Let's say you want to create a new wrapper class around an integer that tracks its name and a history of its values. First you would have to write your class

```
from flambe.compile import Registrable

class SmartInt(Registrable):

    def __init__(self, name: str, initial_value: int):
        self.name = name
        self.initial_value = initial_value  # For dumping later
        self.val = initial_value

    ...  # Rest of implementation here
```

Then you'll want to implement your `from_yaml` and `to_yaml` in a way that makes sense to you. Here, let's say the name and initial value should be separated by a dash character:

```
@classmethod
def to_yaml(cls, representer: Any, node: Any, tag: str) -> Any:
    str_rep = f"{self.name}-{self.val}"
    representer.represent_str(tag, str_rep)

@classmethod
def from_yaml(cls, constructor: Any, node: Any, factory_name: str) -> Any:
    str_rep = constructor.construct_str(node)
    name, initial_value = str_rep.split()
    return cls(name, initial_value)
```

Finally you can now use your new Registrable object in YAML.

```
!Experiment
...
pipeline:
  stage_0: !Trainer
    param: !SmartInt my_param-9
```

Attention: You will need to make sure your code is part of an extension so that Flambé knows about your new class. See *Extensions*

See also:

The official ruamel.yaml documentation for information about `from_yaml` and `to_yaml`

See also:

`MappedRegistrable` can be referenced as another example or used if you just want a basic `Registrable` that can load from a dictionary of kwargs but doesn't have the other features of *`Component`* like delayed initialization

CHAPTER 13

Converting a script to Flambé

In many situations, you may already have a script that performs your full training routine, and you would like to avoid doing any intergation work to leverage some of the tools in Flambé, namely launching variants of the script on a large cluster, and using the Flambé logging system.

For this particular use case, Flambé offers a `Script` component, which takes as input an executable module (your script), and the relevant arguments. The script should read arguments from `sys.argv`, which means that traditional scripts, and scripts that use tools such as argparse are supported.

Attention: The `Script` object only consists of a single step, and is therefore not compatible with checkpointing or trial schedulers such as Hyperband. It is however possible to use with hyperparameter search algorithms.

13.1 Wrapping your script in a pip installable

Say you have the following directory structure for your project:

```
my_project
├── model.py
├── processing.py
└── train.py
```

Where `train.py` is your target script which uses `argparse` to read arguments. The first step is to convert your project into a pip installable.

```
my_pip_installable
├── setup.py
└── my_project
    ├── __init__.py
    ├── model.py
    ├── processing.py
    └── train.py
```

Your setup should contain all of the external library requirements, used by your script. Once this is done, you should make an attempt at running your script using the `-m` argument, which will treat `train.py` as an executable module. You can do so by running:

```
python -m my_project.train --arg1 value1 --arg2 value2
```

Attention: You will need to modify the imports in your script to use either relative imports or import from the top level package. In this example, this can be done by replacing `import model` by import `.model`. Note that you only need to perform this change once and will still be able to run your script, normally, regardless of Flambé, using `python -m`.

13.2 Writing a config file

Once you have done the above step, you can use your script in Flambé as follows:

```
my_project: /path/to/my_pip_installable

---

!Experiment

name: example

pipeline:
  stage_0: !Script
    script: my_project.train  # my_project is the name of the module
    args:
      arg1: !g [1, 5]  # Run a grid search over any arguments to your script
      arg2: 'foo'
```

That's it! You can now execute this configuration file, with the regular command:

In order to see tensorboard logs, simply import the logger, and use it anywhere in your script:

CHAPTER 14

Using Custom Code in Flambé

While Flambé offers a large number of *Component* objects to use in experiments, researchers will typically need to use their own code, or modify one of our current component object.

14.1 Writing your custom code

Flambé configurations support any python object that inherits from *Component*. You can decide to inherit from one of our base classes such as *Module* or *Dataset*, or you can inherit from *Component* directly.

A *Component* must simply implement a *run()* method which returns a boolean indicating whether execution should continue or not (useful for multi-step components such as a *Trainer*).

Additionally, if you would like to run hyperparameter search on your custom component, you must implement the *metric()* method which returns the current best metric.

14.2 Setting up your extension

Flambé offers a simple mechanism to inject custom code in configs. To do so, your code must be wrapped in a pip installable. If you are not familiar with python packages, you can follow these 3 simple steps:

1. Make sure all your code is organised in a python module. In the example below, `my_module` is the name of the module containing our custom model and dataset objects.

```
my_module # The name of your module
├── model.py
├── dataset.py
...
```

2. Next, we wrap the module in another folder, representing our package.

```
my_package # This is the name of your package
└── my_module # This is the name of your module
    ├── __init__.py
    ├── model.py
    ├── dataset.py
    └── ...
```

3. Finally, we write our setup.py file, which will make our package installable. This file is crucial as it indicates the external dependencies of your custom code. Below is a template for your setup file.

```
from setuptools import setup, find_packages

setup(
    name = 'my_package', # the name of your package
    version = '1.0.0', # the version of your extension (optional)
    packages = find_packages(),
    install_requires = ['numpy >= 1.11.1', 'matplotlib >= 1.5.1'], # Dependencies here
)
```

After you add the setup file to your package, your final folder structure should look like the one below:

```
my_package
├── setup.py # This file makes your package installable
└── my_module
    ├── __init__.py
    ├── model.py
    ├── dataset.py
    └── ...
```

14.3 Using your extension

You have built your first extension! You can now use it freely in any configuration, whether that'd be for an *Experiment*, a *Cluster* or any other *Runnable*.

To do so, simply add them at the top of your extension, mapping the name of the module your built (`my_module` in the example) to the location of the package (`my_package` in the example).

Important: The name of the module is used as a prefix in your configurations. Not the name of your package.

```
my_module: path/to/my_package  # Must map from module to package path

---  # Note the 3 dashes here

!Experiment

pipeline:
    dataset: !my_module.MyDataset # We use the name of your custom module as prefix
```

Tip: The path to the package may be a local path, a github URL, or the name of package one pypi. The latter allows you to specify a specific version of your extension. For github, we also support links to specific commit or branches.

Flambé will require your extension to be installed. You can do so manually by running:

```
pip install my_package
```

or Flambé can install all the extensions specified in your configuration automatically while executing your config when using the `-i` flag:

```
flambe -i config.yaml
```

CHAPTER 15

Writing a multistage pipeline: BERT Fine-tuning + Distillation

It is common to want to train a model on a particular task, and reuse that model or part of the model in a fine-tuning stage on a different dataset. Flambé allows users to link directly to objects in a previous stage in the pipeline without having to run two different experiments (more information on linking *here*)

For this tutorial, we look at a recent use case in natural language processing, namely fine-tuning a BERT model on a text classification task, and applying knowledge distillation on that model in order to obtain a smaller model of high performance. Knowledge distillation is interesting as BERT is relativly slow, which can hinder its use in production systems.

15.1 First step: BERT fine-tuning

We start by taking a pretrained BERT encoder, and we fine-tune it on the *SSTDataset* by adding a linear output layer on top of the encoder. We start with the dataset, and apply a special TextField object which can load the pretrained vocabulary learned by BERT.

The *SSTDataset* below inherits from our *TabularDataset* component. This object takes as input a `transform` dictionary, where you can specify *Field* objects. A *Field* is considered a featurizer: it can take an arbitrary number of columns and return an any number of features.

Tip: You are free to completely override the *Dataset* object and not use *Field*, as long as you follow its interface: *Dataset*.

In this example, we apply a *PretrainedTransformerField* and a *LabelField*.

```
dataset: !SSTDataset
    transform:
        text: !PretrainedTransformerField
            alias: 'bert-base-uncased'
        label: !LabelField
```

Tip: By default, fields are aligned with the input columns, but one can also make an explicit mapping if more than one feature should be created from the same column:

```
transform:
    text:
        columns: 0
        field: !PretrainedTransformerField
            alias: 'bert-base-uncased'
    label:
        columns: 1
        field: !LabelField
```

Next we define our model. We use the *TextClassifier* object, which takes an *Embedder*, and an output layer. Here, we use the *PretrainedTransformerEmbedder*

```
teacher: !TextClassifier

  embedder: !PretrainedTransformerEmbedder
    pool: True

  output_layer: !SoftmaxLayer
    input_size: !@ teacher[embedder].hidden_size
    output_size: !@ dataset.label.vocab_size  # We link the to size of the label space
```

Finally we put all of this in a *Trainer* object, which will execute training.

Note: Recall that you can't link to parent objects because they won't be initialized yet; that's why we link directly to the embedder via bracket notation (it will be initialized because it's above in the config and not a parent), and access the intended `hidden_size` attribute

Tip: Any component can be specified at the top level in the pipeline or be an argument to another *Component* objects. A *Component* has a run method which for many objects consists of just a `pass` statement, meaning that using them at the top level is equivalent to declaring them. The *Trainer* however executes training through its run method, and will therefore be both declared and executed.

```
finetune: !Trainer
  dataset: !@ dataset
  train_sampler: !BaseSampler
    batch_size: 16
  val_sampler: !BaseSampler
    batch_size: 16
  model: !@ teacher
  loss_fn: !torch.NLLLoss
  metric_fn: !Accuracy
  optimizer: !AdamW
    params: !@ finetune[model].trainable_params
    lr: 0.00005
```

15.2 Second step: Knowledge distillation

We now introduce a second model, which we will call the student model:

```
student: !TextClassifier

  embedder: !Embedder
    embedding: !Embeddings
      num_embeddings: !@dataset.text.vocab_size
      embedding_dim: 300
    encoder: !PooledRNNEncoder
      input_size: 300
      rnn_type: sru
      n_layers: 2
      hidden_size: 256
    pooling: !LastPooling
  output_layer: !SoftmaxLayer
    input_size: !@ student[embedder][encoder].hidden_size
    output_size: !@ dataset.label.vocab_size
```

Attention: Note how this new model is way less complex than the original layer, being more appropriate for productions systems.

In the above example, we decided to reuse the same embedding layer, which allows us not to have to provide a new *Field* to the dataset. However, you may also decide to perform different preprocessing for the student model:

```
dataset: !SSTDataset
  transform:
    teacher_text: !PretrainedTransformerField
      alias: 'bert-base-uncased'
      lower: true
    label: !LabelField
    student_text: !TextField
```

We can now proceed to the final step of our pipeline which is the *DistillationTrainer*. The key here is to link to the teacher model that was obtained in the `finetune` stage above.

Tip: You can specify to the *DistillationTrainer* which columns of the dataset to pass to the teacher model, and which to pass to the student model through the `teacher_columns` and `student_columns` arguments.

```
distill: !DistillationTrainer
  dataset: !@ dataset
  train_sampler: !BaseSampler
    batch_size: 16
  val_sampler: !BaseSampler
    batch_size: 16
  teacher_model: !@ finetune.model
  student_model: !@ student
  loss_fn: !torch.NLLLoss
  metric_fn: !Accuracy
  optimizer: !torch.Adam
    params: !@ distill[student_model].trainable_params
    lr: 0.00005
```

(continues on next page)

(continued from previous page)

```
  alpha_kl: 0.5
  temperature: 1
```

Attention: Linking to the teacher model directly would use the model pre-finetuning, so we link to the model inside the `finetune` stage. Note that for these links to work, it's important for the *`Trainer`* object to have the `model` as instance attribute.

That's it! You can find the full configuration below.

15.3 Full configuration

```
!Experiment

name: fine-tune-bert-then-distill
pipeline:

  dataset: !SSTDataset
    transform:
      text: !PretrainedTransformerField
        alias: 'bert-base-uncased'
      label: !LabelField

  teacher: !TextClassifier
    embedder: !PretrainedTransformerEmbedder
      alias: 'bert-base-uncased'
      pool: True
    output_layer: !SoftmaxLayer
      input_size: !@ teacher[embedder].hidden_size
      output_size: !@ dataset.label.vocab_size  # We link the to size of the label␣
↪space

  student: !TextClassifier
    embedder: !Embedder
      embedding: !Embeddings
        num_embeddings: !@ dataset.text.vocab_size
        embedding_dim: 300
      encoder: !PooledRNNEncoder
        input_size: 300
        rnn_type: sru
        n_layers: 2
        hidden_size: 256
      pooling: last
    output_layer: !SoftmaxLayer
      input_size: !@ student[embedder][encoder].hidden_size
      output_size: !@ dataset.label.vocab_size

  finetune: !Trainer
    dataset: !@ dataset
    train_sampler: !BaseSampler
      batch_size: 16
    val_sampler: !BaseSampler
      batch_size: 16
```

(continues on next page)

(continued from previous page)

```
    model: !@ teacher
    loss_fn: !torch.NLLLoss
    metric_fn: !Accuracy
    optimizer: !AdamW
      params: !@ finetune[model].trainable_params
      lr: 0.00005

  distill: !DistillationTrainer
    dataset: !@ dataset
    train_sampler: !BaseSampler
      batch_size: 16
    val_sampler: !BaseSampler
      batch_size: 16
    teacher_model: !@ finetune.model
    student_model: !@ student
    loss_fn: !torch.NLLLoss
    metric_fn: !Accuracy
    optimizer: !torch.Adam
      params: !@ distill[student_model].trainable_params
      lr: 0.00005
    alpha_kl: 0.5
    temperature: 1
```

CHAPTER 16

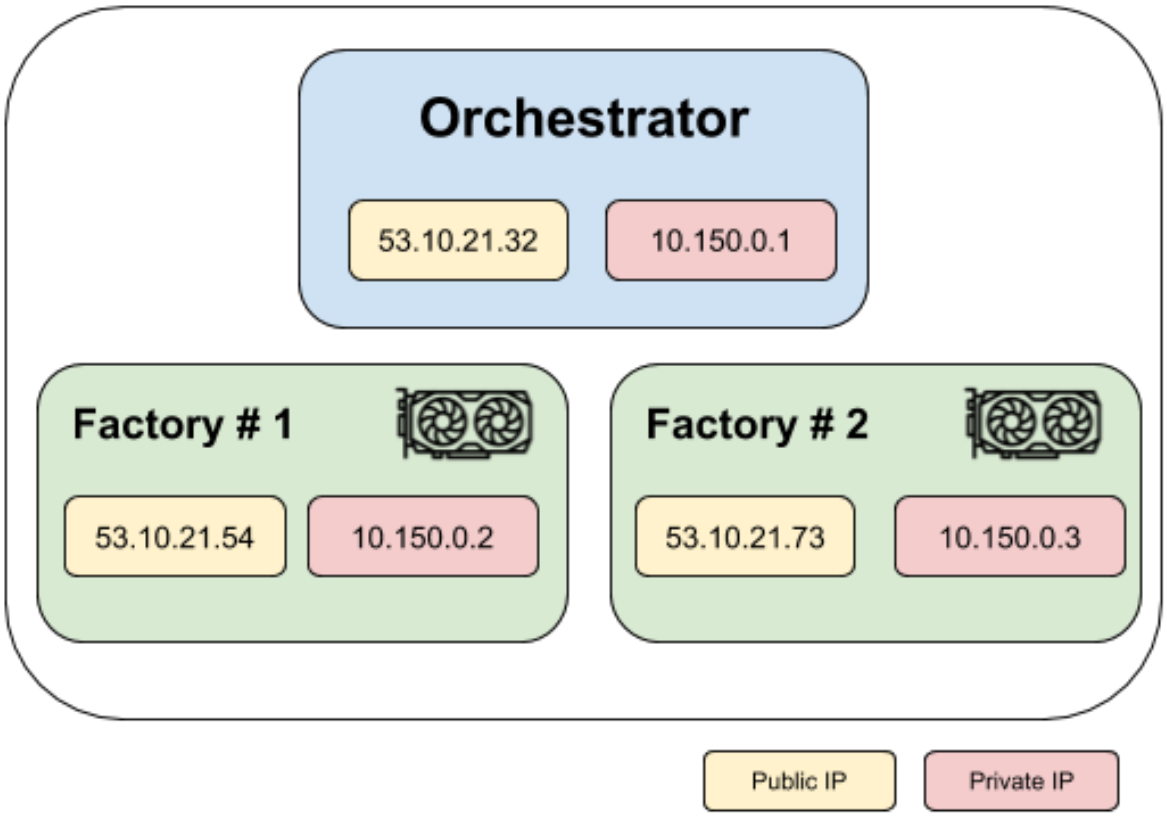
Creating a cluster with existing instances

Flambé provides a `Cluster` implementation called `SSHCluster` that is able to build a cluster from existing instances.

Important: As described in *Clusters*, all clusters have an orchestrator host and a set of factories hosts.

16.1 Instances in a cloud service provider

Let's assume that the user contains the following cluster:



Tip: It's not required that the factories contain GPU.

Important: It is required that:

- All instances are in same private LAN.
- All host have the same username.
- All host are accessible with the same private key.

Implementing an *SSHCluster* is as simple as:

Listing 1: ssh-cluster.yaml

```
!SSHCluster

name: my-cluster

orchestrator_ip: [53.10.21.32, 10.150.0.1]
factories_ips:
  - [53.10.21.54, 10.150.0.2]
  - [53.10.21.73, 10.150.0.3]

key: /path/to/my/key
```

(continues on next page)

(continued from previous page)

```
username: ubuntu
```

Note that all hosts have information about both the public IP and the private IP.

16.2 Instances in the private LAN

If the instances do not have a public IP because they are running on-premise, then *SSHCluster* supports providing private IPs only.

For example:

Listing 2: ssh-cluster.yaml

```
!SSHCluster

name: my-cluster

orchestrator_ip: 10.150.0.10
factories_ips:
  - 10.150.0.20
  - 10.150.0.30

key: /path/to/my/key

username: ubuntu
```

16.3 More information

Refer to the *Clusters* section or checkout the documentation of *SSHCluster*.

CHAPTER 17

Creating a cluster using Amazon Web Services (AWS)

AWSCluster is a *Cluster* implementation that uses AWS as the cloud service provider.

This tutorial will guide you step by step to create your first AWS-based cluster in flambé.

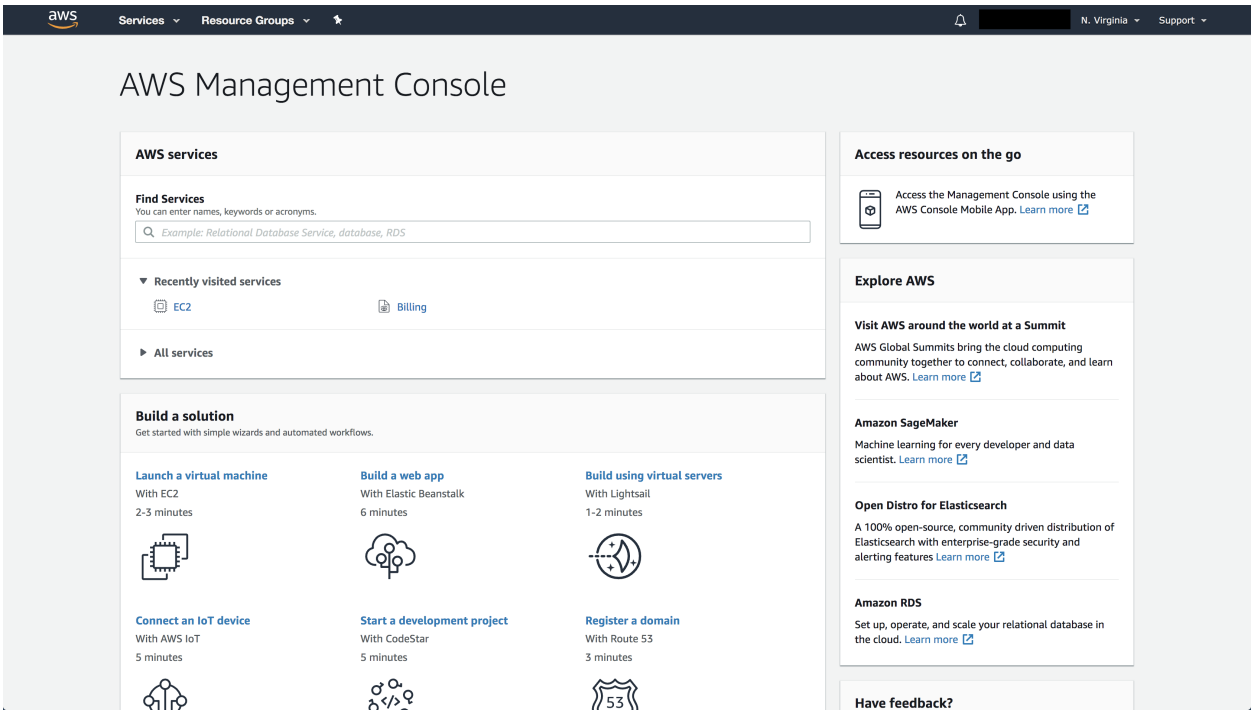
17.1 Setting up your AWS account

Important: If you are already familiar with AWS main concepts (Subnets, type of instances, security groups, etc) and you have your AWS account set up, then feel free to skip this section. **Consider that your Account should be able to:**

- Have a key pair to access instances.
- Create instances with automatic public IPs.
- Connect through SSH from the outside world.
- Have the security credentials and configuration files locally.

If any of this requirements is not met, please review the following steps.

You will first need to create your AWS account here. Once done, go into the console (https://console.aws.amazon.com). You should see something like:



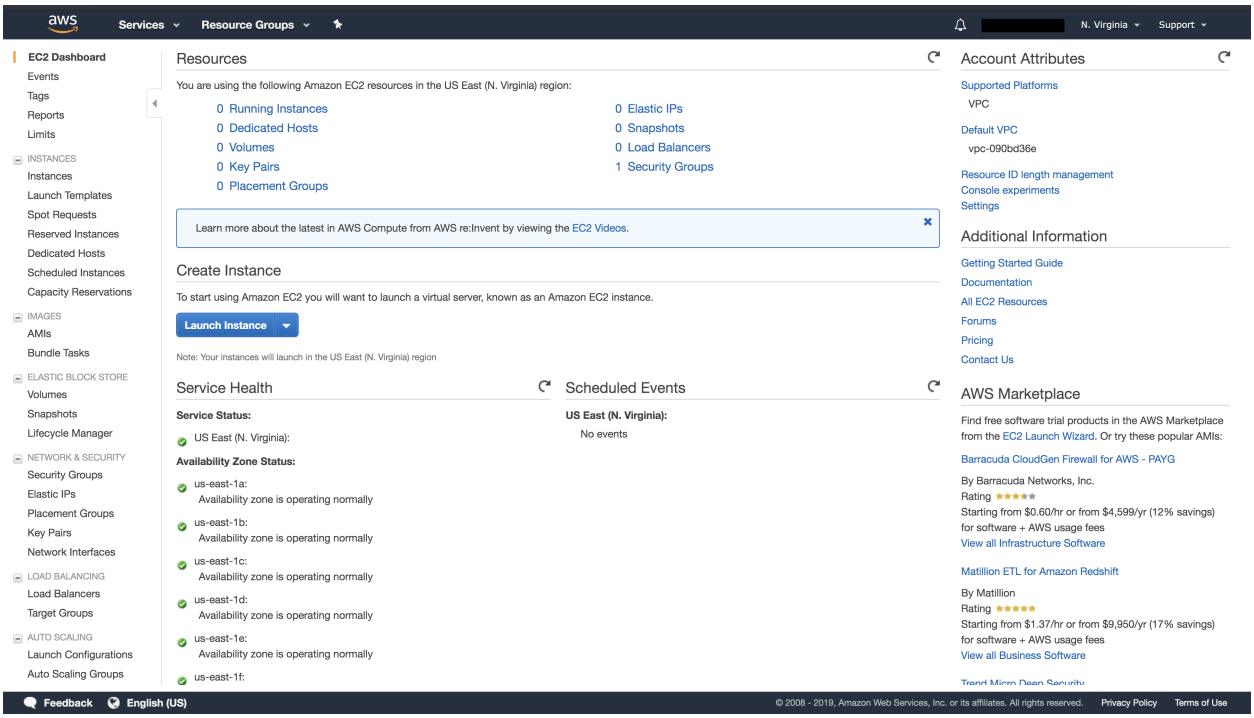
Attention: AWS provides a free tier. If users use this option, the `timeout` feature may not be available and only basic CPU instances are going to be available.

17.1.1 Create key-pair

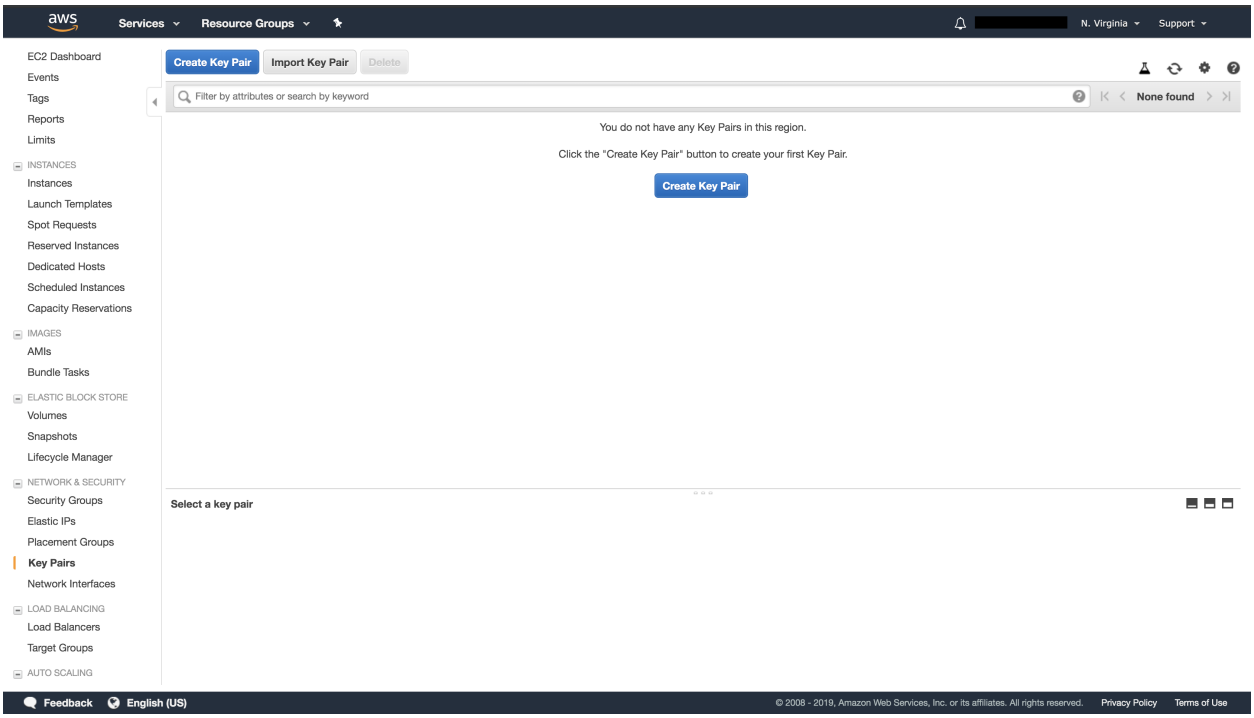
Important: If you already have a key pair feel free to ignore this section.

A key pair will be used to communicate with the instances.

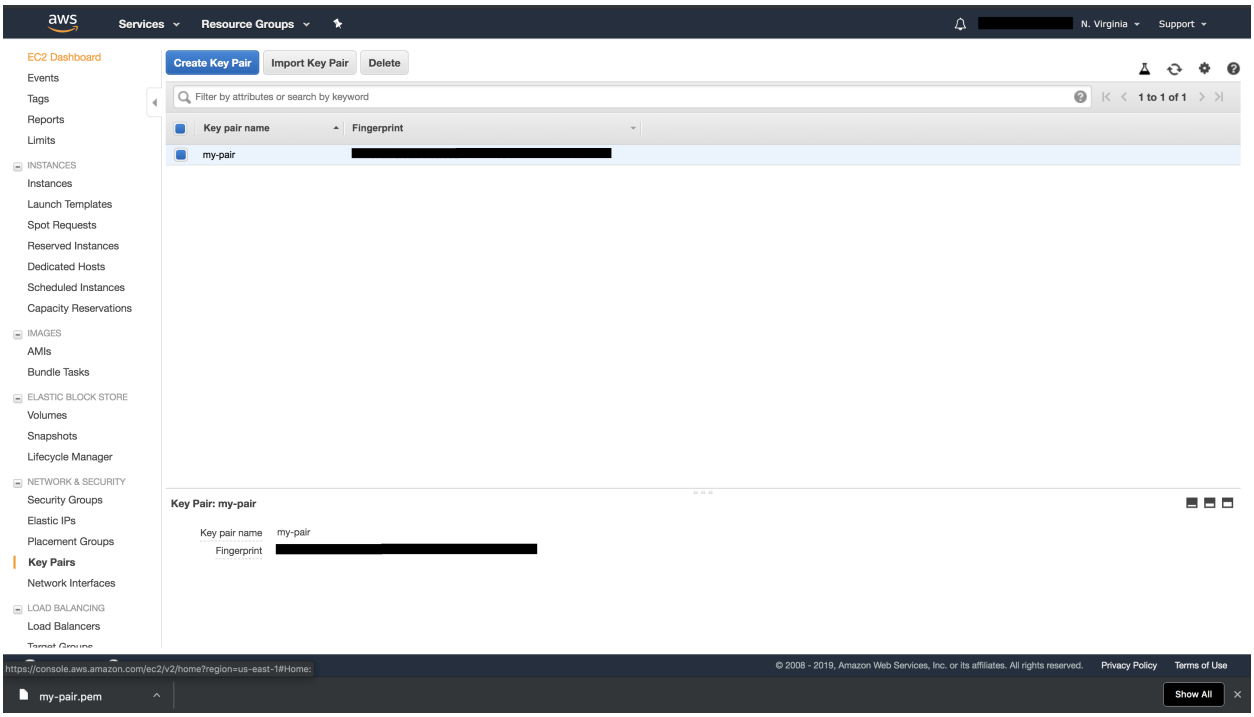
In order to create a Key Pair, go to the **Services -> EC2**:



On the left side list, go to **Key Pairs**:



Create a key pair and notice that a **.pem** file will be downloaded:



Important: Pick a recognazible name because you will use it later.

Important: Save your **.pem** file in a safe location as AWS will not give you access again to the file.

Warning: Set the right permissions to the **pem** file so only the root user can read it:

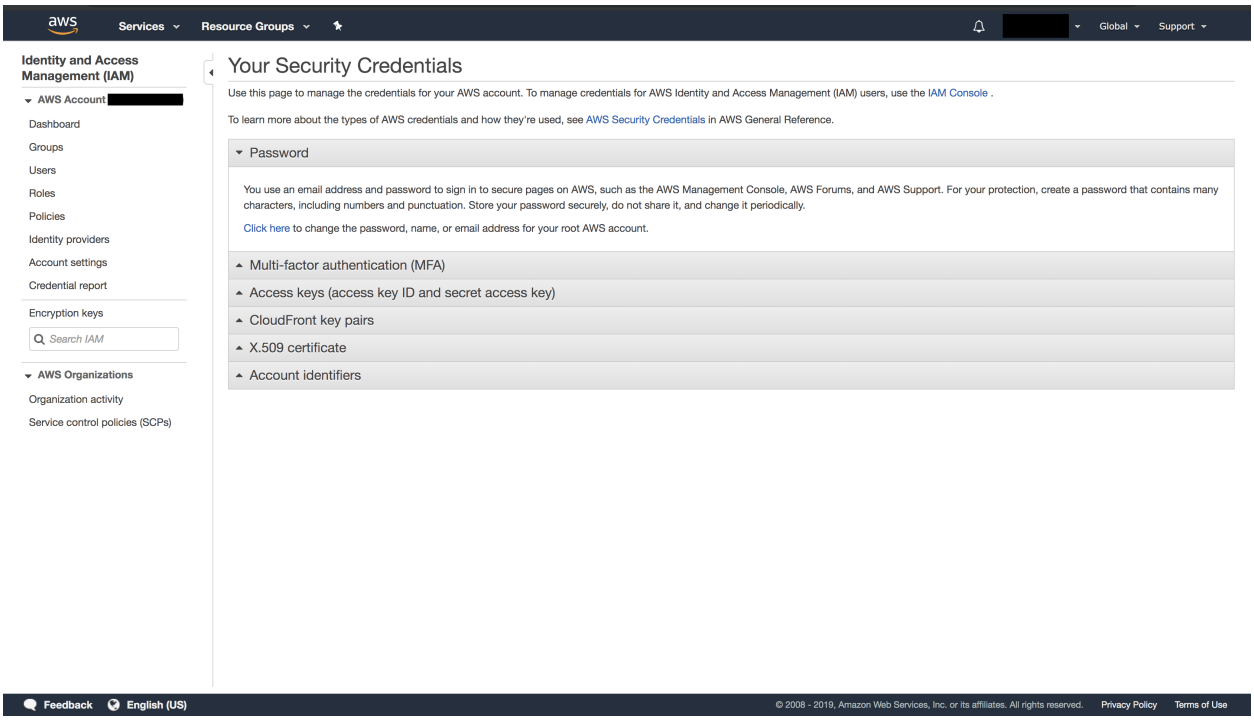
```
chmod 400 /path/to/my-pair.pem
```

17.1.2 Create security credentials

Important: If you already have security credentials, feel free to skip this section.

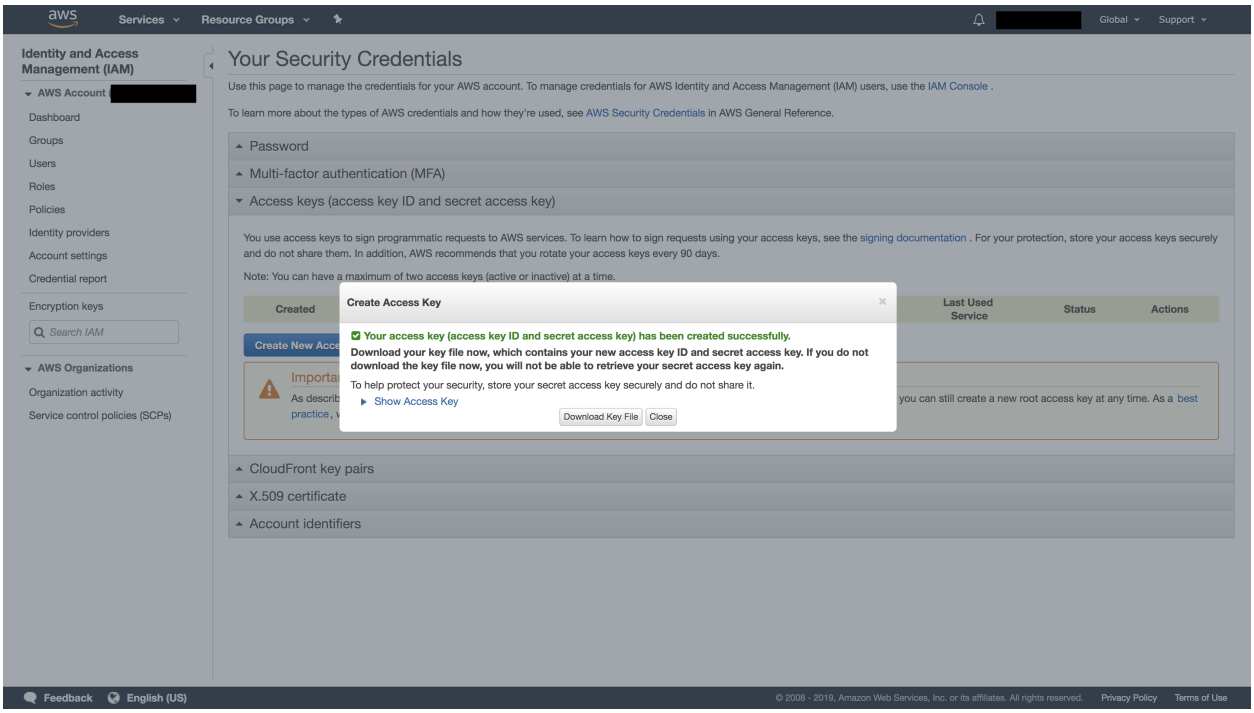
Security credentials are a way of authentication additionally to user/password information. For more information about this, go here

In order to create the Security Credentials, go to the right top section that contains your name. Press on **My Security Credentials**:



Go to **Access Keys** and click **Create New Access Key**.

When creating them, you should see something like:



Important: Download the file and make sure you save it in a safe location. **Note that you won't be able to access this information again from the console**.

Basic local configuration

Having access now to your `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, you will need to configure 2 configuration files:

Tip: This is an initial and basic configuration. More information here.

Important: At this point, you should have full access to AWS from your local computer through the Security Credentials. This snippet should run without raising errors:

```
import boto3
sess = boto3.Session()
sess.client("ec2").describe_instances()  # This may return no content if you have no instances
```

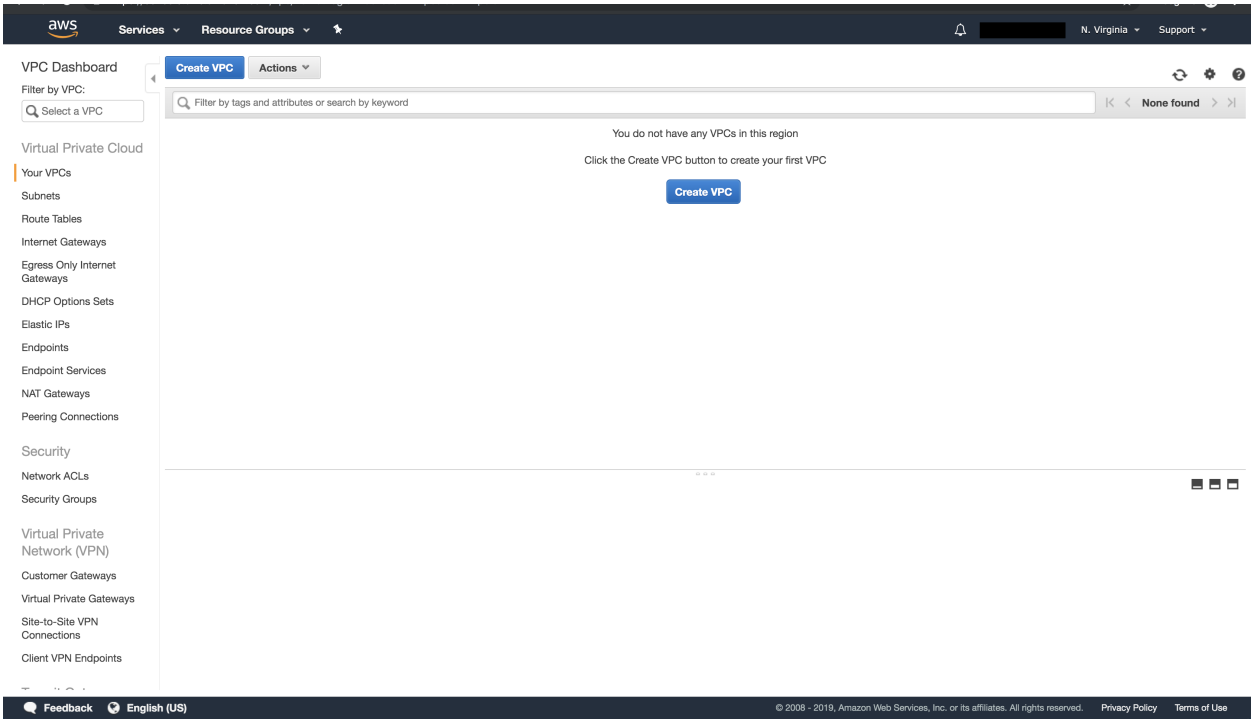
17.1.3 Create VPC and Subnet

You will need to create a VPC and a Subnet where your instances will be running.

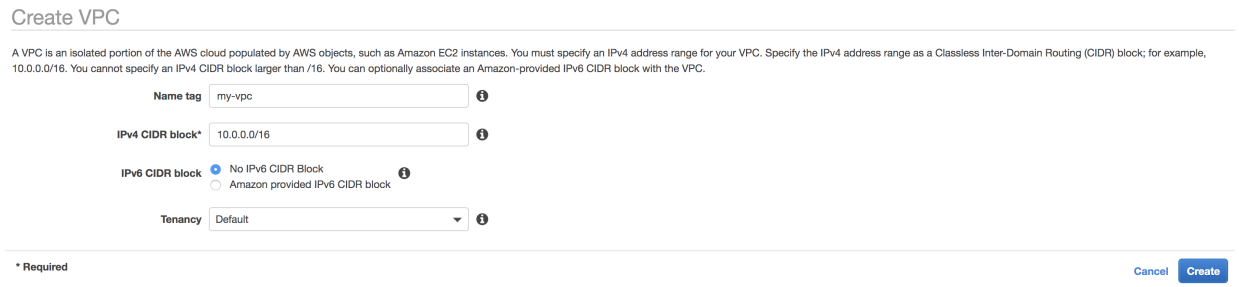
Tip: For more information about these topics, go here

1: Create VPC

In order to create a VPC, go to **Services -> VPC**. On the left side, go to **VPC**:

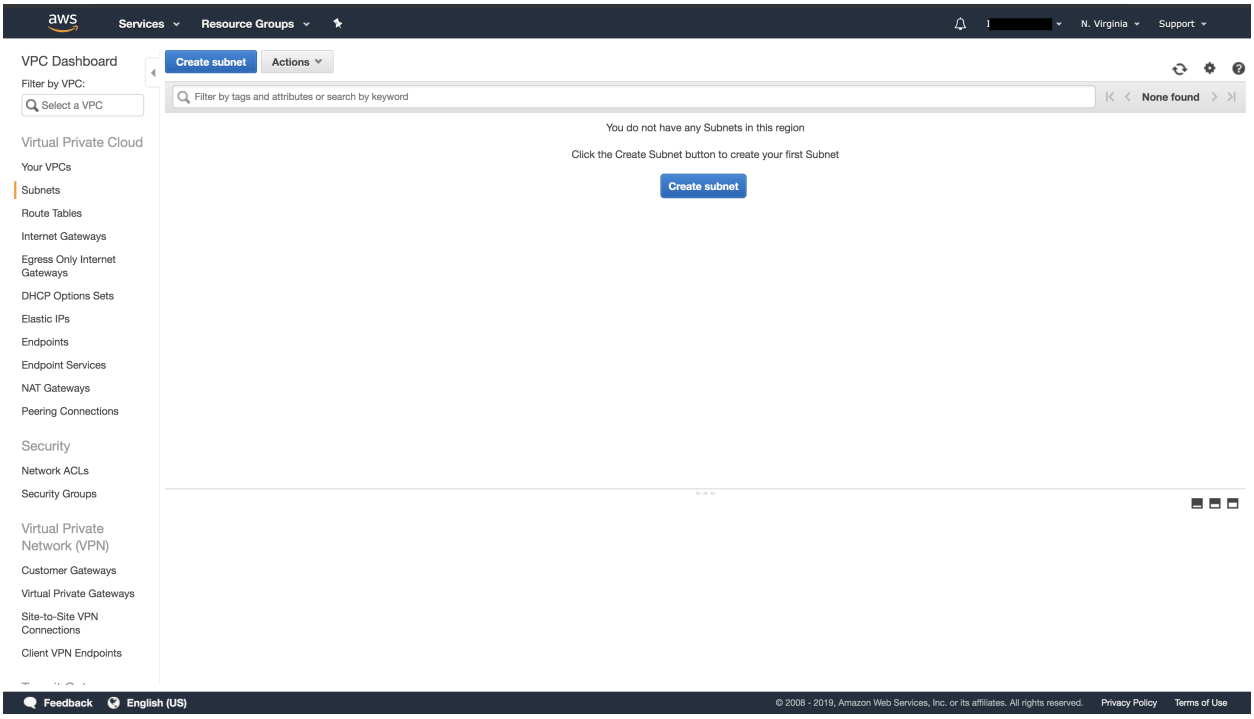


Click on **Create VPC** and choose some values. For example:

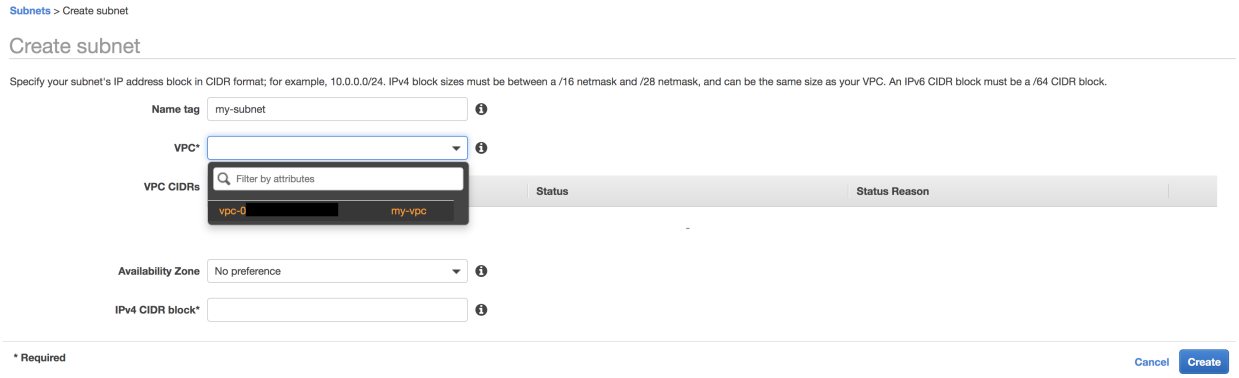


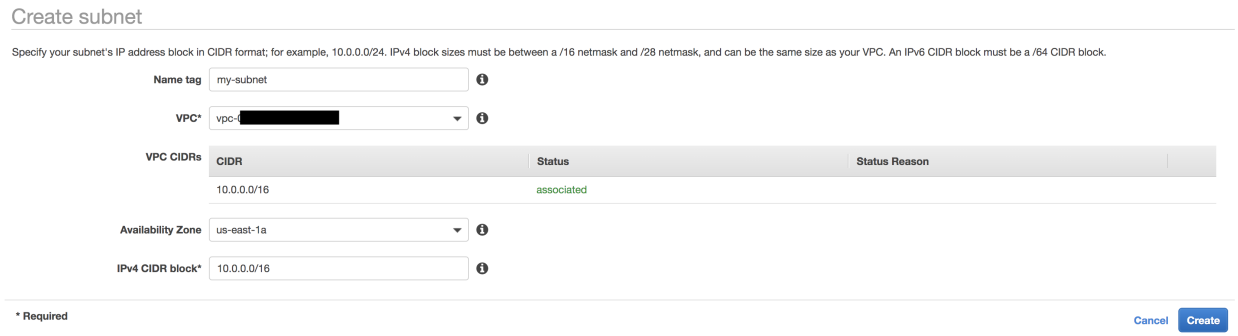
2: Create Subnet

In order to create a Subnet, go to **Services -> VPC**. On the left side, go to **Subnet**:



Click on **Create Subnet** and choose some values. Make sure to reference the **VPC** you just created:



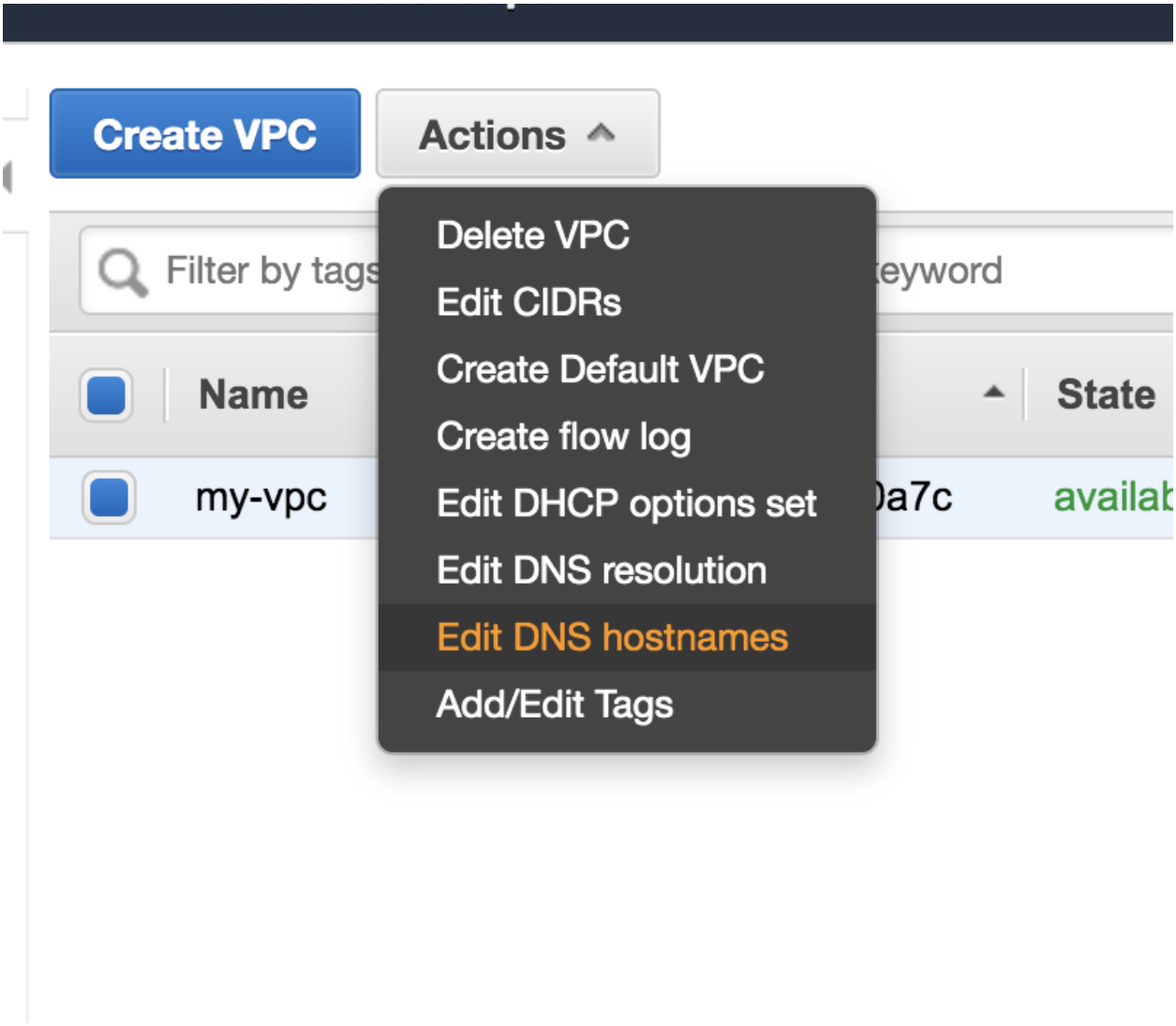


3: Enable auto-assign public IPs

This feature allows AWS to automatically assign public IPs to hosts that are created.

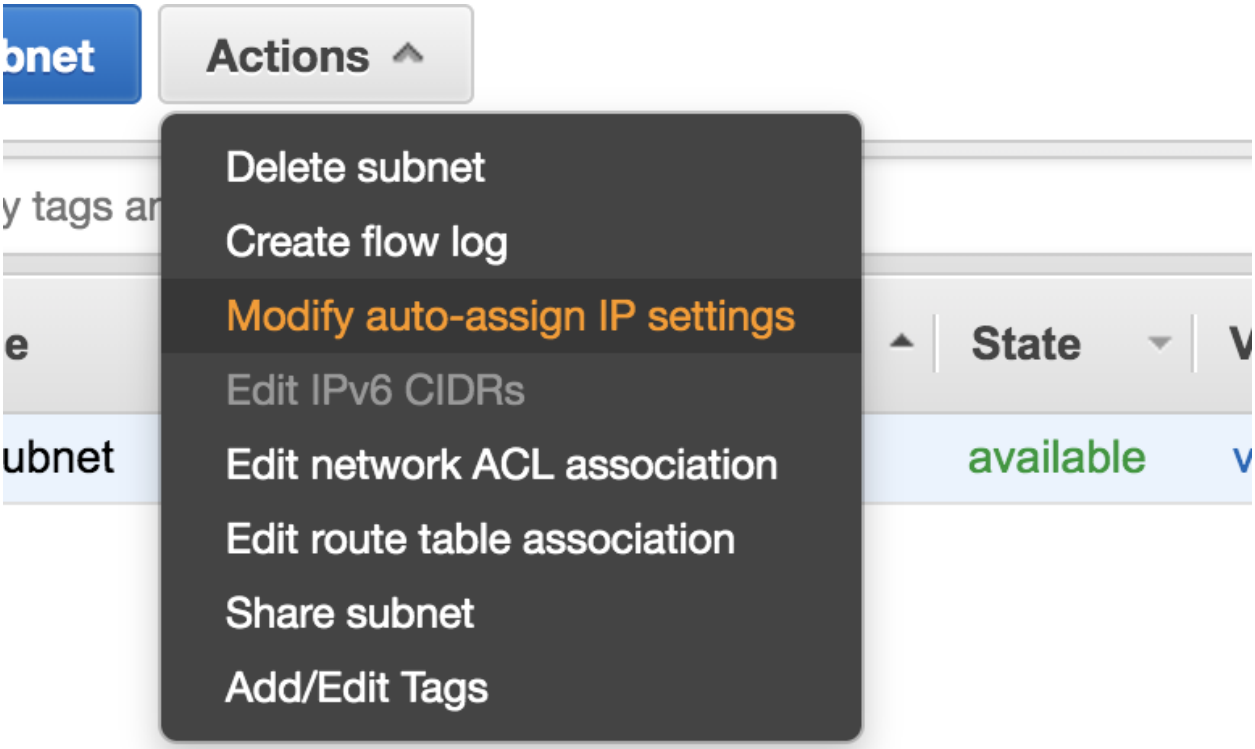
Important: This feature needs to be enabled for flambé.

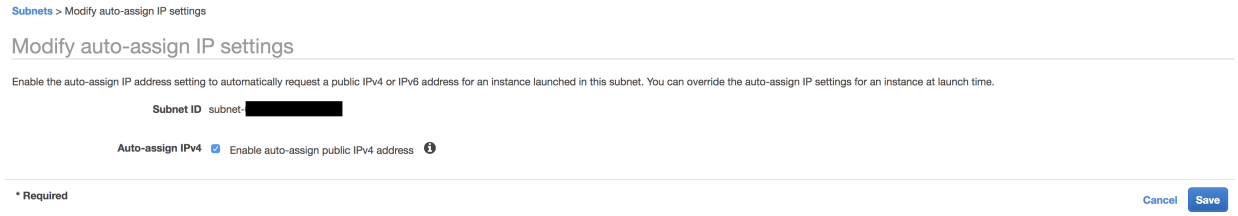
First, go into your **VPC** section and select the **VPC** you created in the first step. select **Actions -> Edit DNS Hostnames**:



Check on **enable** and click **Save**.

After that, go to your **Subnet** section and select the **Subnet** you created in step 2. select **Actions -> Modify auto-assign IP settings**:

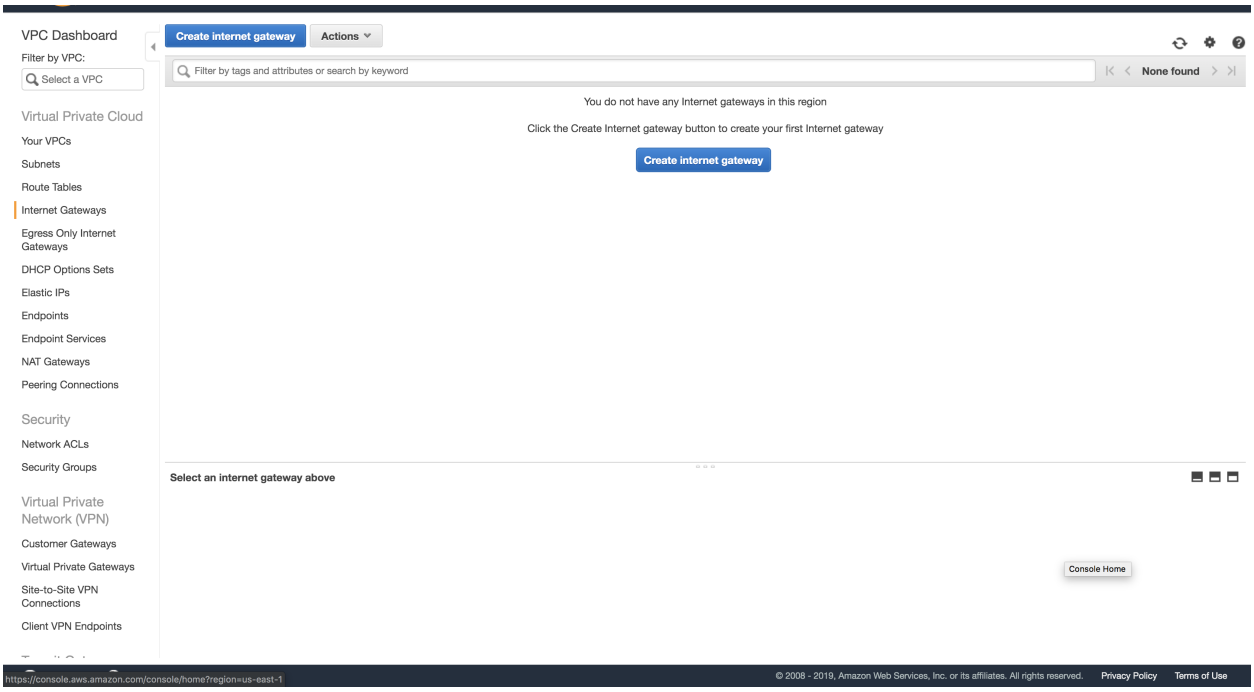




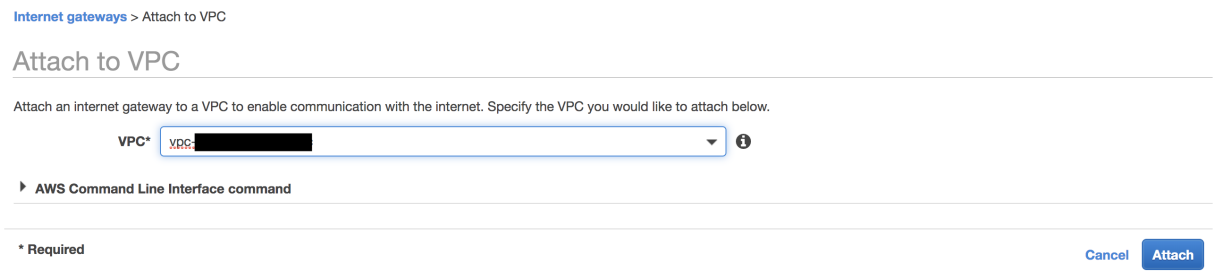
Enable the feature and click **Save**.

3: Configure Internet Gateways and Routes

Go to **Services -> VPC** and choose **Internet Gateways**. Verify that there is an internet gateway attached to your VPC. Otherwise, choose **Create Internet Gateway**:



After creating the internet gateway, go to **Actions -> Attach to VPC**. Follow the instructions to attach it to the created **VPC**:



Finally, go to **Subnet** section and select your **Subnet**. On the **Route Table** tab, verify that there is a route with `0.0.0.0/0` as the destination and the internet gateway for your **VPC** as the target.

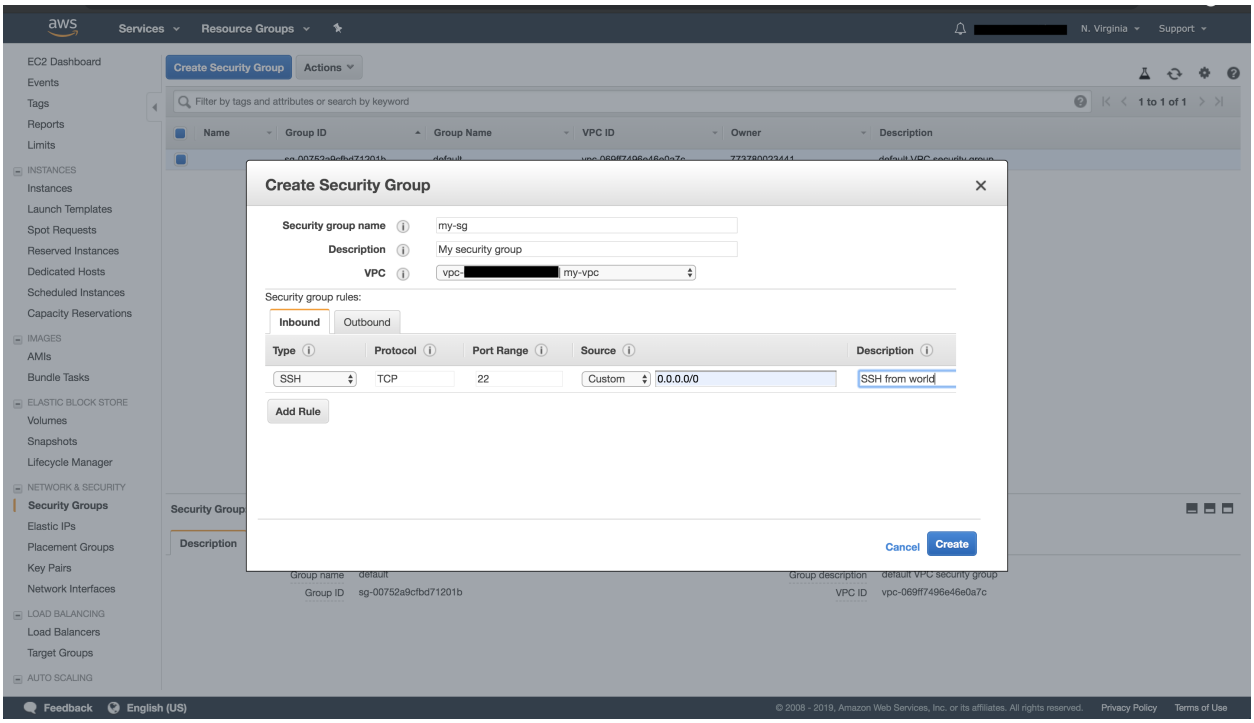
Otherwise, choose the ID of the route table (`rtb-xxxxxxxx`) to navigate to the **Route Table**. On the **Routes tab**, choose **Edit** routes. Choose **Add route**, use `0.0.0.0/0` as the destination and the internet gateway as the target. Choose **Save routes**.

17.1.4 Create Security Group (SG)

Security groups define security policies for the instaces. For more information go here

In order to create a SG, go to **Services -> EC2**. Click **Security Groups** on the left panel and then **Create Security Group** .

Important: The SG must have at least SSH access using standard port 22.



Tip: The above image shows the SG allows ssh traffic from `0.0.0.0/0` (which means from everywhere). If you are under static public IP or VPN, you can make more secure rules.

Important: If this cluster will be running remote *`Experiment`*, you may also want to open HTTP ports `49556` and `49558` for the Report Site and Tensorboard.

17.2 Creating a `AWSCluster`

At this point you should be ready to create your *`AWSCluster`*. You will need:

- The name of the key pair
- The location of the **pem** file and make sure that it has only reading permissions for root.
- The appropriate Security Group's ID
- The Subnet ID you wish all instances to live in.

Template:

Listing 1: aws-cluster.yaml

```
!AWSCluster

name: my-cluster
```

(continues on next page)

(continued from previous page)

```
factories_num: 2

# Type of machines.
factories_type: t3.small
orchestrator_type: t3.small

# Set timeouts for autmatic shutdown
orchestrator_timeout: -1
factories_timeout: -1

creator: user@company.com  # Pick whatever you want here

# Name of my key pair
key_name: my-pair

# Specify you pem location
key: /path/to/my-pair.pem

# You can add additional tags. This is OPTIONAL.
tags:
    project: my-project
    company: my-company

# Specify the Subnet ID
subnet_id: subnet-XXXXXXXXXXXXXXX

# The amount of GB for each instance.
volume_size: 100

# Specify the SG ID
security_group: sg-XXXXXXXXXXXXXXX
```

Create the cluster by executing:

```
flambe aws-cluster.yaml
```

You should see something like:

```
/$$$$$$$ /$$                          /$$           /$
| $$____/| $$                         | $$          /$
| $$     | $$  /$$$$$$  /$$$$$$/$$$$ | $$$$$$$   /$$$$$$
| $$$$$  | $$ |____  $$| $$_  $$_  $$| $$__  $$ /$$__  $$
| $$__/  | $$  /$$$$$$$| $$ \ $$ \ $$| $$  \ $$| $$$$$$$$
| $$     | $$ /$$__  $$| $$ | $$ | $$| $$  | $$| $$_____/
| $$     | $$|  $$$$$$$| $$ | $$ | $$| $$$$$$$/|  $$$$$$$
|__/     |__/ \_______/|__/ |__/ |__/|_______/  \_______/


VERSION: 0.4.0

16:59:48 | New orchestrator created 34.201.32.116
17:00:18 | 2 factories t3.small created (['75.101.205.248', '3.223.192.90']).
17:00:23 | Factories have no timeout
17:00:23 | Orchestrator 34.201.32.116 has no timeout
17:00:23 | Cluster loaded
17:00:36 | Distributed keys
17:00:37 | All instances prepared
17:01:07 | All instances prepared
17:01:07 | Flambe installed in all hosts
17:01:07 | ------------------- Done -------------------
```

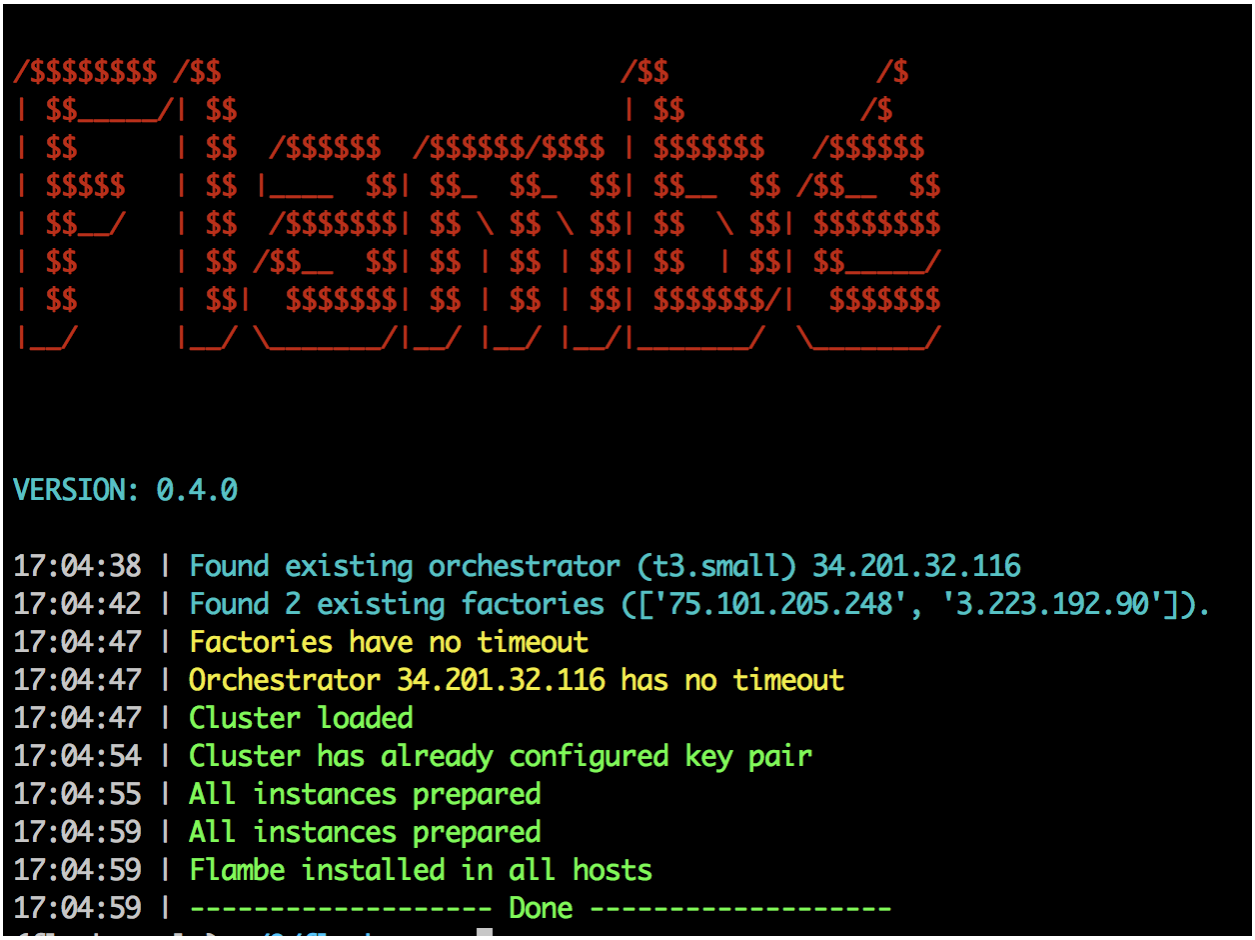
If everything is successfull, you should see you instances in your **EC2** console:

17.3 Reusing a `AWSCluster`

As long as the cluster name hasn't change, you can reuse the same cluster. So if after creating a cluster like the previous one you execute again:

```
flambe aws-cluster.yaml
```

Then flambé will automatically detect an existing cluster and it will reuse it:



Tip: This is particularly useful when running *Experiment* objects in the cluster. While you cannot run multiple experiments in the same cluster simultaneously, you can run them sequentially without having to set up the cluster again like the following:

```
flambe experiment.yaml -c my-cluster.yaml
# after experiment is done...
flambe other_experiment.yaml -c my-cluster.yaml
```

CHAPTER 18

flambe.dataset

18.1 Submodules

18.1.1 flambe.dataset.dataset

Module Contents

class flambe.dataset.dataset.**Dataset**
Bases: flambe.Component

Base Dataset interface.

Dataset objects offer the main interface to loading data into the experiment pipepine. Dataset objects have three attributes: *train*, *dev*, and *test*, each pointing to a list of examples.

Note that Datasets should also be "immutable", and as such, *__setitem__* and *__delitem__* will raise an error. Although this does not mean that the object will not be mutated in other ways, it should help avoid issues now and then.

train :Sequence[Sequence]
Returns the training data as a sequence of examples.

val :Sequence[Sequence]
Returns the validation data as a sequence of examples.

test :Sequence[Sequence]
Returns the test data as a sequence of examples.

__setitem__(*self*)
Raise an error.

__delitem__(*self*)
Raise an error.

18.1.2 flambe.dataset.tabular

Module Contents

class flambe.dataset.tabular.**DataView**(*data: np.ndarray, transform_hooks: List[Tuple[Field, Union[int, List[int]]]], cache: bool*)

TabularDataset view for the train, val or test split. This class must be used only internally in the TabularDataset class.

A DataView is a lazy Iterable that receives the operations from the TabularDataset object. When __getitem__ is called, then all the fields defined in the transform are applied.

This object can cache examples already transformed. To enable this, make sure to use this view under a Singleton pattern (there must only be one DataView per split in the TabularDataset).

raw
Returns an subscriptable version of the data

__getitem__(*self*, *index*)
Get an item from an index and apply the transformations dinamically.

is_empty(*self*)
Return if the DataView has data

cols(*self*)
Return the amount of columns the DataView has.

__len__(*self*)
Return the length of the dataview, ie the amount of examples it contains.

__setitem__(*self*)
Raise an error as DataViews are immutable.

__delitem__(*self*)
Raise an error as DataViews are immutable.

class flambe.dataset.tabular.**TabularDataset**(*train: Iterable[Iterable], val: Optional[Iterable[Iterable]] = None, test: Optional[Iterable[Iterable]] = None, cache: bool = True, named_columns: Optional[List[str]] = None, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.Dataset*

Loader for tabular data, usually in *csv* or *tsv* format.

A TabularDataset can represent any data that can be organized in a table. Internally, we store all information in a 2D numpy generic array. This object also behaves as a sequence over the whole dataset, chaining the training, validation and test data, in that order. This is useful in creating vocabularies or loading embeddings over the full datasets.

train
The list of training examples

Type np.ndarray

val
The list of validation examples

Type np.ndarray

test
The list of text examples

Type np.ndarray

train :np.ndarray
Returns the training data as a numpy nd array

val :np.ndarray
Returns the validation data as a numpy nd array

test :np.ndarray
Returns the test data as a numpy nd array

raw :np.ndarray
Returns all partitions of the data as a numpy nd array

cols :int
Returns the amount of columns in the tabular dataset

_set_transforms(*self, transform: Dict[str, Union[Field, Dict]]*)
Set transformations attributes and hooks to the data splits.

This method adds attributes for each field in the transform dict. It also adds hooks for the 'process' call in each field.

ATTENTION: This method works with the _train, _val and _test hidden attributes as this runs in the constructor and creates the hooks to be used in creating the properties.

classmethod from_path(*cls, train_path: str, val_path: Optional[str] = None, test_path: Optional[str] = None, sep: Optional[str] = 't', header: Optional[str] = 'infer', columns: Optional[Union[List[str], List[int]]] = None, encoding: Optional[str] = 'utf-8', transform: Dict[str, Union[Field, Dict]] = None*)
Load a TabularDataset from the given file paths.

Parameters

- **train_path** (*str*) – The path to the train data
- **val_path** (*str, optional*) – The path to the optional validation data
- **test_path** (*str, optional*) – The path to the optional test data
- **sep** (*str*) – Separator to pass to the *read_csv* method
- **header** (*Optional[Union[str, int]]*) – Use 0 for first line, None for no headers, and 'infer' to detect it automatically, defaults to 'infer'
- **columns** (*List[str]*) – List of columns to load, can be used to select a subset of columns, or change their order at loading time
- **encoding** (*str*) – The encoding format passed to the pandas reader
- **transform** (*Dict[str, Union[Field, Dict]]*) – The fields to be applied to the columns. Each field is identified with a name for easy linking.

classmethod autogen(*cls, data_path: str, test_path: Optional[str] = None, seed: Optional[int] = None, test_ratio: Optional[float] = 0.2, val_ratio: Optional[float] = 0.2, sep: Optional[str] = 't', header: Optional[str] = 'infer', columns: Optional[Union[List[str], List[int]]] = None, encoding: Optional[str] = 'utf-8', transform: Dict[str, Union[Field, Dict]] = None*)
Generate a test and validation set from the given file paths, then load a TabularDataset.

Parameters

- **data_path** (*str*) – The path to the data

- **test_path** (*Optional[str]*) – The path to the test data
- **seed** (*Optional[int]*) – Random seed to be used in test/val generation
- **test_ratio** (*Optional[float]*) – The ratio of the test dataset in relation to the whole dataset. If *test_path* is specified, this field has no effect.
- **val_ratio** (*Optional[float]*) – The ratio of the validation dataset in relation to the training dataset (whole - test)
- **sep** (*str*) – Separator to pass to the *read_csv* method
- **header** (*Optional[Union[str, int]]*) – Use 0 for first line, None for no headers, and 'infer' to detect it automatically, defaults to 'infer'
- **columns** (*List[str]*) – List of columns to load, can be used to select a subset of columns, or change their order at loading time
- **encoding** (*str*) – The encoding format passed to the pandas reader
- **transform** (*Dict[str, Union[Field, Dict]]*) – The fields to be applied to the columns. Each field is identified with a name for easy linking.

classmethod _load_file(*cls, path: str, sep: Optional[str] = 't', header: Optional[str] = 'infer', columns: Optional[Union[List[str], List[int]]] = None, encoding: Optional[str] = 'utf-8'*)

Load data from the given path.

The path may be either a single file or a directory. If it is a directory, each file is loaded according to the specified options and all the data is concatenated into a single list. The files will be processed in order based on file name.

Parameters

- **path** (*str*) – Path to data, could be a directory, a file, or a smart_open link
- **sep** (*str*) – Separator to pass to the *read_csv* method
- **header** (*Optional[Union[str, int]]*) – Use 0 for first line, None for no headers, and 'infer' to detect it automatically, defaults to 'infer'
- **columns** (*Optional[Union[List[str], List[int]]]*) – List of columns to load, can be used to select a subset of columns, or change their order at loading time
- **encoding** (*str*) – The encoding format passed to the pandas reader

Returns A tuple containing the list of examples (where each example is itself also a list or tuple of entries in the dataset) and an optional list of named columns (one string for each column in the dataset)

Return type Tuple[List[Tuple], Optional[List[str]]]

__len__(*self*)

Get the length of the dataset.

__iter__(*self*)

Iterate through the dataset.

__getitem__(*self*, *index*)

Get the item at the given index.

18.2 Package Contents

class flambe.dataset.**Dataset**
Bases: flambe.Component

Base Dataset interface.

Dataset objects offer the main interface to loading data into the experiment pipepine. Dataset objects have three attributes: *train*, *dev*, and *test*, each pointing to a list of examples.

Note that Datasets should also be "immutable", and as such, *__setitem__* and *__delitem__* will raise an error. Although this does not mean that the object will not be mutated in other ways, it should help avoid issues now and then.

train :Sequence[Sequence]
Returns the training data as a sequence of examples.

val :Sequence[Sequence]
Returns the validation data as a sequence of examples.

test :Sequence[Sequence]
Returns the test data as a sequence of examples.

__setitem__(*self*)
Raise an error.

__delitem__(*self*)
Raise an error.

class flambe.dataset.**TabularDataset**(*train: Iterable[Iterable], val: Optional[Iterable[Iterable]] = None, test: Optional[Iterable[Iterable]] = None, cache: bool = True, named_columns: Optional[List[str]] = None, transform: Dict[str, Union[Field, Dict]] = None*)
Bases: *flambe.dataset.Dataset*

Loader for tabular data, usually in *csv* or *tsv* format.

A TabularDataset can represent any data that can be organized in a table. Internally, we store all information in a 2D numpy generic array. This object also behaves as a sequence over the whole dataset, chaining the training, validation and test data, in that order. This is useful in creating vocabularies or loading embeddings over the full datasets.

train
The list of training examples

Type np.ndarray

val
The list of validation examples

Type np.ndarray

test
The list of text examples

Type np.ndarray

train :np.ndarray
Returns the training data as a numpy nd array

val :np.ndarray
Returns the validation data as a numpy nd array

test :np.ndarray
Returns the test data as a numpy nd array

raw :np.ndarray
Returns all partitions of the data as a numpy nd array

cols :int
Returns the amount of columns in the tabular dataset

_set_transforms(*self, transform: Dict[str, Union[Field, Dict]]*)
Set transformations attributes and hooks to the data splits.

This method adds attributes for each field in the transform dict. It also adds hooks for the 'process' call in each field.

ATTENTION: This method works with the _train, _val and _test hidden attributes as this runs in the constructor and creates the hooks to be used in creating the properties.

classmethod from_path(*cls, train_path: str, val_path: Optional[str] = None, test_path: Optional[str] = None, sep: Optional[str] = 't', header: Optional[str] = 'infer', columns: Optional[Union[List[str], List[int]]] = None, encoding: Optional[str] = 'utf-8', transform: Dict[str, Union[Field, Dict]] = None*)
Load a TabularDataset from the given file paths.

Parameters

- **train_path** (*str*) – The path to the train data
- **val_path** (*str, optional*) – The path to the optional validation data
- **test_path** (*str, optional*) – The path to the optional test data
- **sep** (*str*) – Separator to pass to the *read_csv* method
- **header** (*Optional[Union[str, int]]*) – Use 0 for first line, None for no headers, and 'infer' to detect it automatically, defaults to 'infer'
- **columns** (*List[str]*) – List of columns to load, can be used to select a subset of columns, or change their order at loading time
- **encoding** (*str*) – The encoding format passed to the pandas reader
- **transform** (*Dict[str, Union[Field, Dict]]*) – The fields to be applied to the columns. Each field is identified with a name for easy linking.

classmethod autogen(*cls, data_path: str, test_path: Optional[str] = None, seed: Optional[int] = None, test_ratio: Optional[float] = 0.2, val_ratio: Optional[float] = 0.2, sep: Optional[str] = 't', header: Optional[str] = 'infer', columns: Optional[Union[List[str], List[int]]] = None, encoding: Optional[str] = 'utf-8', transform: Dict[str, Union[Field, Dict]] = None*)
Generate a test and validation set from the given file paths, then load a TabularDataset.

Parameters

- **data_path** (*str*) – The path to the data
- **test_path** (*Optional[str]*) – The path to the test data
- **seed** (*Optional[int]*) – Random seed to be used in test/val generation
- **test_ratio** (*Optional[float]*) – The ratio of the test dataset in relation to the whole dataset. If *test_path* is specified, this field has no effect.

- **val_ratio** (*Optional[float]*) – The ratio of the validation dataset in relation to the training dataset (whole - test)
- **sep** (*str*) – Separator to pass to the *read_csv* method
- **header** (*Optional[Union[str, int]]*) – Use 0 for first line, None for no headers, and 'infer' to detect it automatically, defaults to 'infer'
- **columns** (*List[str]*) – List of columns to load, can be used to select a subset of columns, or change their order at loading time
- **encoding** (*str*) – The encoding format passed to the pandas reader
- **transform** (*Dict[str, Union[Field, Dict]]*) – The fields to be applied to the columns. Each field is identified with a name for easy linking.

classmethod _load_file(*cls, path: str, sep: Optional[str] = 't', header: Optional[str] = 'infer', columns: Optional[Union[List[str], List[int]]] = None, encoding: Optional[str] = 'utf-8'*)

Load data from the given path.

The path may be either a single file or a directory. If it is a directory, each file is loaded according to the specified options and all the data is concatenated into a single list. The files will be processed in order based on file name.

Parameters

- **path** (*str*) – Path to data, could be a directory, a file, or a smart_open link
- **sep** (*str*) – Separator to pass to the *read_csv* method
- **header** (*Optional[Union[str, int]]*) – Use 0 for first line, None for no headers, and 'infer' to detect it automatically, defaults to 'infer'
- **columns** (*Optional[Union[List[str], List[int]]]*) – List of columns to load, can be used to select a subset of columns, or change their order at loading time
- **encoding** (*str*) – The encoding format passed to the pandas reader

Returns A tuple containing the list of examples (where each example is itself also a list or tuple of entries in the dataset) and an optional list of named columns (one string for each column in the dataset)

Return type Tuple[List[Tuple], Optional[List[str]]]

__len__(*self*)

Get the length of the dataset.

__iter__(*self*)

Iterate through the dataset.

__getitem__(*self, index*)

Get the item at the given index.

CHAPTER 19

flambe.cluster

19.1 Subpackages

19.1.1 flambe.cluster.instance

Submodules

flambe.cluster.instance.errors

Module Contents

exception flambe.cluster.instance.errors.**RemoteCommandError**
Bases: Exception

Error raised when any remote command/script fail in an Instance.

exception flambe.cluster.instance.errors.**SSHConnectingError**
Bases: Exception

Error raised when opening a SSH connection fails.

exception flambe.cluster.instance.errors.**MissingAuthError**
Bases: Exception

Error raised when there is missing authentication information.

exception flambe.cluster.instance.errors.**RemoteFileTransferError**
Bases: Exception

Error raised when sending a local file to an Instance fails.

flambe.cluster.instance.instance

This modules includes base Instance classes to represent machines.

All Instance objects will be managed by Cluster objects (*flambe.cluster.cluster.Cluster*).

This base implementation is independant to the type of instance used.

Any new instance that flambe should support should inherit from the classes that are defined in this module.

Module Contents

flambe.cluster.instance.instance.**logger**

flambe.cluster.instance.instance.**InsT**

class flambe.cluster.instance.instance.**Instance**(*host: str*, *private_host: str*, *username: str*, *key: str*, *config: ConfigParser*, *debug: bool*, *use_public: bool = True*)

Bases: object

Encapsulates remote instances.

In this context, the instance is a running computer.

All instances used by flambe remote mode will inherit *Intance*. This class provides high-level methods to deal with remote instances (for example, sending a shell command over SSH).

Important: Instance objects should be pickeable. Make sure that all child classes can be pickled.

The flambe local process will communicate with the remote instances using SSH. The authentication mechanism will be using private keys.

Parameters

- **host** (*str*) – The public DNS host of the remote machine.
- **private_host** (*str*) – The private DNS host of the remote machine.
- **username** (*str*) – The machine's username.
- **key** (*str*) – The path to the ssh key used to communicate to the instance.
- **config** (*ConfigParser*) – The config object that contains useful information for the instance. For example, *config['SSH']['SSH_KEY']* should contain the path of the ssh key to login the remote instance.
- **debug** (*bool*) – True in case flambe was installed in dev mode, False otherwise.
- **use_public** (*bool*) – Wether this instance should use public or private IP. By default, the public IP is used. Private host is used when inside a private LAN.

fix_relpaths_in_config(*self*)

Updates all paths to be absolute. For example, if it contains "~/a/b/c" it will be change to /home/user/a/b/c (the appropiate $HOME value)

__enter__(*self*)

Method to use *Instance* instances with context managers

Returns The current instance

Return type *Instance*

__exit__(*self*, *exc_type: Optional[Type[BaseException]]*, *exc_value: Optional[BaseException]*, *traceback: Optional[TracebackType]*)

Exit method for the context manager.

This method will catch any uprising exception and raise it.

Returns If true, exceptions will not be raised.

Return type Optional[bool]

prepare (*self*)
Runs all neccessary processes to prepare the instances.

The child classes should implement this method according to the type of instance.

wait_until_accessible (*self*)
Waits until the instance is accesible through SSHClient

It attempts *const.RETRIES* time to ping SSH port to See if it's listening for incoming connections. In each attempt, it waits *const.RETRY_DELAY*.

Raises `ConnectionError` – If the instance is unaccesible through SSH

is_up (*self*)
Tests wether port 22 is open to incoming SSH connections

Returns True if instance is listening in port 22. False otherwise.

Return type bool

_get_cli (*self*)
Get an *SSHClient* in order to execute commands.

This will cache an existing SSHClient to optimize resource. This is a private method and should only be used in this module.

Returns The client for latter use.

Return type paramiko.SSHClient

Raises `SSHConnectingError` – In case opening an SSH connection fails.

_run_cmd (*self*, *cmd: str*, *retries: int = 1*, *wd: str = None*)
Runs a single shell command in the instance through SSH.

The command will be executed in one ssh connection. Don't expect calling several time to *_run_cmd* expecting to keep state between commands. To use mutliple commands, use: *_run_script*

Important: when running docker containers, don't use -it flag!

This is a private method and should only be used in this module.

Parameters

- **cmd** (*str*) – The command to execute.
- **retries** (*int*) – The amount of attempts to run the command if it fails. Default to 1.
- **wd** (*str*) – The working directory to 'cd' before running the command

Returns A *RemoteCommand* instance with success boolean and message.

Return type RemoteCommand

Examples

To get $HOME env

```
>>> instance._run_cmd("echo $HOME")
RemoteCommand(True, "/home/ubuntu")
```

This will not work

```
>>> instance._run_cmd("export var=10")
>>> instance._run_cmd("echo $var")
RemoteCommand(False, "")
```

This will work

```
>>> instance._run_cmd("export var=10; echo $var")
RemoteCommand(True, "10")
```

Raises `RemoteCommandError` – In case the *cmd* failes after *retries* attempts.

_run_script (*self*, *fname: str*, *desc: str*)

Runs a script by copyinh the script to the instance and executing it.

This is a private method and should only be used in this module.

Parameters

- **fname** (*str*) – The script filename
- **desc** (*str*) – A description for the script purpose. This will be used for the copied filename

Returns A *RemoteCommand* instance with success boolean and message.

Return type RemoteCommand

Raises `RemoteCommandError` – In case the script fails.

_remote_script (*self*, *host_fname: str*, *desc: str*)

Sends a local file containing a script to the instance using Paramiko SFTP.

It should be used as a context manager for latter execution of the script. See *_run_script* on how to use it.

After the context manager exists, then the file is removed from the instance.

This is a private method and should only be used in this module.

Parameters

- **host_fname** (*str*) – The local script filename
- **desc** (*str*) – A description for the script purpose. This will be used for the copied filename

Yields *str* – The remote filename of the copied local file.

Raises `RemoteCommandError` – In case sending the script fails.

run_cmds (*self, setup_cmds: List[str]*)

Execute a list of sequential commands

Parameters **setup_cmds** (*List[str]*) – The list of commands

Returns In case at least one command is not successful

Return type *RemoteCommandError*

send_rsync (*self*, *host_path: str*, *remote_path: str*, *params: List[str] = None*)

Send a local file or folder to a remote instance with rsync.

Parameters

- **host_path** (*str*) – The local filename or folder

- **remote_path** (*str*) – The remote filename or folder to use
- **params** (*List[str], optional*) – Extra parameters to be passed to rsync. For example, ["–filter=':- .gitignore'"]

Raises RemoteFileTransferError – In case sending the file fails.

get_home_path(*self*)
Return the $HOME value of the instance.

Returns The $HOME env value.

Return type str

Raises RemoteCommandError – If after 3 retries it is not able to get $HOME.

clean_containers(*self*)
Stop and remove all containers running

Raises RemoteCommandError – If command fails

clean_container_by_image(*self*, *image_name: str*)
Stop and remove all containers given an image name.

Parameters **image_name** (*str*) – The name of the image for which all containers should be stopped and removed.

Raises RemoteCommandError – If command fails

clean_container_by_command(*self*, *command: str*)
Stop and remove all containers with the given command.

Parameters **command** (*str*) – The command used to stop and remove the containers

Raises RemoteCommandError – If command fails

install_docker(*self*)
Install docker in a Ubuntu 18.04 distribution.

Raises RemoteCommandError – If it's not able to install docker. ie. then the installation script fails

install_extensions(*self, extensions: Dict[str, str]*)
Install local + pypi extensions.

Parameters **extension** (*Dict[str, str]*) – The extensions, as a dict from module_name to location

Raises errors.RemoteCommandError – If could not install an extension

install_flambe(*self*)
Pip install Flambe.

If dev mode is activated, then it rsyncs the local flambe folder and installs that version. If not, downloads from pypi.

Raises RemoteCommandError – If it's not able to install flambe.

is_docker_installed(*self*)
Check if docker is installed in the instance.

Executes command "docker –version" and expect it not to fail.

Returns True if docker is installed. False otherwise.

Return type bool

is_flambe_installed(*self*, *version: bool = True*)
Check if flambe is installed and if it matches version.

Parameters **version** (*bool*) – If True, also the version will be used. That is, if flag is True and the remote flambe version is different from the local flambe version, then this method will return False. If they match, then True. If version is False this method will return if there is ANY flambe version in the host.

Returns

Return type bool

is_docker_running(*self*)
Check if docker is running in the instance.

Executes the command "docker ps" and expects it not to fail.

Returns True if docker is running. False otherwise.

Return type bool

start_docker(*self*)
Restart docker.

Raises RemoteCommandError – If it's not able to restart docker.

is_node_running(*self*)
Return if the host is running a ray node

Returns

Return type bool

is_flambe_running(*self*)
Return if the host is running flambe

Returns

Return type bool

existing_dir(*self*, *_dir: str*)
Return if a directory exists in the host

Parameters **_dir** (*str*) – The name of the directory. It needs to be relative to $HOME

Returns True if exists. Otherwise, False.

Return type bool

shutdown_node(*self*)
Shut down the ray node in the host.

If the node is also the main node, then the entire cluster will shut down

shutdown_flambe(*self*)
Shut down flambe in the host

create_dirs(*self, relative_dirs: List[str]*)
Create the necessary folders in the host.

Parameters **relative_dirs** (*List[str]*) – The directories to create. They should be relative paths and $HOME of each host will be used to add the prefix.

remove_dir(*self*, *_dir: str*, *content_only: bool = True*)
Delete the specified dir result folder.

Parameters

- **_dir** (*str*) – The directory. It needs to be relative to the $HOME path as it will be prepended as a prefix.
- **content_only** (*bool*) – If True, the folder itseld will not be erased.

contains_gpu(*self*)
Return if this machine contains GPU.

This method will be used to possibly upgrade this factory to a GPUFactoryInstance.

class flambe.cluster.instance.instance.**CPUFactoryInstance**
Bases: *flambe.cluster.instance.instance.Instance*

This class represents a CPU Instance in the Ray cluster.

CPU Factories are instances that can run only one worker (no GPUs available). This class is mostly useful debugging.

Factory instances will not keep any important information. All information is going to be sent to an orchestrator machine.

prepare(*self*)
Prepare a CPU machine to be a worker node.

Checks if flambe is installed, and if not, installs it.

Raises RemoteCommandError – In case any step of the preparing process fails.

launch_node(*self*, *redis_address: str*)
Launch the ray worker node.

Parameters **redis_address** (*str*) – The URL of the main node. Must be IP:port

Raises RemoteCommandError – If not able to run node.

num_cpus(*self*)
Return the number of CPUs this host contains.

num_gpus(*self*)
Get the number of GPUs this host contains

Returns The number of GPUs

Return type int

Raises RemoteCommandError – If command to get the number of GPUs fails.

class flambe.cluster.instance.instance.**GPUFactoryInstance**
Bases: *flambe.cluster.instance.instance.CPUFactoryInstance*

This class represents an Nvidia GPU Factory Instance.

Factory instances will not keep any important information. All information is going to be sent to an Orchestrator machine.

prepare(*self*)
Prepare a GPU instance to run a ray worker node. For this, it installs CUDA and flambe if not installed.

Raises RemoteCommandError – In case any step of the preparing process fails.

install_cuda(*self*)
Install CUDA 10.0 drivers in an Ubuntu 18.04 distribution.

Raises RemoteCommandError – If it's not able to install drivers. ie if script fails

is_cuda_installed(*self*)
Check if CUDA is installed trying to execute *nvidia-smi*

Returns True if CUDA is installed. False otherwise.

Return type bool

class flambe.cluster.instance.instance.**OrchestratorInstance**

Bases: *flambe.cluster.instance.instance.Instance*

The orchestrator instance will be the main machine in a cluster.

It is going to be the main node in the ray cluster and it will also host other services. TODO: complete

All services besides ray will run in docker containers.

This instance does not needs to be a GPU machine.

prepare(*self*)

Install docker and flambe

Raises RemoteCommandError – In case any step of the preparing process fails.

launch_report_site(*self*, *progress_file: str*, *port: int*, *output_log: str*, *output_dir: str*, *tensorboard_port: int*)

Launch the report site.

The report site is a Flask web app.

Raises RemoteCommandError – In case the launch process fails

is_tensorboard_running(*self*)

Return wether tensorboard is running in the host as docker.

Returns True if Tensorboard is running, False otherwise.

Return type bool

is_report_site_running(*self*)

Return wether the report site is running in the host

Returns

Return type bool

remove_tensorboard(*self*)

Removes tensorboard from the orchestrator.

remove_report_site(*self*)

Remove report site from the orchestrator.

launch_tensorboard(*self*, *logs_dir: str*, *tensorboard_port: int*)

Launch tensorboard.

Parameters

- **logs_dir** (*str*) – Tensorboard logs directory
- **tensorboard_port** (*int*) – The port where tensorboard will be available

Raises RemoteCommandError – In case the launch process fails

existing_tmux_session(*self*, *session_name: str*)

Return if there is an existing tmux session with the same name

Parameters **session_name** (*str*) – The exact name of the searched tmux session

Returns

Return type bool

kill_tmux_session (*self*, *session_name: str*)
Kill an existing tmux session

Parameters **session_name** (*str*) – The exact name of the tmux session to be removed

launch_flambe (*self*, *config_file: str*, *secrets_file: str*, *force: bool*)
Launch flambe execution in the remote host

Parameters

- **config_file** (*str*) – The config filename relative to the orchestrator
- **secrets_file** (*str*) – The filepath containing the secrets for the orchestrator
- **force** (*bool*) – The force parameters that was originally passed to flambe

launch_node (*self*, *port: int*)
Launch the main ray node in given sftp server in port 49559.

Parameters **port** (*int*) – Available port to launch the redis DB of the main ray node

Raises RemoteCommandError – In case the launch process fails

worker_nodes (*self*)
Returns the list of worker nodes

Returns The list of worker nodes identified by their hostname

Return type List[str]

rsync_folder (*self*, *_from*, *_to*, *exclude=None*)
Rsyncs folders or files.

One of the folders NEEDS to be local. The remaining one can be remote if needed.

Package Contents

class flambe.cluster.instance.**Instance**(*host: str*, *private_host: str*, *username: str*, *key: str*, *config: ConfigParser*, *debug: bool*, *use_public: bool = True*)

Bases: object

Encapsulates remote instances.

In this context, the instance is a running computer.

All instances used by flambe remote mode will inherit *Intance*. This class provides high-level methods to deal with remote instances (for example, sending a shell command over SSH).

Important: Instance objects should be pickeable. Make sure that all child classes can be pickled.

The flambe local process will communicate with the remote instances using SSH. The authentication mechanism will be using private keys.

Parameters

- **host** (*str*) – The public DNS host of the remote machine.
- **private_host** (*str*) – The private DNS host of the remote machine.
- **username** (*str*) – The machine's username.
- **key** (*str*) – The path to the ssh key used to communicate to the instance.

- **config** (*ConfigParser*) – The config object that contains useful information for the instance. For example, *config['SSH']['SSH_KEY']* should contain the path of the ssh key to login the remote instance.
- **debug** (*bool*) – True in case flambe was installed in dev mode, False otherwise.
- **use_public** (*bool*) – Wether this instance should use public or private IP. By default, the public IP is used. Private host is used when inside a private LAN.

fix_relpaths_in_config(*self*)
Updates all paths to be absolute. For example, if it contains "~/a/b/c" it will be change to /home/user/a/b/c (the appropiate $HOME value)

__enter__(*self*)
Method to use *Instance* instances with context managers

Returns The current instance

Return type *Instance*

__exit__(*self, exc_type: Optional[Type[BaseException]], exc_value: Optional[BaseException], traceback: Optional[TracebackType]*)
Exit method for the context manager.

This method will catch any uprising exception and raise it.

Returns If true, exceptions will not be raised.

Return type Optional[bool]

prepare(*self*)
Runs all neccessary processes to prepare the instances.

The child classes should implement this method according to the type of instance.

wait_until_accessible(*self*)
Waits until the instance is accesible through SSHClient

It attempts *const.RETRIES* time to ping SSH port to See if it's listening for incoming connections. In each attempt, it waits *const.RETRY_DELAY*.

Raises ConnectionError – If the instance is unaccesible through SSH

is_up(*self*)
Tests wether port 22 is open to incoming SSH connections

Returns True if instance is listening in port 22. False otherwise.

Return type bool

_get_cli(*self*)
Get an *SSHClient* in order to execute commands.

This will cache an existing SSHClient to optimize resource. This is a private method and should only be used in this module.

Returns The client for latter use.

Return type paramiko.SSHClient

Raises SSHConnectingError – In case opening an SSH connection fails.

_run_cmd(*self, cmd: str, retries: int = 1, wd: str = None*)
Runs a single shell command in the instance through SSH.

The command will be executed in one ssh connection. Don't expect calling several time to *_run_cmd* expecting to keep state between commands. To use mutliple commands, use: *_run_script*

Important: when running docker containers, don't use -it flag!

This is a private method and should only be used in this module.

Parameters

- **cmd** (*str*) – The command to execute.
- **retries** (*int*) – The amount of attempts to run the command if it fails. Default to 1.
- **wd** (*str*) – The working directory to 'cd' before running the command

Returns A *RemoteCommand* instance with success boolean and message.

Return type RemoteCommand

Examples

To get $HOME env

```
>>> instance._run_cmd("echo $HOME")
RemoteCommand(True, "/home/ubuntu")
```

This will not work

```
>>> instance._run_cmd("export var=10")
>>> instance._run_cmd("echo $var")
RemoteCommand(False, "")
```

This will work

```
>>> instance._run_cmd("export var=10; echo $var")
RemoteCommand(True, "10")
```

Raises `RemoteCommandError` – In case the *cmd* failes after *retries* attempts.

_run_script (*self*, *fname: str*, *desc: str*)

Runs a script by copyinh the script to the instance and executing it.

This is a private method and should only be used in this module.

Parameters

- **fname** (*str*) – The script filename
- **desc** (*str*) – A description for the script purpose. This will be used for the copied filename

Returns A *RemoteCommand* instance with success boolean and message.

Return type RemoteCommand

Raises `RemoteCommandError` – In case the script fails.

_remote_script (*self*, *host_fname: str*, *desc: str*)

Sends a local file containing a script to the instance using Paramiko SFTP.

It should be used as a context manager for latter execution of the script. See *_run_script* on how to use it.

After the context manager exists, then the file is removed from the instance.

This is a private method and should only be used in this module.

Parameters

- **host_fname** (*str*) – The local script filename
- **desc** (*str*) – A description for the script purpose. This will be used for the copied filename

Yields *str* – The remote filename of the copied local file.

Raises RemoteCommandError – In case sending the script fails.

run_cmds(*self, setup_cmds: List[str]*)
Execute a list of sequential commands

Parameters **setup_cmds** (*List[str]*) – The list of commands

Returns In case at least one command is not successful

Return type *RemoteCommandError*

send_rsync(*self*, *host_path: str*, *remote_path: str*, *params: List[str] = None*)
Send a local file or folder to a remote instance with rsync.

Parameters

- **host_path** (*str*) – The local filename or folder
- **remote_path** (*str*) – The remote filename or folder to use
- **params** (*List[str], optional*) – Extra parameters to be passed to rsync. For example, ["–filter=':- .gitignore'"]

Raises RemoteFileTransferError – In case sending the file fails.

get_home_path(*self*)
Return the $HOME value of the instance.

Returns The $HOME env value.

Return type str

Raises RemoteCommandError – If after 3 retries it is not able to get $HOME.

clean_containers(*self*)
Stop and remove all containers running

Raises RemoteCommandError – If command fails

clean_container_by_image(*self*, *image_name: str*)
Stop and remove all containers given an image name.

Parameters **image_name** (*str*) – The name of the image for which all containers should be stopped and removed.

Raises RemoteCommandError – If command fails

clean_container_by_command(*self*, *command: str*)
Stop and remove all containers with the given command.

Parameters **command** (*str*) – The command used to stop and remove the containers

Raises RemoteCommandError – If command fails

install_docker(*self*)
Install docker in a Ubuntu 18.04 distribution.

Raises RemoteCommandError – If it's not able to install docker. ie. then the installation script fails

install_extensions(*self, extensions: Dict[str, str]*)
Install local + pypi extensions.

Parameters **extension** (*Dict[str, str]*) – The extensions, as a dict from module_name to location

Raises *errors.RemoteCommandError* – If could not install an extension

install_flambe(*self*)
Pip install Flambe.

If dev mode is activated, then it rsyncs the local flambe folder and installs that version. If not, downloads from pypi.

Raises RemoteCommandError – If it's not able to install flambe.

is_docker_installed(*self*)
Check if docker is installed in the instance.

Executes command "docker –version" and expect it not to fail.

Returns True if docker is installed. False otherwise.

Return type bool

is_flambe_installed(*self*, *version: bool = True*)
Check if flambe is installed and if it matches version.

Parameters **version** (*bool*) – If True, also the version will be used. That is, if flag is True and the remote flambe version is different from the local flambe version, then this method will return False. If they match, then True. If version is False this method will return if there is ANY flambe version in the host.

Returns

Return type bool

is_docker_running(*self*)
Check if docker is running in the instance.

Executes the command "docker ps" and expects it not to fail.

Returns True if docker is running. False otherwise.

Return type bool

start_docker(*self*)
Restart docker.

Raises RemoteCommandError – If it's not able to restart docker.

is_node_running(*self*)
Return if the host is running a ray node

Returns

Return type bool

is_flambe_running(*self*)
Return if the host is running flambe

Returns

Return type bool

existing_dir(*self*, *_dir: str*)
Return if a directory exists in the host

Parameters **_dir** (*str*) – The name of the directory. It needs to be relative to $HOME

Returns True if exists. Otherwise, False.

Return type bool

shutdown_node (*self*)
Shut down the ray node in the host.

If the node is also the main node, then the entire cluster will shut down

shutdown_flambe (*self*)
Shut down flambe in the host

create_dirs (*self, relative_dirs: List[str]*)
Create the necessary folders in the host.

Parameters **relative_dirs** (*List[str]*) – The directories to create. They should be relative paths and $HOME of each host will be used to add the prefix.

remove_dir (*self*, *_dir: str*, *content_only: bool = True*)
Delete the specified dir result folder.

Parameters

- **_dir** (*str*) – The directory. It needs to be relative to the $HOME path as it will be prepended as a prefix.
- **content_only** (*bool*) – If True, the folder itseld will not be erased.

contains_gpu (*self*)
Return if this machine contains GPU.

This method will be used to possibly upgrade this factory to a GPUFactoryInstance.

class flambe.cluster.instance.**CPUFactoryInstance**
Bases: *flambe.cluster.instance.instance.Instance*

This class represents a CPU Instance in the Ray cluster.

CPU Factories are instances that can run only one worker (no GPUs available). This class is mostly useful debugging.

Factory instances will not keep any important information. All information is going to be sent to an orchestrator machine.

prepare (*self*)
Prepare a CPU machine to be a worker node.

Checks if flambe is installed, and if not, installs it.

Raises RemoteCommandError – In case any step of the preparing process fails.

launch_node (*self*, *redis_address: str*)
Launch the ray worker node.

Parameters **redis_address** (*str*) – The URL of the main node. Must be IP:port

Raises RemoteCommandError – If not able to run node.

num_cpus (*self*)
Return the number of CPUs this host contains.

num_gpus (*self*)
Get the number of GPUs this host contains

Returns The number of GPUs

Return type int

Raises RemoteCommandError – If command to get the number of GPUs fails.

class flambe.cluster.instance.**GPUFactoryInstance**
Bases: *flambe.cluster.instance.instance.CPUFactoryInstance*

This class represents an Nvidia GPU Factory Instance.

Factory instances will not keep any important information. All information is going to be sent to an Orchestrator machine.

prepare(*self*)
Prepare a GPU instance to run a ray worker node. For this, it installs CUDA and flambe if not installed.

Raises RemoteCommandError – In case any step of the preparing process fails.

install_cuda(*self*)
Install CUDA 10.0 drivers in an Ubuntu 18.04 distribution.

Raises RemoteCommandError – If it's not able to install drivers. ie if script fails

is_cuda_installed(*self*)
Check if CUDA is installed trying to execute *nvidia-smi*

Returns True if CUDA is installed. False otherwise.

Return type bool

class flambe.cluster.instance.**OrchestratorInstance**
Bases: *flambe.cluster.instance.instance.Instance*

The orchestrator instance will be the main machine in a cluster.

It is going to be the main node in the ray cluster and it will also host other services. TODO: complete

All services besides ray will run in docker containers.

This instance does not needs to be a GPU machine.

prepare(*self*)
Install docker and flambe

Raises RemoteCommandError – In case any step of the preparing process fails.

launch_report_site(*self*, *progress_file: str*, *port: int*, *output_log: str*, *output_dir: str*, *tensorboard_port: int*)
Launch the report site.

The report site is a Flask web app.

Raises RemoteCommandError – In case the launch process fails

is_tensorboard_running(*self*)
Return wether tensorboard is running in the host as docker.

Returns True if Tensorboard is running, False otherwise.

Return type bool

is_report_site_running(*self*)
Return wether the report site is running in the host

Returns

Return type bool

remove_tensorboard(*self*)
Removes tensorboard from the orchestrator.

remove_report_site(*self*)
Remove report site from the orchestrator.

launch_tensorboard(*self*, *logs_dir: str*, *tensorboard_port: int*)
Launch tensorboard.

Parameters

- **logs_dir** (*str*) – Tensorboard logs directory
- **tensorboard_port** (*int*) – The port where tensorboard will be available

Raises RemoteCommandError – In case the launch process fails

existing_tmux_session(*self*, *session_name: str*)
Return if there is an existing tmux session with the same name

Parameters **session_name** (*str*) – The exact name of the searched tmux session

Returns

Return type bool

kill_tmux_session(*self*, *session_name: str*)
Kill an existing tmux session

Parameters **session_name** (*str*) – The exact name of the tmux session to be removed

launch_flambe(*self*, *config_file: str*, *secrets_file: str*, *force: bool*)
Launch flambe execution in the remote host

Parameters

- **config_file** (*str*) – The config filename relative to the orchestrator
- **secrets_file** (*str*) – The filepath containing the secrets for the orchestrator
- **force** (*bool*) – The force parameters that was originally passed to flambe

launch_node(*self*, *port: int*)
Launch the main ray node in given sftp server in port 49559.

Parameters **port** (*int*) – Available port to launch the redis DB of the main ray node

Raises RemoteCommandError – In case the launch process fails

worker_nodes(*self*)
Returns the list of worker nodes

Returns The list of worker nodes identified by their hostname

Return type List[str]

rsync_folder(*self*, *_from*, *_to*, *exclude=None*)
Rsyncs folders or files.

One of the folders NEEDS to be local. The remaining one can be remote if needed.

19.2 Submodules

19.2.1 flambe.cluster.aws

Implementation of a Cluster with AWS EC2 as the cloud provider

Module Contents

flambe.cluster.aws.**logger**

flambe.cluster.aws.**T**

class flambe.cluster.aws.**AWSCluster**(*name: str*, *factories_num: int*, *factories_type: str*, *orchestrator_type: str*, *key_name: str*, *security_group: str*, *subnet_id: str*, *creator: str*, *key: str*, *volume_type: str = 'gp2'*, *region_name: Optional[str] = None*, *username: str = 'ubuntu'*, *tags: Dict[str, str] = None*, *orchestrator_ami: str = None*, *factory_ami: str = None*, *dedicated: bool = False*, *orchestrator_timeout: int = -1*, *factories_timeout: int = 1*, *volume_size: int = 100*, *setup_cmds: Optional[List[str]] = None*)

Bases: *flambe.cluster.cluster.Cluster*

This Cluster implementation uses AWS EC2 as the cloud provider.

This cluster works with AWS Instances that are defined in: *flambe.remote.instance.aws*

Parameters

- **name** (*str*) – The unique name for the cluster
- **factories_num** (*int*) – The amount of factories to use. This is not the amount of workers, as each factories can contain multiple GPUs and therefore, multiple workers.
- **factories_type** (*str*) – The type of instance to use for the Factory Instances. GPU instances are required for AWS the AWSCluster. “p2” and “p3” instances are recommended.
- **factory_ami** (*str*) – The AMI to be used for the Factory instances. Custom Flambe AMI are provided based on Ubuntu 18.04 distribution.
- **orchestrator_type** (*str*) – The type of instance to use for the Orchestrator Instances. This may not be a GPU instances. At least a “t2.small” instance is recommended.
- **key_name** (*str*) – The key name that will be used to connect into the instance.
- **creator** (*str*) – The creator should be a user identifier for the instances. This information will create a tag called ‘creator’ and it will also be used to retrieve existing hosts owned by the user.
- **key** (*str*) – The path to the ssh key used to communicate to all instances. IMPORTANT: all instances must be accessible with the same key.
- **volume_type** (*str*) – The type of volume in AWS to use. Only ‘gp2’ and ‘io1’ are currently available. If ‘io1’ is used, then IOPS will be fixed to 5000. IMPORTANT: ‘io1’ volumes are significantly more expensive than ‘gp2’ volumes. Defaults to ‘gp2’.
- **region_name** (*Optional[str]*) – The region name to use. If not specified, it uses the locally configured region name or ‘us-east-1’ in case it’s not configured.

- **username** (*str*) – The username of the instances the cluster will handle. Defaults to 'ubuntu'. IMPORTANT: for now all instances need to have the same username.
- **tags** (*Dict[str, str]*) – A dictionary with tags that will be added to all created hosts.
- **security_group** (*str*) – The security group to use to create the instances.
- **subnet_id** (*str*) – The subnet ID to use.
- **orchestrator_ami** (*str*) – The AMI to be used for the Factory instances. Custom Flambe AMI are provided based on Ubuntu 18.04 distribution.
- **dedicated** (*bool*) – Wether all created instances are dedicated instances or shared.
- **orchestrator_timeout** (*int*) – Number of consecutive hours before terminating the orchestrator once the experiment is over (either success of failure). Specify -1 to disable automatic shutdown (the orchestrator will stay on until manually terminated) and 0 to shutdown when the experiment is over. For example, if specifying 24, then the orchestrator will be shut down one day after the experiment is over. ATTENTION: This also applies when the experiment ends with an error. Default is -1.
- **factories_timeout** (*int*) – Number of consecutive hours to automatically terminate factories once the experiment is over (either success or failure). Specify -1 to disable automatic shutdown (the factories will stay on until manually terminated) and 0 to shutdown when the experiment is over. For example, if specifying 10, then the factories will be shut down 10 hours after the experiment is over. ATTENTION: This also applies when the experiment ends with an error. Default is 1.
- **volume_size** (*int*) – The disk size in GB that all hosts will contain. Defaults to 100 GB.
- **setup_cmds** (*Optional[List[str]]*) – A list of commands to be run on all hosts for setup purposes. These commands can be used to mount volumes, install software, etc. Defaults to None. IMPORTANT: the commands need to be idempotent and they shouldn't expect user input.

_get_boto_session(*self, region_name: Optional[str]*)

Get the boto3 Session from which the resources and clients will be created.

This method is called by the contructor.

Parameters **region_name** (*Optional[str]*) – The region to use. If None, boto3 will resolve to the locally configured region_name or 'us-east-1' if not configured.

Returns The boto3 Session to use

Return type boto3.Session

load_all_instances(*self*)

Launch all instances for the experiment.

This method launches both the orchestrator and the factories.

_existing_cluster(*self*)

Whether there is an existing cluster that matches name.

The cluster should also match all other tags, including Creator)

Returns Returns the (boto_orchestrator, [boto_factories]) that match the experiment's name.

Return type Tuple[Any, List[Any]]

_get_existing_tags(*self*, *boto_instance: boto3.resources.factory.ec2.Instance*)

Gets the tags of a EC2 instances

Parameters **boto_instance** (*BotoIns*) – The EC2 instance to access the tags.

Returns Key, Value for the specified tags.

Return type Dict[str, str]

flambe_own_running_instances (*self*)
Get running instances with matching tags.

Yields *Tuple['boto3.resources.factory.ec2.Instance', str]* – A tuple with the instance and the name of the EC2 instance.

name_hosts (*self*)
Name the orchestrator and factories.

_get_all_tags (*self*)
Get user tags + default tags to add to the instances and volumes.

update_tags (*self*)
Update user provided tags to all hosts.

In case there is an existing cluster that do not contain all the tags, by executing this all hosts will have the user specified tags.

This won't remove existing tags in the hosts.

_update_tags (*self, boto_instance: boto3.resources.factory.ec2.Instance, tags: Dict[str, str]*)
Create/Overwrite tags on an EC2 instance and its volumes.

Parameters

- **boto_instance** (*'boto3.resources.factory.ec2.Instance'*) – The EC2 instance
- **tags** (*Dict[str, str]*) – The tags to create/overwrite

name_instance (*self*, *boto_instance: boto3.resources.factory.ec2.Instance*, *name: str*)
Renames a EC2 instance

Parameters

- **boto_instance** (*'boto3.resources.factory.ec2.Instance'*) – The EC2 instance
- **name** (*str*) – The new name

_create_orchestrator (*self*)
Create a new EC2 instance to be the Orchestrator instance.

This new machine receives all tags defined in the *.ini file.

Returns The new orchestrator instance.

Return type instance.AWSOrchestratorInstance

_create_factories (*self*, *number: int = 1*)
Creates new AWS EC2 instances to be the Factory instances.

These new machines receive all tags defined in the *.ini file. Factory instances will be named using the factory basename plus an index. For example, "seq2seq_factory_0", "seq2seq_factory_1".

Parameters **number** (*int*) – The number of factories to be created.

Returns The new factory instances.

Return type List[instance.AWSGPUFactoryInstance]

_generic_launch_instances(*self, instance_class: Type[T], number: int, instance_type: str, instance_ami: str, role: str*)
Generic method to launch instances in AWS EC2 using boto3.

This method should not be used outside this module.

Parameters

- **instance_class** (*Type[T]*) – The instance class. It can be AWSOrchestratorInstance or AWSGPUFactoryInstance.
- **number** (*int*) – The amount of instances to create
- **instance_type** (*str*) – The instance type
- **instance_ami** (*str*) – The AMI to be used. Should be an Ubuntu 18.04 based AMI.
- **role** (*str*) – Wether is 'Orchestrator' or 'Factory'

Returns The new Instances.

Return type List[Union[AWSOrchestratorInstance, AWSGPUFactoryInstance]]

_get_boto_public_host(*self*, *boto_ins: boto3.resource.factory.ec2.Instance*)
Return the boto instance IP or DNS that will be used by the local process to reach the current instance.

This method abstracts the way the local process will access the instances in the case it's not the public IP.

Parameters **boto_ins** (*'boto3.resources.factory.ec2.Instance'*) – The boto instance

Returns The host information.

Return type str

_get_boto_private_host(*self*, *boto_ins: boto3.resource.factory.ec2.Instance*)
Return the boto instance IP or DNS that will be used by the other instances to reach the current instance.

This method abstracts the way the other instances will access the instance in the case it's not the private IP.

Parameters **boto_ins** (*'boto3.resources.factory.ec2.Instance'*) – The boto instance

Returns The host information.

Return type str

terminate_instances(*self*)
Terminates all instances.

rollback_env(*self*)
Rollback the environment.

This occurs when an error is caucht during the local stage of the remote experiment (i.e. creating the cluster, sending the data and submitting jobs), this method handles cleanup stages.

parse(*self*)
Checks if the AWSCluster configuration is valid.

This checks that the factories are never terminated after the orchestrator is. Avoids the scenario where the cluster has only factories and no orchestrator, which is useless.

Raises errors.ClusterConfigurationError – If configuration is not valid.

_get_boto_instance_by_host(*self*, *public_host: str*)
Returns the instance id given the public host

This method will use *_get_boto_public_host* to search for the given host.

Parameters **public_host** (*str*) – The host. Depending on how the host was set, it can be an IP or DNS.

Returns The id if found else None

Return type Optional[boto3.resources.factory.ec2.Instance]

_get_instance_id_by_host(*self*, *public_host: str*)
Returns the instance id given the public host

Parameters **public_host** (*str*) – The host. Depending on how the host was set, it can be an IP or DNS.

Returns The id if found else None

Return type Optional[str]

_get_alarm_name(*self*, *instance_id: str*)
Get the alarm name to be used for the given instance.

Parameters **instance_id** (*str*) – The id of the instance

Returns The name of the corresponding alarm

Return type str

has_alarm(*self*, *instance_id: str*)
Whether the instance has an alarm set.

Parameters **instance_id** (*str*) – The id of the instance

Returns True if an alarm is set. False otherwise.

Return type bool

remove_existing_events(*self*)
Remove the current alarm.

In case the orchestrator or factories had an alarm, we remove it to reset the new policies.

create_cloudwatch_events(*self*)
Creates cloudwatch events for orchestrator and factories.

_delete_cloudwatch_event(*self*, *instance_id: str*)
Deletes the alarm related to the instance.

_put_fake_cloudwatch_data(*self*, *instance_id: str*, *value: int = 100*, *points: int = 10*)
Put fake CPU Usage metric in an instance.

This method is useful to avoid triggering alarms when they are created. For example, is an instance was idle for 10 hours and an termination alarm is set for 5 hours, it will be triggered immediately. Adding a fake point will allow the alarms to start the timer from the current moment.

Parameters

- **instance_id** (*str*) – The ID of the EC2 instance
- **value** (*int*) – The CPU percent value to use. Defaults to 100
- **points** (*int*) – The amount of past minutes from the current time to generate metric points. For example, if points is 10, then 10 data metrics will be generated for the past 10 minutes, one per minute.

_create_cloudwatch_event(*self*, *instance_id: str*, *mins: int = 60*, *cpu_thresh: float = 0.1*)
Create CloudWatch alarm.

The alarm is used to terminate an instance based on CPU usage.

Parameters

- **instance_id** (*str*) – The ID of the EC2 instance
- **mins** (*int*) – Number of minutes to trigger the termination event. The evaluation preriod will be always one minute.
- **cpu_thresh** (*float*) – Percentage specifying upper bound for triggering event. If mins is 60 and cpu_thresh is 0.1, then this instance will be deleted after 1 hour of average CPU below 0.1.

_get_images (*self*)
Get the official AWS public AMIs created by Flambe.

ATTENTION: why not just search the tags? We need to make sure the AMIs we pick were created by the Flambe team. Because of tags values not being unique, anyone can create a public AMI with 'Creator: flambe@asapp.com' as a tag. If we pick that AMI, then we could potentially be Creating instances with unknown AMIs, causing potential security issues. By filtering by our acount id (which can be public), then we can make sure that all AMIs that are being scanned were created by Flambe team.

Returns The boto3 API response

Return type Dict

_get_ami (*self*, *_type: str*, *version: str*)
Given a type and a version, get the correct Flambe AMI.

IMPORTANT: we keep the version logic in case we add versioned AMIs in the future.

Parameters

- **_type** (*str*) – It can be either 'factory' or 'orchestrator'. Note that the type is lowercase in the AMI tag.
- **version** (*str*) – For example, "0.2.1" or "2.0".

Returns

Return type The ImageId if it's found. None if not.

_find_default_ami (*self*, *_type: str*)
Returns an AMI with version 0.0.0, which is the default. This means that doesn't contain flambe itself but it has some heavy dependencies already installed (like pytorch).

Parameters **_type** (*str*) – Wether is "orchestrator" or "factory"

Returns The ImageId or None if not found.

Return type Optional[str]

19.2.2 flambe.cluster.cluster

This module contains the base implementation of a Cluster.

A Cluster is in charge of dealing with the different Instance objects that will be part of the remote runnable.

Module Contents

flambe.cluster.cluster.**logger**

flambe.cluster.cluster.**GPUFactoryInsT**

flambe.cluster.cluster.**CPUFactoryInsT**

flambe.cluster.cluster.**FactoryInsT**

flambe.cluster.cluster.**UPLOAD_WARN_LIMIT_MB = 10**

class flambe.cluster.cluster.**Cluster**(*name: str*, *factories_num: int*, *key: str*, *username: str*, *setup_cmds: Optional[List[str]] = None*)

Bases: *flambe.runnable.Runnable*

Basic implementation of a Cluster.

The cluster is in charge of creating the cluster of instances where one host is the Orchestrator while the other ones are Factories.

This implementation should not be used by an end user. In order to give support to a cloud service provider (ex: AWS), a child class must be implemented inheriting from the Cluster class.

Important: when possible, Clusters should context managers

Parameters

- **name** (*str*) – The name of the cluster, used to name the remote instances.
- **factories_num** (*int*) – The amount of factories to use. Note that this differs from the number of workers, as each factories can contain multiple GPUs and therefore, multiple workers.
- **key** (*str*) – The path to the ssh key used to communicate to all instances. IMPORTANT: all instances must be accessible with the same key.
- **username** (*str*) – The username of the instances the cluster will handle. IMPORTANT: for now all instances need to have the same username.
- **setup_cmds** (*Optional[List[str]]*) – A list of commands to be run on all hosts for setup purposes. These commands can be used to mount volumes, install software, etc. Defaults to None. IMPORTANT: the commands need to be idempotent and they shouldn't expect user input.

__enter__(*self*)

A Cluster should be used with a context cluster to handle all possible errors in a clear way.

Examples

```
>>> with cluster as cl:
>>>     cl.launch_orchestrator()
>>>     cl.build_cluster()
>>>     ...
```

__exit__(*self, exc_type: Optional[Type[BaseException]], exc_value: Optional[BaseException], tb: Optional[TracebackType]*)

Exit method for the context cluster.

This method will catch any exception, log it and return True. This means that all exceptions produced in a Cluster (used with the context cluster) will not continue to raise.

Returns True, as an exception should not continue to raise.

Return type Optional[bool]

get_orchestrator_name(*self*)

Get the orchestrator name.

The name is given by *name* with the '_orchestrator' suffix. For example, if name is 'seq2seq-en-fr', then the orchestrator name will be 'seq2seq-en-fr_orchestrator'.

This is an auxiliary method that can be used in child classes.

Returns The orcehstrator name

Return type str

get_factory_basename (*self*)
Get the factory base name.

The name is *name* with the '_factory' suffix. For example, if name is 'seq2seq-en-fr', then the factory basename will be 'seq2seq-en-fr_factory'.

The base name can be used to generate all the factories' names (for example, by also appending an index to the basename).

This is an auxiliary method that can be used in child classes.

Returns The factory basename

Return type str

load_all_instances (*self*)
Method to make all hosts accessible.

Depending on the Cluster type, it behaves differently. For example, AWSCluster or GCPCluster can create the instances in this step. The SSHCluster does nothing (the machines are already created).

_get_all_hosts (*self*)
Auxiliary method to get all the hosts in a list.append(

create_dirs (*self, relative_dirs: List[str]*)
Create folders in all hostss.

If some of the already exist, it will do nothing.

Parameters **relative_dirs** (*List[str]*) – The directories to create. They should be relative paths and $HOME of each host will be used to add the prefix.

prepare_all_instances (*self*)
Prepare all the instances (both orchestrator and factories).

This method assumes that the hosts are running and accesible. It will call the 'prepare' method from all hosts.

run (*self*, *force: bool = False*, ***kwargs*)
Run a cluster and load all the instances.

After this metho runs, the orchestrator and factories objects will be populated.

If a runnable is provided, then the cluster will execute the runnable remotely in the cluster. Currently, only ClusterRunnable is supported.

This method should be idempotent (ie if called N times with the same configuration, only one cluster will be created.)

Parameters **force** (*bool, defaults to False*) – If true, current executions of the same runnable in the cluster will be overriden by a new execution.

run_cmds (*self, setup_cmds: List[str]*)
Run setup commands in all hosts

Parameters **setup_cmds** (*List[str]*) – The list of commands

Raises `errors.RemoteCommandError` – If at least one commands is not successful in at least one host.

get_orchestrator(*self*, *ip: str*, *private_ip: str = None*, *use_public: bool = True*)
Get an orchestrator instance

get_orch_home_path(*self*)
Return the orchestrator home path

Returns

Return type str

get_factory(*self*, *ip: str*, *private_ip: str = None*, *use_public: bool = True*)
Get an CPU factory instance

get_gpu_factory(*self*, *ip: str*, *private_ip: str = None*, *use_public: bool = True*)
Get an GPU factory instance

launch_ray_cluster(*self*)
Create a ray cluster.

The main node is going to be located in the orchestrator machine and all other nodes in the factories.

The main node is executed with –num-cpus=0 flag so that it doesn't do any work and all work is done by the factories.

check_ray_cluster(*self*)
Check if ray cluster was build successfully.

Compares the name of workers available with the requested ones.

Returns Whether the number of workers in the node matches the number of factories

Return type bool

shutdown_ray_cluster(*self*)
Shut down the ray cluster.

Shut down the main node running in the orchestrator.

existing_ray_cluster(*self*)
Return a list of the nodes in the Ray cluster.

Returns The list of nodes

Return type List[*Instance*]

existing_flambe_execution(*self*)
Return a list of the hosts that are running flambe.

Returns The list of nodes

Return type List[*Instance*]

shutdown_flambe_execution(*self*)
Shut down any flambe execution in the hosts.

existing_dir(*self*, *_dir: str*)
Determine if _dir exists in at least one host

is_ray_cluster_up(*self*)
Return if the ray cluster is running.

Returns

Return type bool

rollback_env (*self*)
Rollback the enviornment.

When an error occures during the local stage of the remote runnable (i.e. creating the cluster, sending the data and submitting jobs), this method may be used to destroy the cluster that has been built.

parse (*self*)
Parse the cluster object.

Look for configurations mistakes that don't allow the remote runnable to run. Each different cluster will have it's own policies. For example, AWSCluster could check the instance types that are allowed. By default, checks nothing.

Raises `man_errors.ClusterConfigurationError` – In case the Runnable is not able to run.

send_local_content (*self, content: Dict[str, str], dest: str, all_hosts: bool = False*)
Send local content to the cluster

Parameters

- **content** (*Dict[str, str]*) – The dict of resources key -> local path
- **dest** (*str*) – The orchestator's destination folder
- **all_hosts** (*bool*) – If False, only send the content to the orchestrator. If True, send to all factories.

Returns The new dict of content with orchestrator's paths.

Return type Dict[str, str]

rsync_orch (*self*, *folder*)
Rsync the orchestrator's folder with all factories

Parameters **folder** (*str*) – The folder to rsync. It should be a relative path. $HOME value will be automatically added.

send_secrets (*self*, *whitelist: List[str] = None*)
Send the secrets file to the orchestrator.

This file will be located in $HOME/secrets.ini The injected secrets file will be used.

Parameters **whitelist** (*List[str]*) – A list of sections to filter. For example: ["AWS", "GITHUB"]

execute (*self, cluster_runnable, extensions: Dict[str, str], new_secrets: str, force: bool*)
Execute a ClusterRunnable in the cluster.

It will first upload the runnable file + extensions to the orchestrator (under $HOME/flambe.yaml) and then it will execute it based on the provided secrets

Parameters

- **cluster_runnable** (ClusterRunnable) – The ClusterRunnable to run in the cluster
- **extensions** (*Dict[str, str]*) – The extensions for the ClusterRunnable
- **new_secrets** (*str*) – The path (relative to the orchestrator) where the secrets are located. IMPORTANT: previous to calling this method, the secrets should have been uploaded to the orchestrator
- **force** (*bool*) – The force parameter provided when running flambe locally

remove_dir (*self*, *_dir: str*, *content_only: bool = True*, *all_hosts: bool = True*)
Remove a directory in the ClusterError

Parameters

- **_dir** (*str*) – The directory to remove
- **content_only** (*bool*) – To remove the content only or the folder also. Defaults to True.
- **all_hosts** (*bool*) – To remove it in all hosts or only in the Orchestrator. Defaults to True (in all hosts).

cluster_has_key (*self*)
Whether the cluster already contains a valid common key.

The key must be in all hosts.

Returns If the cluster has a key in all hosts.

Return type bool

distribute_keys (*self*)
Create a new key pair and distributes it to all hosts.

Ensure that the hosts have a safe communication. The name of the key is the cluster's name

contains_gpu_factories (*self*)
Return if the factories contain GPU.

For now, all factories are same machine type, so as soon as a GPU is found, then this method returns.

get_max_resources (*self*)
Return the max common CPU/GPU devices in the factories

For example, if one factory contains 32 CPU + 1 GPU and the other factory contains 16 CPU + 2 GPU, this method will return {"cpu": 16, "gpu": 1} available

Returns The devices, in {"cpu": N, "gpu": M} format

Return type Dict[str, int]

install_extensions_in_orchestrator (*self, extensions: Dict[str, str]*)
Install local + pypi extensions in the orchestrator

Parameters **extension** (*Dict[str, str]*) – The extensions, as a dict from module_name to location

Raises

- errors.RemoteCommandError – If could not install an extension.
- man_errors.ClusterError – If the orchestrator was not loaded.

install_extensions_in_factories (*self, extensions: Dict[str, str]*)
Install local + pypi extensions in all the factories.

Parameters **extension** (*Dict[str, str]*) – The extensions, as a dict from module_name to location

Raises errors.RemoteCommandError – If could not install an extension

get_remote_env (*self, user_provider: Callable[[], str]*)
Get the RemoteEnvironment for this cluster.

The IPs stored will be the private IPs

Returns The RemoteEnvironment with information about this cluster.

Return type *RemoteEnvironment*

19.2.3 flambe.cluster.const

Module Contents

flambe.cluster.const.**SOCKET_TIMEOUT = 50**

flambe.cluster.const.**RETRY_DELAY = 1**

flambe.cluster.const.**RETRIES = 60**

flambe.cluster.const.**TENSORBOARD_IMAGE = tensorflow/tensorflow:1.15.0**

flambe.cluster.const.**RAY_REDIS_PORT = 12345**

flambe.cluster.const.**PRIVATE_KEY = ray_bootstrap_key.pem**

flambe.cluster.const.**PUBLIC_KEY = ray_bootstrap_key.pub**

flambe.cluster.const.**REPORT_SITE_PORT = 49558**

flambe.cluster.const.**TENSORBOARD_PORT = 49556**

flambe.cluster.const.**AWS_FLAMBE_ACCOUNT = 808129580301**

19.2.4 flambe.cluster.errors

Module Contents

exception flambe.cluster.errors.**ClusterError**
Bases: Exception

Error raised in case of any unexpected error in the Ray cluster.

exception flambe.cluster.errors.**ClusterConfigurationError**
Bases: Exception

Error raised when the configuration of the Cluster is not valid.

19.2.5 flambe.cluster.ssh

Implementation of the Manager for SSH hosts

Module Contents

flambe.cluster.ssh.**logger**

flambe.cluster.ssh.**FactoryT**

class flambe.cluster.ssh.**SSHCluster**(*name: str, orchestrator_ip: Union[str, List[str]], factories_ips: Union[List[str], List[List[str]]], key: str, username: str, remote_context=None, use_public: bool = True, setup_cmds: Optional[List[str]] = None*)
Bases: *flambe.cluster.cluster.Cluster*

The SSH Manager needs to be used when having running instances.

For example when having on-prem hardware or just a couple of AWS EC2 instances running.

When using this cluster, the user needs to specify the IPs of the machines to use, both the public one and private one.

load_all_instances (*self*, *exp_name: str = None*, *force: bool = False*)
This manager assumed that instances are running.

This method loads the Python objects to the manager's variables.

Parameters

- **exp_name** (*str*) – The name of the experiment
- **force** (*bool*) – Whether to override the current experiment of the same name

rollback_env (*self*)

rsync_hosts (*self*)
Rsyncs the host's result folders.

First, it rsyncs all worker folders to the orchestrator main folder. After that, so that every worker gets the last changes, the orchestrator rsync with all of them.

19.2.6 flambe.cluster.utils

Module Contents

flambe.cluster.utils.**RemoteCommand**

19.3 Package Contents

class flambe.cluster.**Cluster**(*name: str*, *factories_num: int*, *key: str*, *username: str*, *setup_cmds: Optional[List[str]] = None*)
Bases: *flambe.runnable.Runnable*

Basic implementation of a Cluster.

The cluster is in charge of creating the cluster of instances where one host is the Orchestrator while the other ones are Factories.

This implementation should not be used by an end user. In order to give support to a cloud service provider (ex: AWS), a child class must be implemented inheriting from the Cluster class.

Important: when possible, Clusters should context managers

Parameters

- **name** (*str*) – The name of the cluster, used to name the remote instances.
- **factories_num** (*int*) – The amount of factories to use. Note that this differs from the number of workers, as each factories can contain multiple GPUs and therefore, multiple workers.
- **key** (*str*) – The path to the ssh key used to communicate to all instances. IMPORTANT: all instances must be accessible with the same key.
- **username** (*str*) – The username of the instances the cluster will handle. IMPORTANT: for now all instances need to have the same username.

- **setup_cmds** (*Optional[List[str]]*) – A list of commands to be run on all hosts for setup purposes. These commands can be used to mount volumes, install software, etc. Defaults to None. IMPORTANT: the commands need to be idempotent and they shouldn't expect user input.

__enter__(*self*)
A Cluster should be used with a context cluster to handle all possible errors in a clear way.

Examples

```
>>> with cluster as cl:
>>>     cl.launch_orchestrator()
>>>     cl.build_cluster()
>>>     ...
```

__exit__(*self, exc_type: Optional[Type[BaseException]], exc_value: Optional[BaseException], tb: Optional[TracebackType]*)
Exit method for the context cluster.

This method will catch any exception, log it and return True. This means that all exceptions produced in a Cluster (used with the context cluster) will not continue to raise.

Returns True, as an exception should not continue to raise.

Return type Optional[bool]

get_orchestrator_name(*self*)
Get the orchestrator name.

The name is given by *name* with the '_orchestrator' suffix. For example, if name is 'seq2seq-en-fr', then the orchestrator name will be 'seq2seq-en-fr_orchestrator'.

This is an auxiliary method that can be used in child classes.

Returns The orcehstrator name

Return type str

get_factory_basename(*self*)
Get the factory base name.

The name is *name* with the '_factory' suffix. For example, if name is 'seq2seq-en-fr', then the factory basename will be 'seq2seq-en-fr_factory'.

The base name can be used to generate all the factories' names (for example, by also appending an index to the basename).

This is an auxiliary method that can be used in child classes.

Returns The factory basename

Return type str

load_all_instances(*self*)
Method to make all hosts accessible.

Depending on the Cluster type, it behaves differently. For example, AWSCluster or GCPCluster can create the instances in this step. The SSHCluster does nothing (the machines are already created).

_get_all_hosts(*self*)
Auxiliary method to get all the hosts in a list.append(

create_dirs (*self, relative_dirs: List[str]*)
Create folders in all hostss.

If some of the already exist, it will do nothing.

Parameters **relative_dirs** (*List[str]*) – The directories to create. They should be relative paths and $HOME of each host will be used to add the prefix.

prepare_all_instances (*self*)
Prepare all the instances (both orchestrator and factories).

This method assumes that the hosts are running and accesible. It will call the 'prepare' method from all hosts.

run (*self*, *force: bool = False*, ***kwargs*)
Run a cluster and load all the instances.

After this metho runs, the orchestrator and factories objects will be populated.

If a runnable is provided, then the cluster will execute the runnable remotely in the cluster. Currently, only ClusterRunnable is supported.

This method should be idempotent (ie if called N times with the same configuration, only one cluster will be created.)

Parameters **force** (*bool, defaults to False*) – If true, current executions of the same runnable in the cluster will be overriden by a new execution.

run_cmds (*self, setup_cmds: List[str]*)
Run setup commands in all hosts

Parameters **setup_cmds** (*List[str]*) – The list of commands

Raises errors.RemoteCommandError – If at least one commands is not successful in at least one host.

get_orchestrator (*self*, *ip: str*, *private_ip: str = None*, *use_public: bool = True*)
Get an orchestrator instance

get_orch_home_path (*self*)
Return the orchestrator home path

Returns

Return type str

get_factory (*self*, *ip: str*, *private_ip: str = None*, *use_public: bool = True*)
Get an CPU factory instance

get_gpu_factory (*self*, *ip: str*, *private_ip: str = None*, *use_public: bool = True*)
Get an GPU factory instance

launch_ray_cluster (*self*)
Create a ray cluster.

The main node is going to be located in the orchestrator machine and all other nodes in the factories.

The main node is executed with –num-cpus=0 flag so that it doesn't do any work and all work is done by the factories.

check_ray_cluster (*self*)
Check if ray cluster was build successfully.

Compares the name of workers available with the requested ones.

Returns Whether the number of workers in the node matches the number of factories

Return type bool

shutdown_ray_cluster(*self*)
Shut down the ray cluster.

Shut down the main node running in the orchestrator.

existing_ray_cluster(*self*)
Return a list of the nodes in the Ray cluster.

Returns The list of nodes

Return type List[*Instance*]

existing_flambe_execution(*self*)
Return a list of the hosts that are running flambe.

Returns The list of nodes

Return type List[*Instance*]

shutdown_flambe_execution(*self*)
Shut down any flambe execution in the hosts.

existing_dir(*self*, *_dir: str*)
Determine if _dir exists in at least one host

is_ray_cluster_up(*self*)
Return if the ray cluster is running.

Returns

Return type bool

rollback_env(*self*)
Rollback the enviornment.

When an error occures during the local stage of the remote runnable (i.e. creating the cluster, sending the data and submitting jobs), this method may be used to destroy the cluster that has been built.

parse(*self*)
Parse the cluster object.

Look for configurations mistakes that don't allow the remote runnable to run. Each different cluster will have it's own policies. For example, AWSCluster could check the instance types that are allowed. By default, checks nothing.

Raises `man_errors.ClusterConfigurationError` – In case the Runnable is not able to run.

send_local_content(*self, content: Dict[str, str], dest: str, all_hosts: bool = False*)
Send local content to the cluster

Parameters

- **content** (*Dict[str, str]*) – The dict of resources key -> local path
- **dest** (*str*) – The orchestator's destination folder
- **all_hosts** (*bool*) – If False, only send the content to the orchestrator. If True, send to all factories.

Returns The new dict of content with orchestrator's paths.

Return type Dict[str, str]

rsync_orch (*self*, *folder*)
Rsync the orchestrator's folder with all factories

Parameters **folder** (*str*) – The folder to rsync. It should be a relative path. $HOME value will be automatically added.

send_secrets (*self*, *whitelist: List[str] = None*)
Send the secrets file to the orchestrator.

This file will be located in $HOME/secrets.ini The injected secrets file will be used.

Parameters **whitelist** (*List[str]*) – A list of sections to filter. For example: ["AWS", "GITHUB"]

execute (*self, cluster_runnable, extensions: Dict[str, str], new_secrets: str, force: bool*)
Execute a ClusterRunnable in the cluster.

It will first upload the runnable file + extensions to the orchestrator (under $HOME/flambe.yaml) and then it will execute it based on the provided secrets

Parameters

- **cluster_runnable** (ClusterRunnable) – The ClusterRunnable to run in the cluster
- **extensions** (*Dict[str, str]*) – The extensions for the ClusterRunnable
- **new_secrets** (*str*) – The path (relative to the orchestrator) where the secrets are located. IMPORTANT: previous to calling this method, the secrets should have been uploaded to the orchestrator
- **force** (*bool*) – The force parameter provided when running flambe locally

remove_dir (*self*, *_dir: str*, *content_only: bool = True*, *all_hosts: bool = True*)
Remove a directory in the ClusterError

Parameters

- **_dir** (*str*) – The directory to remove
- **content_only** (*bool*) – To remove the content only or the folder also. Defaults to True.
- **all_hosts** (*bool*) – To remove it in all hosts or only in the Orchestrator. Defaults to True (in all hosts).

cluster_has_key (*self*)
Whether the cluster already contains a valid common key.

The key must be in all hosts.

Returns If the cluster has a key in all hosts.

Return type bool

distribute_keys (*self*)
Create a new key pair and distributes it to all hosts.

Ensure that the hosts have a safe communication. The name of the key is the cluster's name

contains_gpu_factories (*self*)
Return if the factories contain GPU.

For now, all factories are same machine type, so as soon as a GPU is found, then this method returns.

get_max_resources (*self*)
Return the max common CPU/GPU devices in the factories

For example, if one factory contains 32 CPU + 1 GPU and the other factory contains 16 CPU + 2 GPU, this method will return {"cpu": 16, "gpu": 1} available

Returns The devices, in {"cpu": N, "gpu": M} format

Return type Dict[str, int]

install_extensions_in_orchestrator (*self, extensions: Dict[str, str]*)
Install local + pypi extensions in the orchestrator

Parameters **extension** (*Dict[str, str]*) – The extensions, as a dict from module_name to location

Raises

- errors.RemoteCommandError – If could not install an extension.
- man_errors.ClusterError – If the orchestrator was not loaded.

install_extensions_in_factories (*self, extensions: Dict[str, str]*)
Install local + pypi extensions in all the factories.

Parameters **extension** (*Dict[str, str]*) – The extensions, as a dict from module_name to location

Raises errors.RemoteCommandError – If could not install an extension

get_remote_env (*self, user_provider: Callable[[], str]*)
Get the RemoteEnvironment for this cluster.

The IPs stored will be the private IPs

Returns The RemoteEnvironment with information about this cluster.

Return type *RemoteEnvironment*

class flambe.cluster.**AWSCluster**(*name: str, factories_num: int, factories_type: str, orchestrator_type: str, key_name: str, security_group: str, subnet_id: str, creator: str, key: str, volume_type: str = 'gp2', region_name: Optional[str] = None, username: str = 'ubuntu', tags: Dict[str, str] = None, orchestrator_ami: str = None, factory_ami: str = None, dedicated: bool = False, orchestrator_timeout: int = -1, factories_timeout: int = 1, volume_size: int = 100, setup_cmds: Optional[List[str]] = None*)
Bases: *flambe.cluster.cluster.Cluster*

This Cluster implementation uses AWS EC2 as the cloud provider.

This cluster works with AWS Instances that are defined in: *flambe.remote.instance.aws*

Parameters

- **name** (*str*) – The unique name for the cluster
- **factories_num** (*int*) – The amount of factories to use. This is not the amount of workers, as each factories can contain multiple GPUs and therefore, multiple workers.
- **factories_type** (*str*) – The type of instance to use for the Factory Instances. GPU instances are required for AWS the AWSCluster. "p2" and "p3" instances are recommended.
- **factory_ami** (*str*) – The AMI to be used for the Factory instances. Custom Flambe AMI are provided based on Ubuntu 18.04 distribution.

- **orchestrator_type** (*str*) – The type of instance to use for the Orchestrator Instances. This may not be a GPU instances. At least a "t2.small" instance is recommended.
- **key_name** (*str*) – The key name that will be used to connect into the instance.
- **creator** (*str*) – The creator should be a user identifier for the instances. This information will create a tag called 'creator' and it will also be used to retrieve existing hosts owned by the user.
- **key** (*str*) – The path to the ssh key used to communicate to all instances. IMPORTANT: all instances must be accessible with the same key.
- **volume_type** (*str*) – The type of volume in AWS to use. Only 'gp2' and 'io1' are currently available. If 'io1' is used, then IOPS will be fixed to 5000. IMPORTANT: 'io1' volumes are significantly more expensive than 'gp2' volumes. Defaults to 'gp2'.
- **region_name** (*Optional[str]*) – The region name to use. If not specified, it uses the locally configured region name or 'us-east-1' in case it's not configured.
- **username** (*str*) – The username of the instances the cluster will handle. Defaults to 'ubuntu'. IMPORTANT: for now all instances need to have the same username.
- **tags** (*Dict[str, str]*) – A dictionary with tags that will be added to all created hosts.
- **security_group** (*str*) – The security group to use to create the instances.
- **subnet_id** (*str*) – The subnet ID to use.
- **orchestrator_ami** (*str*) – The AMI to be used for the Factory instances. Custom Flambe AMI are provided based on Ubuntu 18.04 distribution.
- **dedicated** (*bool*) – Wether all created instances are dedicated instances or shared.
- **orchestrator_timeout** (*int*) – Number of consecutive hours before terminating the orchestrator once the experiment is over (either success of failure). Specify -1 to disable automatic shutdown (the orchestrator will stay on until manually terminated) and 0 to shutdown when the experiment is over. For example, if specifying 24, then the orchestrator will be shut down one day after the experiment is over. ATTENTION: This also applies when the experiment ends with an error. Default is -1.
- **factories_timeout** (*int*) – Number of consecutive hours to automatically terminate factories once the experiment is over (either success or failure). Specify -1 to disable automatic shutdown (the factories will stay on until manually terminated) and 0 to shutdown when the experiment is over. For example, if specifying 10, then the factories will be shut down 10 hours after the experiment is over. ATTENTION: This also applies when the experiment ends with an error. Default is 1.
- **volume_size** (*int*) – The disk size in GB that all hosts will contain. Defaults to 100 GB.
- **setup_cmds** (*Optional[List[str]]*) – A list of commands to be run on all hosts for setup purposes. These commands can be used to mount volumes, install software, etc. Defaults to None. IMPORTANT: the commands need to be idempotent and they shouldn't expect user input.

_get_boto_session(*self, region_name: Optional[str]*)

Get the boto3 Session from which the resources and clients will be created.

This method is called by the contructor.

Parameters **region_name** (*Optional[str]*) – The region to use. If None, boto3 will resolve to the locally configured region_name or 'us-east-1' if not configured.

Returns The boto3 Session to use

Return type boto3.Session

load_all_instances (*self*)
Launch all instances for the experiment.

This method launches both the orchestrator and the factories.

_existing_cluster (*self*)
Whether there is an existing cluster that matches name.

The cluster should also match all other tags, including Creator)

Returns Returns the (boto_orchestrator, [boto_factories]) that match the experiment's name.

Return type Tuple[Any, List[Any]]

_get_existing_tags (*self*, *boto_instance: boto3.resources.factory.ec2.Instance*)
Gets the tags of a EC2 instances

Parameters **boto_instance** (*BotoIns*) – The EC2 instance to access the tags.

Returns Key, Value for the specified tags.

Return type Dict[str, str]

flambe_own_running_instances (*self*)
Get running instances with matching tags.

Yields *Tuple['boto3.resources.factory.ec2.Instance', str]* – A tuple with the instance and the name of the EC2 instance.

name_hosts (*self*)
Name the orchestrator and factories.

_get_all_tags (*self*)
Get user tags + default tags to add to the instances and volumes.

update_tags (*self*)
Update user provided tags to all hosts.

In case there is an existing cluster that do not contain all the tags, by executing this all hosts will have the user specified tags.

This won't remove existing tags in the hosts.

_update_tags (*self, boto_instance: boto3.resources.factory.ec2.Instance, tags: Dict[str, str]*)
Create/Overwrite tags on an EC2 instance and its volumes.

Parameters

- **boto_instance** (*'boto3.resources.factory.ec2.Instance'*) – The EC2 instance
- **tags** (*Dict[str, str]*) – The tags to create/overwrite

name_instance (*self*, *boto_instance: boto3.resources.factory.ec2.Instance*, *name: str*)
Renames a EC2 instance

Parameters

- **boto_instance** (*'boto3.resources.factory.ec2.Instance'*) – The EC2 instance
- **name** (*str*) – The new name

_create_orchestrator (*self*)
Create a new EC2 instance to be the Orchestrator instance.

This new machine receives all tags defined in the *.ini file.

Returns The new orchestrator instance.

Return type instance.AWSOrchestratorInstance

_create_factories (*self*, *number: int = 1*)
Creates new AWS EC2 instances to be the Factory instances.

These new machines receive all tags defined in the *.ini file. Factory instances will be named using the factory basename plus an index. For example, "seq2seq_factory_0", "seq2seq_factory_1".

Parameters **number** (*int*) – The number of factories to be created.

Returns The new factory instances.

Return type List[instance.AWSGPUFactoryInstance]

_generic_launch_instances (*self, instance_class: Type[T], number: int, instance_type: str, instance_ami: str, role: str*)
Generic method to launch instances in AWS EC2 using boto3.

This method should not be used outside this module.

Parameters

- **instance_class** (*Type[T]*) – The instance class. It can be AWSOrchestratorInstance or AWSGPUFactoryInstance.
- **number** (*int*) – The amount of instances to create
- **instance_type** (*str*) – The instance type
- **instance_ami** (*str*) – The AMI to be used. Should be an Ubuntu 18.04 based AMI.
- **role** (*str*) – Wether is 'Orchestrator' or 'Factory'

Returns The new Instances.

Return type List[Union[AWSOrchestratorInstance, AWSGPUFactoryInstance]]

_get_boto_public_host (*self*, *boto_ins: boto3.resource.factory.ec2.Instance*)
Return the boto instance IP or DNS that will be used by the local process to reach the current instance.

This method abstracts the way the local process will access the instances in the case it's not the public IP.

Parameters **boto_ins** (*'boto3.resources.factory.ec2.Instance'*) – The boto instance

Returns The host information.

Return type str

_get_boto_private_host (*self*, *boto_ins: boto3.resource.factory.ec2.Instance*)
Return the boto instance IP or DNS that will be used by the other instances to reach the current instance.

This method abstracts the way the other instances will access the instance in the case it's not the private IP.

Parameters **boto_ins** (*'boto3.resources.factory.ec2.Instance'*) – The boto instance

Returns The host information.

Return type str

terminate_instances (*self*)
Terminates all instances.

rollback_env (*self*)
Rollback the environment.

This occurs when an error is caucht during the local stage of the remote experiment (i.e. creating the cluster, sending the data and submitting jobs), this method handles cleanup stages.

parse (*self*)
Checks if the AWSCluster configuration is valid.

This checks that the factories are never terminated after the orchestrator is. Avoids the scenario where the cluster has only factories and no orchestrator, which is useless.

Raises *errors.ClusterConfigurationError* – If configuration is not valid.

_get_boto_instance_by_host (*self*, *public_host: str*)
Returns the instance id given the public host

This method will use *_get_boto_public_host* to search for the given host.

Parameters **public_host** (*str*) – The host. Depending on how the host was set, it can be an IP or DNS.

Returns The id if found else None

Return type Optional[boto3.resources.factory.ec2.Instance]

_get_instance_id_by_host (*self*, *public_host: str*)
Returns the instance id given the public host

Parameters **public_host** (*str*) – The host. Depending on how the host was set, it can be an IP or DNS.

Returns The id if found else None

Return type Optional[str]

_get_alarm_name (*self*, *instance_id: str*)
Get the alarm name to be used for the given instance.

Parameters **instance_id** (*str*) – The id of the instance

Returns The name of the corresponding alarm

Return type str

has_alarm (*self*, *instance_id: str*)
Whether the instance has an alarm set.

Parameters **instance_id** (*str*) – The id of the instance

Returns True if an alarm is set. False otherwise.

Return type bool

remove_existing_events (*self*)
Remove the current alarm.

In case the orchestrator or factories had an alarm, we remove it to reset the new policies.

create_cloudwatch_events (*self*)
Creates cloudwatch events for orchestrator and factories.

_delete_cloudwatch_event (*self*, *instance_id: str*)
Deletes the alarm related to the instance.

_put_fake_cloudwatch_data (*self*, *instance_id: str*, *value: int = 100*, *points: int = 10*)
Put fake CPU Usage metric in an instance.

This method is useful to avoid triggering alarms when they are created. For example, is an instance was idle for 10 hours and an termination alarm is set for 5 hours, it will be triggered immediately. Adding a fake point will allow the alarms to start the timer from the current moment.

Parameters

- **instance_id** (*str*) – The ID of the EC2 instance
- **value** (*int*) – The CPU percent value to use. Defaults to 100
- **points** (*int*) – The amount of past minutes from the current time to generate metric points. For example, if points is 10, then 10 data metrics will be generated for the past 10 minutes, one per minute.

_create_cloudwatch_event (*self*, *instance_id: str*, *mins: int = 60*, *cpu_thresh: float = 0.1*)
Create CloudWatch alarm.

The alarm is used to terminate an instance based on CPU usage.

Parameters

- **instance_id** (*str*) – The ID of the EC2 instance
- **mins** (*int*) – Number of minutes to trigger the termination event. The evaluation preriod will be always one minute.
- **cpu_thresh** (*float*) – Percentage specifying upper bound for triggering event. If mins is 60 and cpu_thresh is 0.1, then this instance will be deleted after 1 hour of average CPU below 0.1.

_get_images (*self*)
Get the official AWS public AMIs created by Flambe.

ATTENTION: why not just search the tags? We need to make sure the AMIs we pick were created by the Flambe team. Because of tags values not being unique, anyone can create a public AMI with 'Creator: flambe@asapp.com' as a tag. If we pick that AMI, then we could potentially be Creating instances with unknown AMIs, causing potential security issues. By filtering by our acount id (which can be public), then we can make sure that all AMIs that are being scanned were created by Flambe team.

Returns The boto3 API response

Return type Dict

_get_ami (*self*, *_type: str*, *version: str*)
Given a type and a version, get the correct Flambe AMI.

IMPORTANT: we keep the version logic in case we add versioned AMIs in the future.

Parameters

- **_type** (*str*) – It can be either 'factory' or 'orchestrator'. Note that the type is lowercase in the AMI tag.
- **version** (*str*) – For example, "0.2.1" or "2.0".

Returns

Return type The ImageId if it's found. None if not.

_find_default_ami (*self*, *_type: str*)
Returns an AMI with version 0.0.0, which is the default. This means that doesn't contain flambe itself but it has some heavy dependencies already installed (like pytorch).

Parameters **_type** (*str*) – Wether is "orchestrator" or "factory"

Returns The ImageId or None if not found.

Return type Optional[str]

class flambe.cluster.**SSHCluster**(*name: str, orchestrator_ip: Union[str, List[str]], factories_ips: Union[List[str], List[List[str]]], key: str, username: str, remote_context=None, use_public: bool = True, setup_cmds: Optional[List[str]] = None*)

Bases: *flambe.cluster.cluster.Cluster*

The SSH Manager needs to be used when having running instances.

For example when having on-prem hardware or just a couple of AWS EC2 instances running.

When using this cluster, the user needs to specify the IPs of the machines to use, both the public one and private one.

load_all_instances(*self*, *exp_name: str = None*, *force: bool = False*)

This manager assumed that instances are running.

This method loads the Python objects to the manager's variables.

Parameters

- **exp_name** (*str*) – The name of the experiment
- **force** (*bool*) – Whether to override the current experiment of the same name

rollback_env(*self*)

rsync_hosts(*self*)

Rsyncs the host's result folders.

First, it rsyncs all worker folders to the orchestrator main folder. After that, so that every worker gets the last changes, the orchestrator rsync with all of them.

CHAPTER 20

flambe.compile

20.1 Submodules

20.1.1 flambe.compile.component

Module Contents

flambe.compile.component.**_EMPTY**

flambe.compile.component.**A**

flambe.compile.component.**C**

flambe.compile.component.**YAML_TYPES**

flambe.compile.component.**logger**

exception flambe.compile.component.**CompilationError**
Bases: Exception

exception flambe.compile.component.**LoadError**
Bases: Exception

class flambe.compile.component.**Schema**(*component_subclass: Type[C], _flambe_custom_factory_name: Optional[str] = None, **keywords: Any*)
Bases: MutableMapping[str, Any]

Holds and recursively initializes Component's with kwargs

Holds a Component subclass and keyword arguments to that class's compile method. When an instance is called it will perform the recursive compilation process, turning the nested structure of Schema's into initialized Component objects

Implements MutableMapping methods to facilitate inspection and updates to the keyword args. Implements dot-notation access to the keyword args as well.

Parameters

- **component_subclass** (*Type[Component]*) – Subclass of Component that will be compiled
- ****keywords** (*Any*) – kwargs passed into the Schema's *compile* method

Examples

Create a Schema from a Component subclass

```
>>> class Test(Component):
...     def __init__(self, x=2):
...         self.x = x
...
>>> tp = Schema(Test)
>>> t1 = tp()
>>> t2 = tp()
>>> assert t1 is t2  # the same Schema always gives you same obj
>>> tp = Schema(Test) # create a new Schema
>>> tp['x'] = 3
>>> t3 = tp()
>>> assert t1.x == 3  # dot notation works as well
```

component_subclass
Subclass of Schema that will be compiled

Type Type[*Component*]

keywords
kwargs passed into the Schema's *compile* method

Type Dict[str, Any]

__call__ (*self*, *stash: Optional[Dict[str, Any]] = None*, ***keywords: Any*)

add_extensions_metadata (*self*, *extensions: Dict[str, str]*)
Add extensions used when loading this schema and children

Uses `component_subclass.__module__` to filter for only the single relevant extension for this object; extensions relevant for children are saved only on those children schemas directly. Use `aggregate_extensions_metadata` to generate a dictionary of all extensions used in the object hierarchy.

aggregate_extensions_metadata (*self*)
Aggregate extensions used in object hierarchy

contains (*self*, *schema: Schema*, *original_link: Link*)

__setitem__ (*self*, *key: str*, *value: Any*)

__getitem__ (*self*, *key: str*)

__delitem__ (*self*, *key: str*)

__iter__ (*self*)

__len__ (*self*)

__getattr__ (*self*, *item: str*)

__setattr__ (*self*, *key: str*, *value: Any*)

__repr__(*self*)
Identical to super (schema), but sorts keywords

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str = ''*)

static serialize(*obj: Any*)
Return dictionary representation of schema

Includes yaml as a string, and extensions

Parameters **obj** (*Any*) – Should be schema or dict of schemas

Returns dictionary containing yaml and extensions dictionary

Return type Dict[str, Any]

static deserialize(*data: Dict[str, Any]*)
Construct Schema from dict returned by Schema.serialize

Parameters **data** (*Dict[str, Any]*) – dictionary returned by `Schema.serialize`

Returns Schema or dict of schemas (depending on yaml in `data`)

Return type Any

flambe.compile.component.**_link_root_obj :Optional['Component']**

flambe.compile.component.**_link_context_active = False**

flambe.compile.component.**_link_obj_stash :Dict[str, Any]**

class flambe.compile.component.**contextualized_linking**(*root_obj: Any*, *prefix: str*)
Context manager used to change the representation of links

Links are always defined in relation to some root object and an attribute path, so when representing some piece of a larger object all the links need to be redefined in relation to the target object

__enter__(*self*)

__exit__(*self*, *exc_type: Any*, *exc_value: Any*, *traceback: Any*)

class flambe.compile.component.**PickledDataLink**(*obj_id: str*, *value: Any = None*)
Bases: *flambe.compile.registrable.Registrable*

__call__(*self, stash: Dict[str, Any]*)

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)

classmethod from_yaml(*cls*, *constructor: Any*, *node: Any*, *factory_name: str*)

exception flambe.compile.component.**LinkError**
Bases: `Exception`

exception flambe.compile.component.**MalformedLinkError**
Bases: *flambe.compile.component.LinkError*

exception flambe.compile.component.**UnpreparedLinkError**
Bases: *flambe.compile.component.LinkError*

flambe.compile.component.**parse_link_str**(*link_str: str*) → Tuple[Sequence[str], Sequence[str]]
Parse link to extract schematic and attribute paths

Links should be of the format `obj[key1][key2].attr1.attr2` where obj is the entry point; in a pipeline, obj would be the stage name, in a single-object config obj would be the target keyword at the top level. The following keys surrounded in brackets traverse the nested dictionary structure that appears in the config; this is

intentonally analagous to how you would access properties in the dictionary when loaded into python. Then, you can use the dot notation to access the runtime instance attributes of the object at that location.

Parameters **link_str** (*str*) – Link to earlier object in the config of the format `obj[key1][key2].attr1.attr2`

Returns Tuple of the schematic and attribute paths respectively

Return type Tuple[Sequence[str], Sequence[str]]

Raises *MalformedLinkError* – If the link is written incorrectly

Examples

Examples should be written in doctest format, and should illustrate how to use the function/class. >>> parse_link_str('obj[key1][key2].attr1.attr2') (['obj', 'key1', 'key2'], ['attr1', 'attr2'])

`flambe.compile.component.`**`create_link_str`**(*schematic_path: Sequence[str], attr_path: Optional[Sequence[str]] = None*) → str

Create a string representation of the specified link

Performs the reverse operation of *parse_link_str()*

Parameters

- **schematic_path** (*Sequence[str]*) – List of entries corresponding to dictionary keys in a nested *Schema*
- **attr_path** (*Optional[Sequence[str]]*) – List of attributes to access on the target object (the default is None).

Returns The string representation of the schematic + attribute paths

Return type str

Raises *MalformedLinkError* – If the schematic_path is empty

Examples

Examples should be written in doctest format, and should illustrate how to use the function/class. >>> create_link_str(['obj', 'key1', 'key2'], ['attr1', 'attr2']) 'obj[key1][key2].attr1.attr2'

class `flambe.compile.component.`**`Link`**(*schematic_path: Sequence[str], attr_path: Optional[Sequence[str]] = None, target: Optional[Schema] = None, local: bool = True*)

Bases: *flambe.compile.registrable.Registrable*

Delayed access to another object in an object hiearchy

Currently only supported in the context of Experiment but this may be updated in a future release

A Link delays the access of some property, or the calling of some method, until the Link is called. Links can be passed directly into a Component subclass *compile*, Component's method called compile will automatically record the links and call them to access their values before running *__new__* and *__init__*. The recorded links will show up in the config if *yaml.dump()* is called on your object hierarchy. This typically happens when logging individual configs during a grid search, and when serializing between multiple processes.

For example, if the schematic path is ['model', 'encoder'] and the attribute path is ['rnn', 'hidden_size'] then before the link can be compiled, the target attribute should be set to point to the model schema (this is handled automatically by Experiment) then, during compilation the child schema 'encoder' will be accessed, and finally the attribute encoder.rnn.hidden_size will be returned

Parameters

- **schematic_path** (*Sequence[str]*) – Path to the relevant schema denoted by dictionary-like bracket access e.g. ['model', 'encoder']
- **attr_path** (*Sequence[str]*) – Path to the relevant attribute on the given schema (after it's been compiled) using standard attribute dot notation e.g. ['rnn', 'hidden_size']
- **target** (*Optional[Schema]*) – The root object corresponding to the first element in the schematic path; needs to be passed in here or set later before link can be resolved
- **local** (*bool*) – if true, changes tune convert behavior to insert a dummy link; used for links to global variables ("resources" in config) (defaults to True)

root_schema :str

__repr__(*self*)

__call__(*self*)

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)
Build contextualized link based on the root node

If the link refers to something inside of the current object hierarchy (as determined by the schema of `_link_root_obj`) then it will be represented as a link; if the link refers to something out-of-scope, i.e. not inside the current object hiearchy, then replace the link with the resolved value. If the value cannot be represented, pickle it and include a reference to its id in the object stash that will be saved alongside the config

classmethod from_yaml(*cls*, *constructor: Any*, *node: Any*, *factory_name: str*)

convert(*self*)

class flambe.compile.component.**FunctionCallLink**
Bases: *flambe.compile.component.Link*

Calls the link attribute instead of just accessing it

__call__(*self*)

flambe.compile.component.**K**

flambe.compile.component.**fill_defaults**(*kwargs: Dict[str, Any]*, *function: Callable[..., Any]*) → Dict[str, Any]
Use function signature to add missing kwargs to a dictionary

flambe.compile.component.**merge_kwargs**(*kwargs: Dict[str, Any]*, *compiled_kwargs: Dict[str, Any]*) → Dict[str, Any]
Replace non links in kwargs with corresponding compiled values

For every key in *kwargs* if the value is NOT a link and IS a Schema, replace with the corresponding value in *compiled_kwargs*

Parameters

- **kwargs** (*Dict[str, Any]*) – Original kwargs containing Links and Schemas
- **compiled_kwargs** (*Dict[str, Any]*) – Processes kwargs containing no links and no Schemas

Returns kwargs with links, but with Schemas replaced by compiled objects

Return type Dict[str, Any]

class flambe.compile.component.**Component**(**kwargs*)
Bases: *flambe.compile.registrable.Registrable*

Class which can be serialized to yaml and implements *compile*

IMPORTANT: ALWAYS inherit from Component BEFORE *torch.nn.Module*

Automatically registers subclasses via Registrable and facilitates immediate usage in YAML with tags. When loaded, subclasses' initialization is delayed; kwargs are wrapped in a custom schema called Schema that can be easily initialized later.

_flambe_version = 0.0.0

_config_str
Represent object's architecture as a YAML string

Includes the extensions relevant to the object as well; NOTE: currently this section may include a superset of the extensions actually needed, but this will be changed in a future release.

run(*self*)
Run a single computational step.

When used in an experiment, this computational step should be on the order of tens of seconds to about 10 minutes of work on your intended hardware; checkpoints will be performed in between calls to run, and resources or search algorithms will be updated. If you want to run everything all at once, make sure a single call to run does all the work and return False.

Returns True if should continue running later i.e. more work to do

Return type bool

metric(*self*)
Override this method to enable scheduling and searching.

Returns The metric to compare different variants of your Component

Return type float

register_attrs(*self*, **names: str*)
Set attributes that should be included in state_dict

Equivalent to overriding *obj._state* and *obj._load_state* to save and load these attributes. Recommended usage: call inside *__init__* at the end: *self.register_attrs(attr1, attr2, ...)* Should ONLY be called on existing attributes.

Parameters ***names** (*str*) – The names of the attributes to register

Raises AttributeError – If *self* does not have existing attribute with that name

static _state_dict_hook(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any]*)
Add metadata and recurse on Component children

This hook is used to integrate with the PyTorch *state_dict* mechanism; as either *nn.Module.state_dict* or *Component.get_state* recurse, this hook is responsible for adding Flambe specific metadata and recursing further on any Component children of *self* that are not also nn.Modules, as PyTorch will handle recursing to the latter.

Flambe specific metadata includes the class version specified in the *Component._flambe_version* class property, the name of the class, the source code, and the fact that this class is a *Component* and should correspond to a directory in our hiearchical save format

Finally, this hook calls a helper *_state* that users can implement to add custom state to a given class

Parameters

- **state_dict** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **prefix** (*str*) – The current prefix for new compound keys that reflects the location of this instance in the object hierarchy being represented
- **local_metadata** (*Dict[str, Any]*) – A subset of the metadata relevant just to this object and its children

Returns The modified state_dict

Return type type

Raises ExceptionName – Why the exception is raised.

_add_registered_attrs(*self*, *state_dict: State*, *prefix: str*)

_state(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any]*)
Add custom state to state_dict

Parameters

- **state_dict** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **prefix** (*str*) – The current prefix for new compound keys that reflects the location of this instance in the object hierarchy being represented
- **local_metadata** (*Dict[str, Any]*) – A subset of the metadata relevant just to this object and its children

Returns The modified state_dict

Return type *State*

get_state(*self*, *destination: Optional[State] = None*, *prefix: str = ''*, *keep_vars: bool = False*)
Extract PyTorch compatible state_dict

Adds Flambe specific properties to the state_dict, including special metadata (the class version, source code, and class name). By default, only includes state that PyTorch *nn.Module* includes (Parameters, Buffers, child Modules). Custom state can be added via the *_state* helper method which subclasses should override.

The metadata *_flambe_directories* indicates which objects are Components and should be a subdirectory in our hierarchical save format. This object will recurse on *Component* and *nn.Module* children, but NOT *torch.optim.Optimizer* subclasses, *torch.optim.lr_scheduler._LRScheduler* subclasses, or any other arbitrary python objects.

Parameters

- **destination** (*Optional[State]*) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **prefix** (*str*) – The current prefix for new compound keys that reflects the location of this instance in the object hierarchy being represented
- **keep_vars** (*bool*) – Whether or not to keep Variables (only used by PyTorch) (the default is False).

Returns The state_dict object

Return type *State*

Raises ExceptionName – Why the exception is raised.

_load_state_dict_hook(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any], strict: bool, missing_keys: List[Any], unexpected_keys: List[Any], error_msgs: List[Any]*)

Load flambe-specific state

Parameters

- **state_dict** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **prefix** (*str*) – The current prefix for new compound keys that reflects the location of this instance in the object hierarchy being represented
- **local_metadata** (*Dict[str, Any]*) – A subset of the metadata relevant just to this object and its children
- **strict** (*bool*) – Whether missing or unexpected keys should be allowed; should always be False in Flambe
- **missing_keys** (*List[Any]*) – Missing keys so far
- **unexpected_keys** (*List[Any]*) – Unexpected keys so far
- **error_msgs** (*List[Any]*) – Any error messages so far

Raises *LoadError* – If the state for some object does not have a matching major version number

_load_registered_attrs(*self*, *state_dict: State*, *prefix: str*)

_load_state(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any], strict: bool, missing_keys: List[Any], unexpected_keys: List[Any], error_msgs: List[Any]*)

Load custom state (that was included via *_state*)

Subclasses should override this function to add custom state that isn't normally included by PyTorch nn.Module

Parameters

- **state_dict** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **prefix** (*str*) – The current prefix for new compound keys that reflects the location of this instance in the object hierarchy being represented
- **local_metadata** (*Dict[str, Any]*) – A subset of the metadata relevant just to this object and its children
- **strict** (*bool*) – Whether missing or unexpected keys should be allowed; should always be False in Flambe
- **missing_keys** (*List[Any]*) – Missing keys so far
- **unexpected_keys** (*List[Any]*) – Unexpected keys so far
- **error_msgs** (*List[Any]*) – Any error messages so far

load_state(*self*, *state_dict: State*, *strict: bool = False*)

Load *state_dict* into *self*

Loads state produced by *get_state* into the current object, recursing on child *Component* and *nn.Module* objects

Parameters

- **state_dict** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **strict** (*bool*) – Whether missing or unexpected keys should be allowed; should ALWAYS be False in Flambe (the default is False).

Raises *LoadError* – If the state for some object does not have a matching major version number

classmethod load_from_path(*cls*, *path: str*, *map_location: Union[torch.device, str] = None*, *use_saved_config_defaults: bool = True*, ***kwargs: Any*)

save(*self*, *path: str*, ***kwargs: Any*)

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)

classmethod from_yaml(*cls: Type[C], constructor: Any, node: Any, factory_name: str*)

classmethod precompile(*cls: Type[C], **kwargs: Any*)

Change kwargs before compilation occurs.

This hook is called after links have been activated, but before calling the recursive initialization process on all other objects in kwargs. This is useful in a number of cases, for example, in Trainer, we compile several objects ahead of time and move them to the GPU before compiling the optimizer, because it needs to be initialized with the model parameters *after* they have been moved to GPU.

Parameters

- **cls** (*Type[C]*) – Class on which method is called
- ****kwargs** (*Any*) – Current kwargs that will be compiled and used to initialize an instance of cls after this hook is called

aggregate_extensions_metadata(*self*)

Aggregate extensions used in object hierarchy

TODO: remove or combine with schema implementation in refactor

classmethod compile(*cls: Type[C], _flambe_custom_factory_name: Optional[str] = None, _flambe_extensions: Optional[Dict[str, str]] = None, _flambe_stash: Optional[Dict[str, Any]] = None, **kwargs: Any*)

Create instance of cls after recursively compiling kwargs

Similar to normal initialization, but recursively initializes any arguments that should be compiled and allows overriding arbitrarily deep kwargs before initializing if needed. Also activates any Link instances passed in as kwargs, and saves the original kwargs for dumping to yaml later.

Parameters ****kwargs** (*Any*) – Keyword args that should be forwarded to the initialization function (a specified factory, or the normal *__new__* and *__init__* methods)

Returns An instance of the class *cls*

Return type C

flambe.compile.component.**dynamic_component**(*class_: Type[A], tag: str, tag_namespace: Optional[str] = None, parent_component_class: Type[Component] = Component*) → Type[Component]

Decorate given class, creating a dynamic *Component*

Creates a dynamic subclass of *class_* that inherits from *Component* so it will be registered with the yaml loader and receive the appropriate functionality (*from_yaml*, *to_yaml* and *compile*). *class_* should not implement any of the aforementioned functions.

Parameters

- **class** (*Type[A]*) – Class to register with yaml and the compilation system
- **tag** (*str*) – Tag that will be used with yaml
- **tag_namespace** (*str*) – Namespace aka the prefix, used. e.g. for *!torch.Adam* torch is the namespace

Returns New subclass of *_class* and *Component*

Return type Type[*Component*]

20.1.2 flambe.compile.const

Module Contents

flambe.compile.const.**STATE_DICT_DELIMETER = .**

flambe.compile.const.**FLAMBE_SOURCE_KEY = _flambe_source**

flambe.compile.const.**FLAMBE_CLASS_KEY = _flambe_class**

flambe.compile.const.**FLAMBE_CONFIG_KEY = _flambe_config**

flambe.compile.const.**FLAMBE_DIRECTORIES_KEY = _flambe_directories**

flambe.compile.const.**FLAMBE_STASH_KEY = _flambe_stash**

flambe.compile.const.**KEEP_VARS_KEY = keep_vars**

flambe.compile.const.**VERSION_KEY = _flambe_version**

flambe.compile.const.**HIGHEST_SERIALIZATION_PROTOCOL_VERSION = 1**

flambe.compile.const.**DEFAULT_SERIALIZATION_PROTOCOL_VERSION = 1**

flambe.compile.const.**DEFAULT_PROTOCOL = 2**

flambe.compile.const.**STATE_FILE_NAME = state.pt**

flambe.compile.const.**VERSION_FILE_NAME = version.txt**

flambe.compile.const.**SOURCE_FILE_NAME = source.py**

flambe.compile.const.**CONFIG_FILE_NAME = config.yaml**

flambe.compile.const.**STASH_FILE_NAME = stash.pkl**

flambe.compile.const.**PROTOCOL_VERSION_FILE_NAME = protocol_version.txt**

20.1.3 flambe.compile.downloader

Module Contents

flambe.compile.downloader.**logger**

flambe.compile.downloader.**s3_exists**(*url: ParseResult*) → bool

Return is an S3 resource exists.

Parameters **url** (*ParseResult*) – The parsed URL.

Returns True if it exists. False otherwise.

Return type bool

flambe.compile.downloader.**s3_remote_file**(*url: ParseResult*) → bool
Check if an existing S3 hosted artifact is a file or a folder.

Parameters **url** (*ParseResult*) – The parsed URL.

Returns True if it's a file, False if it's a folder.

Return type bool

flambe.compile.downloader.**download_s3_file**(*url: str*, *destination: str*) → None
Download an S3 file.

Parameters

- **url** (*str*) – The S3 URL. Should follow the format: 's3://<bucket-name>[/path/to/file]'
- **destination** (*str*) – The output file where to copy the content

flambe.compile.downloader.**http_exists**(*url: str*) → bool
Check if an HTTP/HTTPS file exists.

Parameters **url** (*str*) – The HTTP/HTTPS URL.

Returns True if the HTTP file exists

Return type bool

flambe.compile.downloader.**download_http_file**(*url: str*, *destination: str*) → None
Download an HTTP/HTTPS file.

Parameters

- **url** (*str*) – The HTTP/HTTPS URL.
- **destination** (*str*) – The output file where to copy the content. Needs to support binary writing.

flambe.compile.downloader.**download_s3_folder**(*url: str*, *destination: str*) → None
Download an S3 folder.

Parameters

- **url** (*str*) – The S3 URL. Should follow the format: 's3://<bucket-name>[/path/to/folder]'
- **destination** (*str*) – The output folder where to copy the content

flambe.compile.downloader.**download_manager**(*path: str*, *destination: Optional[str] = None*)
Manager for downloading remote URLs

Parameters

- **path** (*str*) – The remote URL to download. Currently, only S3 and http/https URLs are supported. In case it's already a local path, it yields the same path.
- **destination** (*Optional[str]*) – The path where the artifact will be downloaded (this includes the file/folder name also). In case of not given, a temporary directory will be used and the name of the artifact will be inferred from the path.

Examples

```
>>> with download_manager("https://host.com/my/file.zip") as path:
>>>     os.path.exists(path)
>>> True
```

Yields *str* – The new local path

20.1.4 flambe.compile.extensions

This module provides methods to orchestrate all extensions

Module Contents

flambe.compile.extensions.**logger**

flambe.compile.extensions.**download_extensions**(*extensions: Dict[str, str], container_folder: str*) → Dict[str, str]

Iterate through the extensions and download the remote urls.

Parameters

- **extensions** (*Dict[str, str]*) – The extensions that may contain both local or remote locations.
- **container_folder** (*str*) – The auxiliary folder where to download the remote repo

Returns A new extensions dict with the local paths instead of remote urls. The local paths contain the downloaded remote resources.

Return type Dict[str, str]

flambe.compile.extensions.**_download_remote_extension**(*extension_url: str, location: str*) → str

Download a remote hosted extension.

It fully supports github urls only (for now).

Parameters

- **extension_url** (*str*) – The github url pointing to an extension. For example: https://github.com/user/folder/tree/branch/path/to/ext
- **location** (*str*) – The location to download the repo

Returns The location of the installed package (which it could not match the location passed as parameter)

Return type str

flambe.compile.extensions.**_has_svn**() → bool

Return if the host has svn installed

flambe.compile.extensions.**_download_svn**(*svn_url: str, location: str, username: Optional[str] = None, password: Optional[str] = None*) → None

Use svn to download a specific folder inside a git repo.

This works only with remote Github repositories.

Parameters

- **svn_url** (*str*) – The github URL adapted to use the SVN protocol
- **location** (*str*) – The location to download the folder
- **username** (*str*) – The username
- **password** (*str*) – The password

flambe.compile.extensions.**install_extensions**(*extensions: Dict[str, str], user_flag: bool = False*) → None

Install extensions.

At this point, all extensions must be either local paths or valid pypi packages.

Remote extensions hosted in Github must have been download first.

Parameters

- **extensions** (*Dict[str, str]*) – Dictionary of extensions
- **user_flag** (*bool*) – Use –user flag when running pip install

flambe.compile.extensions.**is_installed_module**(*module_name: str*) → bool

Whether the module is installed.

Parameters **module_name** (*str*) – The name of the module to check for

Returns True if the module is installed locally, False otherwise.

Return type bool

flambe.compile.extensions.**import_modules**(*modules: Iterable[str]*) → None

Dinamically import modules

Parameters **modules** (*Iterable[str]*) – An iterable of strings containing the modules to import

flambe.compile.extensions.**setup_default_modules**()

20.1.5 flambe.compile.registrable

Module Contents

flambe.compile.registrable.**logger**

flambe.compile.registrable.**yaml**

flambe.compile.registrable.**_reg_prefix :Optional[str]**

flambe.compile.registrable.**R**

flambe.compile.registrable.**A**

flambe.compile.registrable.**RT**

exception flambe.compile.registrable.**RegistrationError**

Bases: Exception

Error thrown when acessing yaml tag on a non-registered class

Thrown when trying to access the default yaml tag for a class typically occurs when called on an abstract class

flambe.compile.registrable.**make_from_yaml_with_metadata**(*from_yaml_fn: Callable[..., Any], tag: str, factory_name: Optional[str] = None*) → Callable[..., Any]

flambe.compile.registrable.**make_to_yaml_with_metadata**(*to_yaml_fn: Callable[..., Any]*) → Callable[..., Any]

class flambe.compile.registrable.**registration_context**(*namespace: str*)

__enter__(*self*)

__exit__(*self*, **args: Any*)

__call__(*self*, *func: Callable[..., Any]*)

class flambe.compile.registrable.**Registrable**
Bases: abc.ABC

Subclasses automatically registered as yaml tags

Automatically registers subclasses with the yaml loader by adding a constructor and representer which can be overridden

_yaml_tags :Dict[Any, List[str]]

_yaml_tag_namespace :Dict[Type, str]

_yaml_registered_factories :Set[str]

classmethod __init_subclass__(*cls: Type[R], should_register: Optional[bool] = True, tag_override: Optional[str] = None, tag_namespace: Optional[str] = None, **kwargs: Mapping[str, Any]*)

static register_tag(*class_: RT*, *tag: str*, *tag_namespace: Optional[str] = None*)

static get_default_tag(*class_: RT*, *factory_name: Optional[str] = None*)
Retrieve default yaml tag for class *cls*

Retrieve the default tag (aka the last one, which will be the only one, or the alias if it exists) for use in yaml representation

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)
Use representer to create yaml representation of node

See Component class, and experiment/options for examples

classmethod from_yaml(*cls*, *constructor: Any*, *node: Any*, *factory_name: str*)
Use constructor to create an instance of cls

See Component class, and experiment/options for examples

flambe.compile.registrable.**alias**(*tag: str, tag_namespace: Optional[str] = None*) → Callable[[RT], RT]
Decorate a Registrable subclass with a new tag

Can be added multiple times to give a class multiple aliases, however the top most alias tag will be the default tag which means it will be used when representing the class in YAML

flambe.compile.registrable.**register**(*cls: Type[A], tag: str*) → Type[A]
Safely register a new tag for a class

Similar to alias, but it's intended to be used on classes that are not already subclasses of Registrable, and it is NOT a decorator

class flambe.compile.registrable.**registrable_factory**(*fn: Any*)
Decorate Registrable factory method for use in the config

This Descriptor class will set properties that allow the factory method to be specified directly in the config as a suffix to the tag; for example:

```
class MyModel(Component):

    @registrable_factory
    def from_file(cls, path):
```

(continues on next page)

(continued from previous page)

```
        # load instance from path
        ...
        return instance
```

defines the factory, which can then be used in yaml:

```
model: !MyModel.from_file
    path: some/path/to/file.pt
```

__set_name__(*self*, *owner: type*, *name: str*)

class flambe.compile.registrable.**MappedRegistrable**
Bases: *flambe.compile.registrable.Registrable*

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)
Use representer to create yaml representation of node

classmethod from_yaml(*cls*, *constructor: Any*, *node: Any*, *factory_name: str*)
Use constructor to create an instance of cls

20.1.6 flambe.compile.serialization

Module Contents

flambe.compile.serialization.**logger**

exception flambe.compile.serialization.**LoadError**
Bases: Exception

Error thrown because of fatal error when loading

class flambe.compile.serialization.**SaveTreeNode**
Bases: typing.NamedTuple

Tree representation corresponding to the directory save format

state :Dict[str, Any]

version :str

class_name :str

source_code :str

config :str

object_stash :Dict[str, Any]

children :Dict[str, Any]

class flambe.compile.serialization.**State**
Bases: collections.OrderedDict

A state object for Flambe.

_metadata :Dict[str, Any]

flambe.compile.serialization.**_convert_to_tree**(*metadata: Dict[str, Any]*) → SaveTreeNode

flambe.compile.serialization.**_update_save_tree**(*save_tree: SaveTreeNode*, *key: Sequence[str]*, *value: Any*) → None

flambe.compile.serialization.**_traverse_all_nodes**(*save_tree: SaveTreeNode*, *path: Optional[List[str]] = None*) → Iterable[Tuple[List[str], SaveTreeNode]]

flambe.compile.serialization.**_extract_prefix**(*root*, *directory*)

flambe.compile.serialization.**_prefix_keys**(*state*, *prefix*)

flambe.compile.serialization.**traverse**(*nested: Mapping[str, Any], path: Optional[List[str]] = None*) → Iterable[Any]

Iterate over a nested mapping returning the path and key, value.

Parameters

- **nested** (*Mapping[str, Any]*) – Mapping where some values are also mappings that should be traversed
- **path** (*List[str]*) – List of keys that were used to reach the current mapping

Returns Iterable of path, key, value triples

Return type Iterable[Any]

flambe.compile.serialization.**_update_link_refs**(*schema: Mapping*) → None

Resolve links in schemas at *block_id*.

Parameters **schema** (*Dict[str, Schema[Any]]*) – Map from *block_id* to *Schema* object

flambe.compile.serialization.**save_state_to_file**(*state: State*, *path: str*, *compress: bool = False*, *pickle_only: bool = False*, *overwrite: bool = False*, *pickle_module=dill*, *pickle_protocol=DEFAULT_PROTOCOL*) → None

Save state to given path

By default the state will be saved in directory structure that mirrors the object hierarchy, so that you can later inspect the save file and load individual components more easily. If you would like to compress this directory structure using tar + gz, set *compress* to True. You can also use pickle to write a single output file, more similar to how PyTorch's save function operates.

Parameters

- **state** (*State*) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **path** (*str*) – Location to save the file / save directory to; This should be a new non-existent path; if the path is an existing directory and it contains files an exception will be raised. This is because the path includes the final name of the save file (if using pickle or compress) or the final name of the save directory.
- **compress** (*bool*) – Whether to compress the save file / directory via tar + gz
- **pickle_only** (*bool*) – Use given pickle_module instead of the hiearchical save format (the default is False).
- **overwrite** (*bool*) – If true, overwrites the contents of the given path to create a directory at that location
- **pickle_module** (*type*) – Pickle module that has load and dump methods; dump should accpet a pickle_protocol parameter (the default is dill).
- **pickle_protocol** (*type*) – Pickle protocol to use; see pickle for more details (the default is 2).

Raises `ValueError` – If the given path exists, is a directory, and already contains some files.

`flambe.compile.serialization.`**`save`**(*obj: Any*, *path: str*, *compress: bool = False*, *pickle_only: bool = False*, *overwrite: bool = False*, *pickle_module=dill*, *pickle_protocol=DEFAULT_PROTOCOL*) → None

Save *Component* object to given path

See *save_state_to_file* for a more detailed explanation

Parameters

- **`obj`** (`Component`) – The component to save.
- **`path`** (*str*) – Location to save the file / save directory to
- **`compress`** (*bool*) – Whether to compress the save file / directory via tar + gz
- **`pickle_only`** (*bool*) – Use given pickle_module instead of the hiearchical save format (the default is False).
- **`overwrite`** (*bool*) – If true, overwrites the contents of the given path to create a directory at that location
- **`pickle_module`** (*type*) – Pickle module that has load and dump methods; dump should accept a pickle_protocol parameter (the default is dill).
- **`pickle_protocol`** (*type*) – Pickle protocol to use; see pickle for more details (the default is 2).

`flambe.compile.serialization.`**`load_state_from_file`**(*path: str*, *map_location=None*, *pickle_module=dill*, ***pickle_load_args*) → State

Load state from the given path

Loads a flambe save directory, pickled save object, or a compressed version of one of these two formats (using tar + gz). Will automatically infer the type of save format and if the directory structure is used, the serialization protocol version as well.

Parameters

- **`path`** (*str*) – Path to the save file or directory
- **`map_location`** (*type*) – Location (device) where items will be moved. ONLY used when the directory save format is used. See torch.load documentation for more details (the default is None).
- **`pickle_module`** (*type*) – Pickle module that has load and dump methods; dump should accept a pickle_protocol parameter (the default is dill).
- **`**pickle_load_args`** (*type*) – Additional args that *pickle_module* should use to load; see torch.load documentation for more details

Returns state_dict that can be loaded into a compatible Component

Return type *State*

`flambe.compile.serialization.`**`load`**(*path: str*, *map_location=None*, *auto_install=False*, *pickle_module=dill*, ***pickle_load_args*)

Load object with state from the given path

Loads a flambe object by using the saved config files, and then loads the saved state into said object. See *load_state_from_file* for details regarding how the state is loaded from the save file or directory.

Parameters

- **`path`** (*str*) – Path to the save file or directory

- **map_location** (*type*) – Location (device) where items will be moved. ONLY used when the directory save format is used. See torch.load documentation for more details (the default is None).
- **auto_install** (*bool*) – If True, automatically installs extensions as needed.
- **pickle_module** (*type*) – Pickle module that has load and dump methods; dump should accept a pickle_protocol parameter (the default is dill).
- ****pickle_load_args** (*type*) – Additional args that *pickle_module* should use to load; see torch.load documentation for more details

Returns object with both the architecture (config) and state that was saved to path

Return type *Component*

Raises *LoadError* – If a Component object is not loadable from the given path because extensions are not installed, or the config is empty, nonexistent, or otherwise invalid.

20.1.7 flambe.compile.utils

Module Contents

flambe.compile.utils.**all_subclasses**(*class_: Type[Any]*) → Set[Type[Any]]

Return a set of all subclasses for a given class object

Recursively collects all subclasses of *class_* down the object hierarchy into one set.

Parameters **class** (*Type[Any]*) – Class to retrieve all subclasses for

Returns All subclasses of **class_**

Return type Set[Type[Any]]

flambe.compile.utils.**make_component**(*class_: type*, *tag_namespace: Optional[str] = None*, *only_module: Optional[str] = None*, *parent_component_class: Optional[Type] = None*) → None

Make class and all its children a *Component*

For example a call to *make_component(torch.optim.Adam, "torch")* will make the tag *!torch.Adam* accessible in any yaml configs. This does *NOT* monkey patch (aka swizzle) torch, but instead creates a dynamic subclass which will be used by the yaml constructor i.e. only classes loaded from the config will be affected, anything imported and used in code.

Parameters

- **class** (*type*) – To be registered with yaml as a component, along with all its children
- **tag_prefix** (*str*) – Added to beginning of all the tags
- **only_module** (*str*) – Module prefix used to limit the scope of what gets registered
- **parent_component_class** (*Type*) – Parent class to use for creating a new component class; should be a subclass of :class:~flambe.compile.Component (defaults to Component)

Returns

Return type None

flambe.compile.utils.**_is_url**(*resource: str*) → bool
Whether a given resource is a remote URL.

Resolve by searching for a scheme.

Parameters **resource** (*str*) – The given resource

Returns If the resource is a remote URL.

Return type bool

20.2 Package Contents

exception flambe.compile.**RegistrationError**
Bases: Exception

Error thrown when acessing yaml tag on a non-registered class

Thrown when trying to access the default yaml tag for a class typically occurs when called on an abstract class

class flambe.compile.**Registrable**
Bases: abc.ABC

Subclasses automatically registered as yaml tags

Automatically registers subclasses with the yaml loader by adding a constructor and representer which can be overridden

_yaml_tags :Dict[Any, List[str]]

_yaml_tag_namespace :Dict[Type, str]

_yaml_registered_factories :Set[str]

classmethod __init_subclass__(*cls: Type[R], should_register: Optional[bool] = True, tag_override: Optional[str] = None, tag_namespace: Optional[str] = None, **kwargs: Mapping[str, Any]*)

static register_tag(*class_: RT*, *tag: str*, *tag_namespace: Optional[str] = None*)

static get_default_tag(*class_: RT*, *factory_name: Optional[str] = None*)
Retrieve default yaml tag for class *cls*

Retrieve the default tag (aka the last one, which will be the only one, or the alias if it exists) for use in yaml representation

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)
Use representer to create yaml representation of node

See Component class, and experiment/options for examples

classmethod from_yaml(*cls*, *constructor: Any*, *node: Any*, *factory_name: str*)
Use constructor to create an instance of cls

See Component class, and experiment/options for examples

flambe.compile.**alias**(*tag: str*, *tag_namespace: Optional[str] = None*) → Callable[[RT], RT]
Decorate a Registrable subclass with a new tag

Can be added multiple times to give a class multiple aliases, however the top most alias tag will be the default tag which means it will be used when representing the class in YAML

flambe.compile.**yaml**

flambe.compile.**register**(*cls: Type[A], tag: str*) → Type[A]
Safely register a new tag for a class

Similar to alias, but it's intended to be used on classes that are not already subclasses of Registrable, and it is NOT a decorator

class flambe.compile.**registrable_factory**(*fn: Any*)
Decorate Registrable factory method for use in the config

This Descriptor class will set properties that allow the factory method to be specified directly in the config as a suffix to the tag; for example:

```
class MyModel(Component):

    @registrable_factory
    def from_file(cls, path):
        # load instance from path
        ...
        return instance
```

defines the factory, which can then be used in yaml:

```
model: !MyModel.from_file
    path: some/path/to/file.pt
```

__set_name__(*self*, *owner: type*, *name: str*)

class flambe.compile.**registration_context**(*namespace: str*)

__enter__(*self*)

__exit__(*self*, **args: Any*)

__call__(*self, func: Callable[..., Any]*)

class flambe.compile.**MappedRegistrable**
Bases: *flambe.compile.registrable.Registrable*

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)
Use representer to create yaml representation of node

classmethod from_yaml(*cls*, *constructor: Any*, *node: Any*, *factory_name: str*)
Use constructor to create an instance of cls

class flambe.compile.**Schema**(*component_subclass: Type[C], _flambe_custom_factory_name: Optional[str] = None, **keywords: Any*)
Bases: MutableMapping[str, Any]

Holds and recursively initializes Component's with kwargs

Holds a Component subclass and keyword arguments to that class's compile method. When an instance is called it will perform the recursive compilation process, turning the nested structure of Schema's into initialized Component objects

Implements MutableMapping methods to facilitate inspection and updates to the keyword args. Implements dot-notation access to the keyword args as well.

Parameters

- **component_subclass** (*Type[Component]*) – Subclass of Component that will be compiled
- ****keywords** (*Any*) – kwargs passed into the Schema's *compile* method

Examples

Create a Schema from a Component subclass

```
>>> class Test(Component):
...     def __init__(self, x=2):
...         self.x = x
...
>>> tp = Schema(Test)
>>> t1 = tp()
>>> t2 = tp()
>>> assert t1 is t2  # the same Schema always gives you same obj
>>> tp = Schema(Test) # create a new Schema
>>> tp['x'] = 3
>>> t3 = tp()
>>> assert t1.x == 3  # dot notation works as well
```

component_subclass
Subclass of Schema that will be compiled

Type Type[*Component*]

keywords
kwargs passed into the Schema's *compile* method

Type Dict[str, Any]

__call__ (*self*, *stash: Optional[Dict[str, Any]] = None*, ***keywords: Any*)

add_extensions_metadata (*self, extensions: Dict[str, str]*)
Add extensions used when loading this schema and children

Uses `component_subclass.__module__` to filter for only the single relevant extension for this object; extensions relevant for children are saved only on those children schemas directly. Use `aggregate_extensions_metadata` to generate a dictionary of all extensions used in the object hierarchy.

aggregate_extensions_metadata (*self*)
Aggregate extensions used in object hierarchy

contains (*self*, *schema: Schema*, *original_link: Link*)

__setitem__ (*self*, *key: str*, *value: Any*)

__getitem__ (*self*, *key: str*)

__delitem__ (*self*, *key: str*)

__iter__ (*self*)

__len__ (*self*)

__getattr__ (*self*, *item: str*)

__setattr__ (*self*, *key: str*, *value: Any*)

__repr__ (*self*)
Identical to super (schema), but sorts keywords

classmethod to_yaml (*cls*, *representer: Any*, *node: Any*, *tag: str = ''*)

static serialize (*obj: Any*)
Return dictionary representation of schema

Includes yaml as a string, and extensions

Parameters **obj** (*Any*) – Should be schema or dict of schemas

Returns dictionary containing yaml and extensions dictionary

Return type Dict[str, Any]

static deserialize(*data: Dict[str, Any]*)
Construct Schema from dict returned by Schema.serialize

Parameters **data** (*Dict[str, Any]*) – dictionary returned by `Schema.serialize`

Returns Schema or dict of schemas (depending on yaml in `data`)

Return type Any

class flambe.compile.**Component**(***kwargs*)
Bases: *flambe.compile.registrable.Registrable*

Class which can be serialized to yaml and implements *compile*

IMPORTANT: ALWAYS inherit from Component BEFORE *torch.nn.Module*

Automatically registers subclasses via Registrable and facilitates immediate usage in YAML with tags. When loaded, subclasses' initialization is delayed; kwargs are wrapped in a custom schema called Schema that can be easily initialized later.

_flambe_version = 0.0.0

_config_str
Represent object's architecture as a YAML string

Includes the extensions relevant to the object as well; NOTE: currently this section may include a superset of the extensions actually needed, but this will be changed in a future release.

run(*self*)
Run a single computational step.

When used in an experiment, this computational step should be on the order of tens of seconds to about 10 minutes of work on your intended hardware; checkpoints will be performed in between calls to run, and resources or search algorithms will be updated. If you want to run everything all at once, make sure a single call to run does all the work and return False.

Returns True if should continue running later i.e. more work to do

Return type bool

metric(*self*)
Override this method to enable scheduling and searching.

Returns The metric to compare different variants of your Component

Return type float

register_attrs(*self*, **names: str*)
Set attributes that should be included in state_dict

Equivalent to overriding *obj._state* and *obj._load_state* to save and load these attributes. Recommended usage: call inside *__init__* at the end: *self.register_attrs(attr1, attr2, ...)* Should ONLY be called on existing attributes.

Parameters ***names** (*str*) – The names of the attributes to register

Raises `AttributeError` – If *self* does not have existing attribute with that name

static _state_dict_hook(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any]*)
Add metadata and recurse on Component children

This hook is used to integrate with the PyTorch *state_dict* mechanism; as either *nn.Module.state_dict* or *Component.get_state* recurse, this hook is responsible for adding Flambe specific metadata and recursing further on any Component children of *self* that are not also nn.Modules, as PyTorch will handle recursing to the latter.

Flambe specific metadata includes the class version specified in the *Component._flambe_version* class property, the name of the class, the source code, and the fact that this class is a *Component* and should correspond to a directory in our hiearchical save format

Finally, this hook calls a helper *_state* that users can implement to add custom state to a given class

Parameters

- **state_dict** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **prefix** (*str*) – The current prefix for new compound keys that reflects the location of this instance in the object hierarchy being represented
- **local_metadata** (*Dict[str, Any]*) – A subset of the metadata relevant just to this object and its children

Returns The modified state_dict

Return type type

Raises ExceptionName – Why the exception is raised.

_add_registered_attrs(*self*, *state_dict: State*, *prefix: str*)

_state(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any]*)
Add custom state to state_dict

Parameters

- **state_dict** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **prefix** (*str*) – The current prefix for new compound keys that reflects the location of this instance in the object hierarchy being represented
- **local_metadata** (*Dict[str, Any]*) – A subset of the metadata relevant just to this object and its children

Returns The modified state_dict

Return type *State*

get_state(*self*, *destination: Optional[State] = None*, *prefix: str = ''*, *keep_vars: bool = False*)
Extract PyTorch compatible state_dict

Adds Flambe specific properties to the state_dict, including special metadata (the class version, source code, and class name). By default, only includes state that PyTorch *nn.Module* includes (Parameters, Buffers, child Modules). Custom state can be added via the *_state* helper method which subclasses should override.

The metadata *_flambe_directories* indicates which objects are Components and should be a subdirectory in our hierarchical save format. This object will recurse on *Component* and *nn.Module* children, but NOT *torch.optim.Optimizer* subclasses, *torch.optim.lr_scheduler._LRScheduler* subclasses, or any other arbitrary python objects.

Parameters

- **destination** (*Optional[State]*) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **prefix** (*str*) – The current prefix for new compound keys that reflects the location of this instance in the object hierarchy being represented
- **keep_vars** (*bool*) – Whether or not to keep Variables (only used by PyTorch) (the default is False).

Returns The state_dict object

Return type *State*

Raises `ExceptionName` – Why the exception is raised.

_load_state_dict_hook(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any], strict: bool, missing_keys: List[Any], unexpected_keys: List[Any], error_msgs: List[Any]*)

Load flambe-specific state

Parameters

- **state_dict** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **prefix** (*str*) – The current prefix for new compound keys that reflects the location of this instance in the object hierarchy being represented
- **local_metadata** (*Dict[str, Any]*) – A subset of the metadata relevant just to this object and its children
- **strict** (*bool*) – Whether missing or unexpected keys should be allowed; should always be False in Flambe
- **missing_keys** (*List[Any]*) – Missing keys so far
- **unexpected_keys** (*List[Any]*) – Unexpected keys so far
- **error_msgs** (*List[Any]*) – Any error messages so far

Raises `LoadError` – If the state for some object does not have a matching major version number

_load_registered_attrs(*self*, *state_dict: State*, *prefix: str*)

_load_state(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any], strict: bool, missing_keys: List[Any], unexpected_keys: List[Any], error_msgs: List[Any]*)

Load custom state (that was included via *_state*)

Subclasses should override this function to add custom state that isn't normally included by PyTorch nn.Module

Parameters

- **state_dict** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **prefix** (*str*) – The current prefix for new compound keys that reflects the location of this instance in the object hierarchy being represented
- **local_metadata** (*Dict[str, Any]*) – A subset of the metadata relevant just to this object and its children
- **strict** (*bool*) – Whether missing or unexpected keys should be allowed; should always be False in Flambe

- **missing_keys** (*List[Any]*) – Missing keys so far
- **unexpected_keys** (*List[Any]*) – Unexpected keys so far
- **error_msgs** (*List[Any]*) – Any error messages so far

load_state(*self*, *state_dict: State*, *strict: bool = False*)
Load *state_dict* into *self*

Loads state produced by *get_state* into the current object, recursing on child *Component* and *nn.Module* objects

Parameters

- **state_dict** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **strict** (*bool*) – Whether missing or unexpected keys should be allowed; should ALWAYS be False in Flambe (the default is False).

Raises LoadError – If the state for some object does not have a matching major version number

classmethod load_from_path(*cls*, *path: str*, *map_location: Union[torch.device, str] = None*, *use_saved_config_defaults: bool = True*, ***kwargs: Any*)

save(*self*, *path: str*, ***kwargs: Any*)

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)

classmethod from_yaml(*cls: Type[C], constructor: Any, node: Any, factory_name: str*)

classmethod precompile(*cls: Type[C], **kwargs: Any*)
Change kwargs before compilation occurs.

This hook is called after links have been activated, but before calling the recursive initialization process on all other objects in kwargs. This is useful in a number of cases, for example, in Trainer, we compile several objects ahead of time and move them to the GPU before compiling the optimizer, because it needs to be initialized with the model parameters *after* they have been moved to GPU.

Parameters

- **cls** (*Type[C]*) – Class on which method is called
- ****kwargs** (*Any*) – Current kwargs that will be compiled and used to initialize an instance of cls after this hook is called

aggregate_extensions_metadata(*self*)
Aggregate extensions used in object hierarchy

TODO: remove or combine with schema implementation in refactor

classmethod compile(*cls: Type[C], _flambe_custom_factory_name: Optional[str] = None, _flambe_extensions: Optional[Dict[str, str]] = None, _flambe_stash: Optional[Dict[str, Any]] = None, **kwargs: Any*)
Create instance of cls after recursively compiling kwargs

Similar to normal initialization, but recursively initializes any arguments that should be compiled and allows overriding arbitrarily deep kwargs before initializing if needed. Also activates any Link instances passed in as kwargs, and saves the original kwargs for dumping to yaml later.

Parameters ****kwargs** (*Any*) – Keyword args that should be forwarded to the initialization function (a specified factory, or the normal *__new__* and *__init__* methods)

Returns An instance of the class *cls*

Return type C

class flambe.compile.**Link**(*schematic_path: Sequence[str], attr_path: Optional[Sequence[str]] = None, target: Optional[Schema] = None, local: bool = True*)

Bases: *flambe.compile.registrable.Registrable*

Delayed access to another object in an object hiearchy

Currently only supported in the context of Experiment but this may be updated in a future release

A Link delays the access of some property, or the calling of some method, until the Link is called. Links can be passed directly into a Component subclass *compile*, Component's method called compile will automatically record the links and call them to access their values before running *__new__* and *__init__*. The recorded links will show up in the config if *yaml.dump()* is called on your object hierarchy. This typically happens when logging individual configs during a grid search, and when serializing between multiple processes.

For example, if the schematic path is ['model', 'encoder'] and the attribute path is ['rnn', 'hidden_size'] then before the link can be compiled, the target attribute should be set to point to the model schema (this is handled automatically by Experiment) then, during compilation the child schema 'encoder' will be accessed, and finally the attribute encoder.rnn.hidden_size will be returned

Parameters

- **schematic_path** (*Sequence[str]*) – Path to the relevant schema denoted by dictionary-like bracket access e.g. ['model', 'encoder']
- **attr_path** (*Sequence[str]*) – Path to the relevant attribute on the given schema (after it's been compiled) using standard attribute dot notation e.g. ['rnn', 'hidden_size']
- **target** (*Optional[Schema]*) – The root object corresponding to the first element in the schematic path; needs to be passed in here or set later before link can be resolved
- **local** (*bool*) – if true, changes tune convert behavior to insert a dummy link; used for links to global variables ("resources" in config) (defaults to True)

root_schema :str

__repr__(*self*)

__call__(*self*)

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)

Build contextualized link based on the root node

If the link refers to something inside of the current object hierarchy (as determined by the schema of `_link_root_obj`) then it will be represented as a link; if the link refers to something out-of-scope, i.e. not inside the current object hiearchy, then replace the link with the resolved value. If the value cannot be represented, pickle it and include a reference to its id in the object stash that will be saved alongside the config

classmethod from_yaml(*cls*, *constructor: Any*, *node: Any*, *factory_name: str*)

convert(*self*)

flambe.compile.**dynamic_component**(*class_: Type[A], tag: str, tag_namespace: Optional[str] = None, parent_component_class: Type[Component] = Component*) → Type[Component]

Decorate given class, creating a dynamic *Component*

Creates a dynamic subclass of *class_* that inherits from *Component* so it will be registered with the yaml loader and receive the appropriate functionality (*from_yaml*, *to_yaml* and *compile*). *class_* should not implement any of the aforementioned functions.

Parameters

- **class** (*Type[A]*) – Class to register with yaml and the compilation system
- **tag** (*str*) – Tag that will be used with yaml
- **tag_namespace** (*str*) – Namespace aka the prefix, used. e.g. for *!torch.Adam* torch is the namespace

Returns New subclass of *_class* and *Component*

Return type Type[*Component*]

flambe.compile.**make_component**(*class_: type*, *tag_namespace: Optional[str] = None*, *only_module: Optional[str] = None*, *parent_component_class: Optional[Type] = None*) → None

Make class and all its children a *Component*

For example a call to *make_component(torch.optim.Adam, "torch")* will make the tag *!torch.Adam* accessible in any yaml configs. This does *NOT* monkey patch (aka swizzle) torch, but instead creates a dynamic subclass which will be used by the yaml constructor i.e. only classes loaded from the config will be affected, anything imported and used in code.

Parameters

- **class** (*type*) – To be registered with yaml as a component, along with all its children
- **tag_prefix** (*str*) – Added to beginning of all the tags
- **only_module** (*str*) – Module prefix used to limit the scope of what gets registered
- **parent_component_class** (*Type*) – Parent class to use for creating a new component class; should be a subclass of :class:~flambe.compile.Component (defaults to Component)

Returns

Return type None

flambe.compile.**all_subclasses**(*class_: Type[Any]*) → Set[Type[Any]]

Return a set of all subclasses for a given class object

Recursively collects all subclasses of *class_* down the object hierarchy into one set.

Parameters **class** (*Type[Any]*) – Class to retrieve all subclasses for

Returns All subclasses of **class_**

Return type Set[Type[Any]]

flambe.compile.**save**(*obj: Any*, *path: str*, *compress: bool = False*, *pickle_only: bool = False*, *overwrite: bool = False*, *pickle_module=dill*, *pickle_protocol=DEFAULT_PROTOCOL*) → None

Save *Component* object to given path

See *save_state_to_file* for a more detailed explanation

Parameters

- **obj** (Component) – The component to save.
- **path** (*str*) – Location to save the file / save directory to
- **compress** (*bool*) – Whether to compress the save file / directory via tar + gz
- **pickle_only** (*bool*) – Use given pickle_module instead of the hiearchical save format (the default is False).

- **overwrite** (*bool*) – If true, overwrites the contents of the given path to create a directory at that location
- **pickle_module** (*type*) – Pickle module that has load and dump methods; dump should accept a pickle_protocol parameter (the default is dill).
- **pickle_protocol** (*type*) – Pickle protocol to use; see pickle for more details (the default is 2).

flambe.compile.**load**(*path: str*, *map_location=None*, *auto_install=False*, *pickle_module=dill*, ***pickle_load_args*)

Load object with state from the given path

Loads a flambe object by using the saved config files, and then loads the saved state into said object. See *load_state_from_file* for details regarding how the state is loaded from the save file or directory.

Parameters

- **path** (*str*) – Path to the save file or directory
- **map_location** (*type*) – Location (device) where items will be moved. ONLY used when the directory save format is used. See torch.load documentation for more details (the default is None).
- **auto_install** (*bool*) – If True, automatically installs extensions as needed.
- **pickle_module** (*type*) – Pickle module that has load and dump methods; dump should accept a pickle_protocol parameter (the default is dill).
- ****pickle_load_args** (*type*) – Additional args that *pickle_module* should use to load; see torch.load documentation for more details

Returns object with both the architecture (config) and state that was saved to path

Return type *Component*

Raises LoadError – If a Component object is not loadable from the given path because extensions are not installed, or the config is empty, nonexistent, or otherwise invalid.

flambe.compile.**save_state_to_file**(*state: State*, *path: str*, *compress: bool = False*, *pickle_only: bool = False*, *overwrite: bool = False*, *pickle_module=dill*, *pickle_protocol=DEFAULT_PROTOCOL*) → None

Save state to given path

By default the state will be saved in directory structure that mirrors the object hierarchy, so that you can later inspect the save file and load individual components more easily. If you would like to compress this directory structure using tar + gz, set *compress* to True. You can also use pickle to write a single output file, more similar to how PyTorch's save function operates.

Parameters

- **state** (State) – The state_dict as defined by PyTorch; a flat dictionary with compound keys separated by '.'
- **path** (*str*) – Location to save the file / save directory to; This should be a new non-existent path; if the path is an existing directory and it contains files an exception will be raised. This is because the path includes the final name of the save file (if using pickle or compress) or the final name of the save directory.
- **compress** (*bool*) – Whether to compress the save file / directory via tar + gz
- **pickle_only** (*bool*) – Use given pickle_module instead of the hiearchical save format (the default is False).

- **overwrite** (*bool*) – If true, overwrites the contents of the given path to create a directory at that location
- **pickle_module** (*type*) – Pickle module that has load and dump methods; dump should accpet a pickle_protocol parameter (the default is dill).
- **pickle_protocol** (*type*) – Pickle protocol to use; see pickle for more details (the default is 2).

Raises ValueError – If the given path exists, is a directory, and already contains some files.

flambe.compile.**load_state_from_file**(*path: str*, *map_location=None*, *pickle_module=dill*, ***pickle_load_args*) → State

Load state from the given path

Loads a flambe save directory, pickled save object, or a compressed version of one of these two formats (using tar + gz). Will automatically infer the type of save format and if the directory structure is used, the serialization protocol version as well.

Parameters

- **path** (*str*) – Path to the save file or directory
- **map_location** (*type*) – Location (device) where items will be moved. ONLY used when the directory save format is used. See torch.load documentation for more details (the default is None).
- **pickle_module** (*type*) – Pickle module that has load and dump methods; dump should accept a pickle_protocol parameter (the default is dill).
- ****pickle_load_args** (*type*) – Additional args that *pickle_module* should use to load; see torch.load documentation for more details

Returns state_dict that can be loaded into a compatible Component

Return type *State*

class flambe.compile.**State**

Bases: collections.OrderedDict

A state object for Flambe.

_metadata :Dict[str, Any]

CHAPTER 21

flambe.experiment

21.1 Subpackages

21.1.1 flambe.experiment.webapp

Submodules

flambe.experiment.webapp.app

Report site using Flambe

The report site is the main control centre for the remote experiment. It's responsible of showing the current status of the experiment and to provide the resources that the experiment is producing (metrics, trained models, etc.)

The website consumes the files that flambe outputs, it doesn't require extra running resource (like for example, Redis DB).

The website is implemented using Flask.

Important: no typing is provided for now.

Module Contents

flambe.experiment.webapp.app.**app**

flambe.experiment.webapp.app.**load_state**()
Get status about the blocks (runned, running, remaining)

flambe.experiment.webapp.app.**analyze_download_params**(*block*, *variant*)

flambe.experiment.webapp.app.**download**(*block=None*, *variant=None*)
Downloads models + logs stored in the filesystem of the Orchestrator

flambe.experiment.webapp.app.**download_logs**(*block=None*, *variant=None*)
Downloads all artifacts but the models stored in the filesystem of the Orchestrator machine

flambe.experiment.webapp.app.**stream**()

flambe.experiment.webapp.app.**state**()

flambe.experiment.webapp.app.**console_log**()
View console with the logs

flambe.experiment.webapp.app.**index**()
Main endpoint. Works with the *index.html* template.

21.2 Submodules

21.2.1 flambe.experiment.experiment

Module Contents

flambe.experiment.experiment.**logger**

flambe.experiment.experiment.**OptionalSearchAlgorithms**

flambe.experiment.experiment.**OptionalTrialSchedulers**

class flambe.experiment.experiment.**Experiment**(*name: str, pipeline: Dict[str, Schema], resume: Optional[Union[str, Sequence[str]]] = None, devices: Dict[str, int] = None, save_path: Optional[str] = None, resources: Optional[Dict[str, Union[str, ClusterResource]]] = None, search: OptionalSearchAlgorithms = None, schedulers: OptionalTrialSchedulers = None, reduce: Optional[Dict[str, int]] = None, env: RemoteEnvironment = None, max_failures: int = 1, stop_on_failure: bool = True, merge_plot: bool = True, user_provider: Callable[[], str] = None*)

Bases: *flambe.runnable.ClusterRunnable*

A Experiment object.

The Experiment object is the top level module in the Flambé workflow. The object is responsible for starting workers, assiging the orchestrator machine, as well as converting the input blocks into Ray Tune Experiment objects.

Parameters

- **name** (*str*) – A name for the experiment
- **pipeline** (*OrderedDict[str, Schema[Component]]*) – Ordered mapping from block id to a schema of the block
- **force** (*bool*) – When running a local experiment this flag will make flambe override existing results from previous experiments. When running remote experiments this flag will reuse an existing cluster (in case of any) that is running an experiment with the same name in the same cloud service. The use of this flag is discouraged as you may lose useful data.

- **resume** (*Union[str, List[str]]*) – If a string is given, resume all blocks up until the given block_id. If a list is given, resume all blocks in that list.
- **save_path** (*Optional[str]*) – A directory where to save the experiment.
- **devices** (*Dict[str, int]*) – Tune's resources per trial. For example: {"cpu": 12, "gpu": 2}.
- **resources** (*Optional[Dict[str, Dict[str, Any]]]*) – Variables to use in the pipeline section with !@ notation. This section is splitted into 2 sections: local and remote.
- **search** (*Mapping[str, SearchAlgorithm], optional*) – Map from block id to hyperparameter search space generator. May have Schemas of SearchAlgorithm as well.
- **schedulers** (*Mapping[str, TrialScheduler], optional*) – Map from block id to search scheduler. May have Schemas of TrialScheduler as well.
- **reduce** (*Mapping[str, int], optional*) – Map from block to number of trials to reduce to.
- **env** (RemoteEnvironment) – Contains remote information about the cluster. This object will be received in case this Experiment is running remotely.
- **max_failures** (*int*) – Number of times to retry running the pipeline if it hits some type of failure, defaults to one.
- **merge_plot** (*bool*) – Display all tensorboard logs in the same plot (per block type). Defaults to True.
- **user_provider** (*Callable[[], str]*) – The logic for specifying the user triggering this Runnable. If not passed, by default it will pick the computer's user.

process_resources (*self, resources: Dict[str, Union[str, ClusterResource]], folder: str*)
Download resources that are not tagged with '!cluster' into a given directory.

Parameters

- **resources** (*Dict[str, Union[str, ClusterResource]]*) – The resources dict
- **folder** (*str*) – The directory where the remote resources will be downloaded.

Returns The resources dict where the remote urls that don't contain '!cluster' point now to the local path where the resource was downloaded.

Return type Dict[str, Union[str, *ClusterResource*]]

run (*self*, *force: bool = False*, *verbose: bool = False*, *debug: bool = False*, ***kwargs*)
Run an Experiment

teardown (*self*)

setup (*self, cluster: Cluster, extensions: Dict[str, str], force: bool, **kwargs*)
Prepare the cluster for the Experiment remote execution.

This involves:

1) [Optional] Kill previous flambe execution
2) [Optional] Remove existing results
3) Create supporting dirs (exp/synced_results, exp/resources)
4) Install extensions in all factories

5) Launch ray cluster
6) Send resources
7) Launch Tensorboard + Report site

Parameters

- **cluster** (*Cluster*) – The cluster where this Runnable will be running
- **extensions** (*Dict[str, str]*) – The ClusterRunnable extensions
- **force** (*bool*) – The force value provided to Flambe

parse (*self*)
Parse the experiment.

Parse the Experiment in search of errors that won't allow the experiment to run successfully. If it finds any error, then it raises an ParsingExperimentError.

Raises ParsingExperimentError – In case a parsing error is found.

get_user (*self*)
Get the user that triggered this experiment.

Returns The user as a string.

Return type str

_dump_experiment_file (*self*)
Dump the experiment YAML representation to the output folder.

21.2.2 flambe.experiment.options

Module Contents

flambe.experiment.options.**Number**

class flambe.experiment.options.**Options**
Bases: *flambe.compile.Registrable*, abc.ABC

classmethod from_sequence (*cls, options: Sequence[Any]*)
Construct an options class from a sequence of values

Parameters **options** (*Sequence[Any]*) – Discrete sequence that defines what values to search over

Returns Returns a subclass of DiscreteOptions

Return type T

convert (*self*)
Convert the options to Ray Tune representation.

Returns The Ray Tune conversion

Return type Dict

classmethod to_yaml (*cls*, *representer: Any*, *node: Any*, *tag: str*)

classmethod from_yaml (*cls*, *constructor: Any*, *node: Any*, *factory_name: str*)

class flambe.experiment.options.**GridSearchOptions**(*elements: Sequence[Any]*)
Bases: Sequence[Any], *flambe.experiment.options.Options*

Discrete set of values used for grid search

Defines a finite, discrete set of values to be substituted at the location where the set currently resides in the config

classmethod from_sequence(*cls, options: Sequence[Any]*)

convert(*self*)

__getitem__(*self, key: Any*)

__len__(*self*)

__repr__(*self*)

class flambe.experiment.options.**SampledUniformSearchOptions**(*low: Number, high: Number, k: int, decimals: int = 10*)
Bases: Sequence[Number], *flambe.experiment.options.Options*

Yields k values from the range (low, high)

Randomly yields k values from the range (low, high) to be substituted at the location where the class currently resides in the config

classmethod from_sequence(*cls, options: Sequence[Any]*)

convert(*self*)

__getitem__(*self, key: Any*)

__len__(*self*)

__repr__(*self*)

classmethod to_yaml(*cls, representer: Any, node: Any, tag: str*)

class flambe.experiment.options.**ClusterResource**(*location: str*)
Bases: *flambe.compile.Registrable*

classmethod to_yaml(*cls, representer: Any, node: Any, tag: str*)

classmethod from_yaml(*cls, constructor: Any, node: Any, factory_name: str*)

21.2.3 flambe.experiment.progress

Module Contents

class flambe.experiment.progress.**ProgressState**(*name, save_path, dependency_dag, config, factories_num=0*)

checkpoint_start(*self, block_id*)

refresh(*self*)

checkpoint_end(*self, block_id, block_success*)

finish(*self*)

_save(*self*)

toJSON(*self*)

21.2.4 flambe.experiment.tune_adapter

Module Contents

class flambe.experiment.tune_adapter.**TuneAdapter**
Bases: ray.tune.Trainable

Adapter to the tune.Trainable inteface.

_setup (*self*, *config: Dict*)
Subclasses should override this for custom initialization.

save (*self*, *checkpoint_dir: Optional[str] = None*)
Override to replace checkpoint.

_train (*self*)
Subclasses should override this to implement train().

_save (*self*, *checkpoint_dir: str*)
Subclasses should override this to implement save().

_restore (*self*, *checkpoint: str*)
Subclasses should override this to implement restore().

_stop (*self*)
Subclasses should override this for any cleanup on stop.

21.2.5 flambe.experiment.utils

Module Contents

flambe.experiment.utils.**check_links** (*blocks: Dict[str, Schema], global_vars: Optional[Dict[str, Any]] = None*) → None
Check validity of links between blocks.

Ensures dependency order, and that only Comparable blocks are being reduced through a LinkBest object.

Parameters **blocks** (*OrderedDict[str, Schema[Component]]*) – The blocks to check, in order

Raises

- LinkError – On undeclared blocks (i.e not the right config order)
- ProtocolError – Attempt to reduce a non-comparable block

flambe.experiment.utils.**check_search** (*blocks: Dict[str, Schema], search: Mapping[str, SearchAlgorithm], schedulers: Mapping[str, TrialScheduler]*)
Check validity of links between blocks.

Ensures dependency order, and that only Comparable blocks are being reduced through a LinkBest object.

Parameters

- **blocks** (*OrderedDict[str, Schema[Component]]*) – Ordered mapping from block id to a schema of the block
- **search** (*Mapping[str, SearchAlgorithm], optional*) – Map from block id to hyperparameter search space generator

- **schedulers** (*Mapping[str, TrialScheduler], optional*) – Map from block id to search scheduler

Raises

- ProtocolError – Non computable block assigned a search or scheduler.
- ProtocolError – Non comparable block assigned a non default search or scheduler

flambe.experiment.utils.**convert_tune**(*data: Any*)

Convert the options and links in the block.

Convert Option objects to tune.grid_search or tune.sample_from functions, depending on the type.

Parameters **data** (*Any*) – Input object that may contain Options objects that should be converted to a Tune-compatible representation

flambe.experiment.utils.**traverse**(*nested: Mapping[str, Any], path: Optional[List[str]] = None*) → Iterable[Any]

Iterate over a nested mapping returning the path and key, value.

Parameters

- **nested** (*Mapping[str, Any]*) – Mapping where some values are also mappings that should be traversed
- **path** (*List[str]*) – List of keys that were used to reach the current mapping

Returns Iterable of path, key, value triples

Return type Iterable[Any]

flambe.experiment.utils.**traverse_all**(*obj: Any*) → Iterable[Any]

Iterate over all nested mappings and sequences

Parameters **obj** (*Any*) –

Returns Iterable of child values to obj

Return type Iterable[Any]

flambe.experiment.utils.**traverse_spec**(*nested: Mapping[str, Any], path: Optional[List[str]] = None*) → Iterable[Any]

Iterate over a nested mapping returning the path and key, value.

Parameters

- **nested** (*Mapping[str, Any]*) – Mapping where some values are also mappings that should be traversed
- **path** (*List[str]*) – List of keys that were used to reach the current mapping

Returns Iterable of path, key, value triples

Return type Iterable[Any]

flambe.experiment.utils.**update_nested**(*nested: MutableMapping[str, Any], prefix: Iterable[str], key: str, new_value: Any*) → None

Multi-level set operation for nested mapping.

Parameters

- **nested** (*Mapping[str, Any]*) – Nested dictionary where keys are all strings
- **prefix** (*Iterable[str]*) – List of keys specifying path to value to be updated
- **key** (*str*) – Final key corresponding to value to be updated

- **new_value** (*Any*) – New value to set for *[p1]...[key]* in *nested*

flambe.experiment.utils.**get_nested**(*nested: Mapping[str, Any], prefix: Iterable[str], key: str*) → Any

Get nested value in standard Mapping.

Parameters

- **nested** (*Mapping[str, Any]*) – The mapping to index in
- **prefix** (*Iterable[str]*) – The path to the final key in the nested input
- **key** (*str*) – The key to query

Returns The value at the given path and key

Return type Any

flambe.experiment.utils.**update_schema_with_params**(*schema: Schema, params: Dict[str, Any]*) → Schema

Replace options in the schema recursivly.

Parameters

- **schema** (Schema*[Any]*) – The schema object to update
- **params** (*Dict[str, Any]*) – The corresponding nested diciontary with values

Returns The update schema (same object as the input, not a copy)

Return type *Schema*[Any]

flambe.experiment.utils.**has_schemas_or_options**(*x: Any*) → bool

Check if object contains Schemas or Options.

Recurses for Mappings and Sequences

Parameters **x** (*Any*) – Input object to check for Schemas and Options

Returns True iff contains any Options or Schemas.

Return type bool

flambe.experiment.utils.**divide_nested_grid_search_options**(*config: MutableMapping[str, Any]*) → Iterable[Mapping[str, Any]]

Divide config into a config Iterable to remove nested Options.

For every GridSearchOptions or SampledUniformSearchOptions, if any values contain more Options or Schemas, create copies with a single value selected in place of the option. Resulting configs will have no nested options.

Parameters **config** (*MutableMapping[str, Any]*) – MutableMapping (or Schema) containing Options and Schemas

Returns Each Mapping contains variants from original config without nested options

Return type Iterable[Mapping[str, Any]]

flambe.experiment.utils.**extract_dict**(*config: Mapping[str, Any]*) → Dict[str, Any]

Turn the schema into a dictionary, ignoring types.

NOTE: We recurse if any value is itself a *Schema*, a *Sequence* of *Schema's, or a 'Mapping* of 'Schema's. Other unconvential collections will not be inspected.

Parameters **schema** (Schema) – The object to be converted into a dictionary

Returns The output dictionary representation.

Return type Dict

flambe.experiment.utils.**extract_needed_blocks**(*schemas: Dict[str, Schema], block_id: str, global_vars: Optional[Dict[str, Any]] = None*) → Set[str]

Returns the set of all blocks that the input block links to.

Parameters

- **schemas** (*Dict[str, Schema[Any]]*) – Map from *block_id* to *Schema* object
- **block_id** (*str*) – The block containing links

Returns The list of ancestor block ids

Return type List[str]

flambe.experiment.utils.**update_link_refs**(*schemas: Dict[str, Schema], block_id: str, global_vars: Dict[str, Any]*) → None

Resolve links in schemas at *block_id*.

Parameters

- **schemas** (*Dict[str, Schema[Any]]*) – Map from *block_id* to *Schema* object
- **block_id** (*str*) – The block where links should be activated
- **global_vars** (*Dict[str, Any]*) – The environment links (ex: resources)

flambe.experiment.utils.**get_best_trials**(*trials: List[Trial], topk: int, metric='episode_reward_mean'*) → List[Trial]

Get the trials with the best result.

Parameters

- **trials** (*List[ray.tune.Trial]*) – The list of trials to examine
- **topk** (*int*) – The number of trials to reduce to
- **metric** (*str, optional*) – The metric used in comparaison (higher is better)

Returns The list of best trials

Return type List[ray.tune.Trial]

flambe.experiment.utils.**get_non_remote_config**(*experiment*)

Returns a copy of the original config file without the remote configuration

Parameters **experiment** (Experiment) – The experiment object

flambe.experiment.utils.**local_has_gpu**() → bool

Returns is local process has GPU

Returns

Return type bool

flambe.experiment.utils.**rel_to_abs_paths**(*d: Dict[str, str]*) → Dict[str, str]

Convert relative paths to absolute paths.

Parameters **d** (*Dict[str, str]*) – A dict from name -> path.

Returns The same dict received as parameter with relative paths replaced with absolute.

Return type Dict[str, str]

flambe.experiment.utils.**shutdown_ray_node**() → int
Call 'ray stop' locally to terminate the ray node.

flambe.experiment.utils.**shutdown_remote_ray_node**(*host: str*, *user: str*, *key: str*) → int
Execute 'ray stop' on a remote machine through ssh to terminate the ray node.

IMPORTANT: this method is intended to be run in the cluster.

Parameters

- **host** (*str*) – The Orchestrator's IP that is visible by the factories (usually the private IP)
- **user** (*str*) – The username for that machine.
- **key** (*str*) – The key that communicate with the machine.

21.2.6 flambe.experiment.wording

Module Contents

flambe.experiment.wording.**logger**

flambe.experiment.wording.**print_useful_local_info**(*full_save_path*) → None
Information to display before experiment is running.

flambe.experiment.wording.**print_useful_remote_info**(*manager*, *experiment_name*) → None
Once the local process of the remote run is over, this information is shown to the user.

flambe.experiment.wording.**print_useful_metrics_only_info**() → None
Printable warning when debug flag is active

flambe.experiment.wording.**print_extensions_cache_size_warning**(*location*, *limit*) → None
Print message when the extensions cache folder is getting big.

21.3 Package Contents

class flambe.experiment.**Experiment**(*name: str, pipeline: Dict[str, Schema], resume: Optional[Union[str, Sequence[str]]] = None, devices: Dict[str, int] = None, save_path: Optional[str] = None, resources: Optional[Dict[str, Union[str, ClusterResource]]] = None, search: OptionalSearchAlgorithms = None, schedulers: OptionalTrialSchedulers = None, reduce: Optional[Dict[str, int]] = None, env: RemoteEnvironment = None, max_failures: int = 1, stop_on_failure: bool = True, merge_plot: bool = True, user_provider: Callable[[], str] = None*)
Bases: *flambe.runnable.ClusterRunnable*

A Experiment object.

The Experiment object is the top level module in the Flambé workflow. The object is responsible for starting workers, assiging the orchestrator machine, as well as converting the input blocks into Ray Tune Experiment objects.

Parameters

- **name** (*str*) – A name for the experiment

- **pipeline** (*OrderedDict[str, Schema[Component]]*) – Ordered mapping from block id to a schema of the block
- **force** (*bool*) – When running a local experiment this flag will make flambe override existing results from previous experiments. When running remote experiments this flag will reuse an existing cluster (in case of any) that is running an experiment with the same name in the same cloud service. The use of this flag is discouraged as you may lose useful data.
- **resume** (*Union[str, List[str]]*) – If a string is given, resume all blocks up until the given block_id. If a list is given, resume all blocks in that list.
- **save_path** (*Optional[str]*) – A directory where to save the experiment.
- **devices** (*Dict[str, int]*) – Tune's resources per trial. For example: {"cpu": 12, "gpu": 2}.
- **resources** (*Optional[Dict[str, Dict[str, Any]]]*) – Variables to use in the pipeline section with !@ notation. This section is splitted into 2 sections: local and remote.
- **search** (*Mapping[str, SearchAlgorithm], optional*) – Map from block id to hyperparameter search space generator. May have Schemas of SearchAlgorithm as well.
- **schedulers** (*Mapping[str, TrialScheduler], optional*) – Map from block id to search scheduler. May have Schemas of TrialScheduler as well.
- **reduce** (*Mapping[str, int], optional*) – Map from block to number of trials to reduce to.
- **env** (RemoteEnvironment) – Contains remote information about the cluster. This object will be received in case this Experiment is running remotely.
- **max_failures** (*int*) – Number of times to retry running the pipeline if it hits some type of failure, defaults to one.
- **merge_plot** (*bool*) – Display all tensorboard logs in the same plot (per block type). Defaults to True.
- **user_provider** (*Callable[[], str]*) – The logic for specifying the user triggering this Runnable. If not passed, by default it will pick the computer's user.

process_resources (*self, resources: Dict[str, Union[str, ClusterResource]], folder: str*)
Download resources that are not tagged with '!cluster' into a given directory.

Parameters

- **resources** (*Dict[str, Union[str, ClusterResource]]*) – The resources dict
- **folder** (*str*) – The directory where the remote resources will be downloaded.

Returns The resources dict where the remote urls that don't contain '!cluster' point now to the local path where the resource was downloaded.

Return type Dict[str, Union[str, *ClusterResource*]]

run (*self, force: bool = False, verbose: bool = False, debug: bool = False, **kwargs*)
Run an Experiment

teardown (*self*)

setup (*self, cluster: Cluster, extensions: Dict[str, str], force: bool, **kwargs*)
Prepare the cluster for the Experiment remote execution.

This involves:

1) [Optional] Kill previous flambe execution
2) [Optional] Remove existing results
3) Create supporting dirs (exp/synced_results, exp/resources)
4) Install extensions in all factories
5) Launch ray cluster
6) Send resources
7) Launch Tensorboard + Report site

Parameters

- **cluster** (Cluster) – The cluster where this Runnable will be running
- **extensions** (*Dict[str, str]*) – The ClusterRunnable extensions
- **force** (*bool*) – The force value provided to Flambe

parse(*self*)
Parse the experiment.

Parse the Experiment in search of errors that won't allow the experiment to run successfully. If it finds any error, then it raises an ParsingExperimentError.

Raises ParsingExperimentError – In case a parsing error is found.

get_user(*self*)
Get the user that triggered this experiment.

Returns The user as a string.

Return type str

_dump_experiment_file(*self*)
Dump the experiment YAML representation to the output folder.

class flambe.experiment.**ProgressState**(*name*, *save_path*, *dependency_dag*, *config*, *factories_num=0*)

checkpoint_start(*self*, *block_id*)

refresh(*self*)

checkpoint_end(*self*, *block_id*, *block_success*)

finish(*self*)

_save(*self*)

toJSON(*self*)

class flambe.experiment.**TuneAdapter**
Bases: ray.tune.Trainable

Adapter to the tune.Trainable inteface.

_setup(*self*, *config: Dict*)
Subclasses should override this for custom initialization.

save(*self*, *checkpoint_dir: Optional[str] = None*)
Override to replace checkpoint.

_train (*self*)
Subclasses should override this to implement train().

_save (*self*, *checkpoint_dir: str*)
Subclasses should override this to implement save().

_restore (*self*, *checkpoint: str*)
Subclasses should override this to implement restore().

_stop (*self*)
Subclasses should override this for any cleanup on stop.

class flambe.experiment.**GridSearchOptions** (*elements: Sequence[Any]*)
Bases: `Sequence[Any]`, *flambe.experiment.options.Options*

Discrete set of values used for grid search

Defines a finite, discrete set of values to be substituted at the location where the set currently resides in the config

classmethod from_sequence (*cls, options: Sequence[Any]*)

convert (*self*)

__getitem__ (*self*, *key: Any*)

__len__ (*self*)

__repr__ (*self*)

class flambe.experiment.**SampledUniformSearchOptions** (*low: Number*, *high: Number*, *k: int*, *decimals: int = 10*)
Bases: `Sequence[Number]`, *flambe.experiment.options.Options*

Yields k values from the range (low, high)

Randomly yields k values from the range (low, high) to be substituted at the location where the class currently resides in the config

classmethod from_sequence (*cls, options: Sequence[Any]*)

convert (*self*)

__getitem__ (*self*, *key: Any*)

__len__ (*self*)

__repr__ (*self*)

classmethod to_yaml (*cls*, *representer: Any*, *node: Any*, *tag: str*)

CHAPTER 22

flambe.export

22.1 Submodules

22.1.1 flambe.export.builder

Module Contents

flambe.export.builder.**logger**

class flambe.export.builder.**Builder**(*component: Schema*, *destination: str*, *storage: str = 'local'*, *compress: bool = False*, *pickle_only: bool = False*, *pickle_module=dill*, *pickle_protocol=DEFAULT_PROTOCOL*)

Bases: *flambe.runnable.Runnable*

Implement a Builder.

A builder is a simple object that can be used to create any Component post-experiment, and export it to a local or remote location.

Currently supports local, and S3 locations.

config
The secrets that the user provides. For example, 'config["AWS"]["ACCESS_KEY"]'

Type configparser.ConfigParser

run(*self*, *force: bool = False*, ***kwargs*)
Run the Builder.

save_local(*self*, *force*)
Save an object locally.

Parameters **force** (*bool*) – Wheter to use a non-empty folder or not

get_boto_session(*self*)
Get a boto Session

save_s3 (*self*, *force*)
Save an object to s3 using awscli

Parameters **force** (*bool*) – Wheter to use a non-empty bucket folder or not

22.1.2 flambe.export.exporter

Module Contents

class flambe.export.exporter.**Exporter**(***kwargs: Dict[str, Any]*)
Bases: flambe.Component

Implement an Exporter computable.

This object can be viewed as a dummy computable. It is useful to group objects into a block when those get save, to more easily refer to them later on, for instance in an object builder.

run (*self*)
Run the exporter.

Returns False, as this is a single step Component.

Return type bool

22.2 Package Contents

class flambe.export.**Builder**(*component: Schema*, *destination: str*, *storage: str = 'local'*, *compress: bool = False*, *pickle_only: bool = False*, *pickle_module=dill*, *pickle_protocol=DEFAULT_PROTOCOL*)
Bases: *flambe.runnable.Runnable*

Implement a Builder.

A builder is a simple object that can be used to create any Component post-experiment, and export it to a local or remote location.

Currently supports local, and S3 locations.

config
The secrets that the user provides. For example, 'config["AWS"]["ACCESS_KEY"]'

Type configparser.ConfigParser

run (*self*, *force: bool = False*, ***kwargs*)
Run the Builder.

save_local (*self*, *force*)
Save an object locally.

Parameters **force** (*bool*) – Wheter to use a non-empty folder or not

get_boto_session (*self*)
Get a boto Session

save_s3 (*self*, *force*)
Save an object to s3 using awscli

Parameters **force** (*bool*) – Wheter to use a non-empty bucket folder or not

class flambe.export.**Exporter**(***kwargs: Dict[str, Any]*)

Bases: flambe.Component

Implement an Exporter computable.

This object can be viewed as a dummy computable. It is useful to group objects into a block when those get save, to more easily refer to them later on, for instance in an object builder.

run(*self*)

Run the exporter.

Returns False, as this is a single step Component.

Return type bool

CHAPTER 23

flambe.field

23.1 Submodules

23.1.1 flambe.field.bow

Module Contents

class flambe.field.bow.**BoWField**(*tokenizer: Optional[Tokenizer] = None*, *lower: bool = False*, *unk_token: str = '<unk>'*, *min_freq: int = 5*, *normalize: bool = False*, *scale_factor: float = None*)

Bases: *flambe.field.Field*

Featurize raw text inputs using bag of words (BoW)

This class performs tokenization and numericalization.

The pad, unk, when given, are assigned the first indices in the vocabulary, in that order. This means, that whenever a pad token is specified, it will always use the 0 index.

Examples

```
>>> f = BoWField(min_freq=2, normalize=True)
>>> f.setup(['thank you', 'thank you very much', 'thanks a lot'])
>>> f._vocab.keys()
['thank', you']
```

Note that 'thank' and 'you' are the only ones that appear twice.

```
>>> f.process("thank you really. You help was awesome")
tensor([1, 2])
```

vocab_size :int
Get the vocabulary length.

Returns The length of the vocabulary

Return type int

process(*self*, *example*)

setup(*self*, **data*)

23.1.2 flambe.field.field

Module Contents

class flambe.field.field.**Field**
Bases: flambe.Component

Base Field interface.

A field processes raw examples and produces Tensors.

setup(*self*, **data: np.ndarray*)
Setup the field.

This method will be called with all the data in the dataset and it can be used to compute aggregated information (for example, vocabulary in Fields that process text).

ATTENTION: this method could be called multiple times in case the same field is used in different datasets. Take this into account and build a stateful implementation.

Parameters ***data** (*np.ndarray*) – Multiple 2d arrays (ex: train_data, dev_data, test_data). First dimension is for the examples, second dimension for the columns specified for this specific field.

process(*self*, **example: Any*)
Process an example into a Tensor or tuple of Tensor.

This method allows N to M mappings from example columns (N) to tensors (M).

Parameters ***example** (*Any*) – Column values of the example

Returns The processed example, as a tensor or tuple of tensors

Return type Union[torch.Tensor, Tuple[torch.Tensor, ..]]

23.1.3 flambe.field.label

Module Contents

class flambe.field.label.**LabelField**(*one_hot: bool = False*, *multilabel_sep: Optional[str] = None*, *labels: Optional[Sequence[str]] = None*)
Bases: *flambe.field.field.Field*

Featurizes input labels.

The class also handles multilabel inputs and one hot encoding.

vocab_size :int
Get the vocabulary length.

Returns The length of the vocabulary

Return type int

label_count :torch.Tensor
Get the label count.

Returns Tensor containing the count for each label, indexed by the id of the label in the vocabulary.

Return type torch.Tensor

label_freq :torch.Tensor
Get the frequency of each label.

Returns Tensor containing the frequency of each label, indexed by the id of the label in the vocabulary.

Return type torch.Tensor

label_inv_freq :torch.Tensor
Get the inverse frequency for each label.

Returns Tensor containing the inverse frequency of each label, indexed by the id of the label in the vocabulary.

Return type torch.Tensor

setup(*self*, **data: np.ndarray*)
Build the vocabulary.

Parameters **data** (*Iterable[str]*) – List of input strings.

process(*self*, *example*)
Featurize a single example.

Parameters **example** (*str*) – The input label

Returns A list of integer tokens

Return type torch.Tensor

23.1.4 flambe.field.text

Module Contents

class flambe.field.text.**TextField**(*tokenizer: Optional[Tokenizer] = None*, *lower: bool = False*, *pad_token: Optional[str] = '<pad>'*, *unk_token: Optional[str] = '<unk>'*, *sos_token: Optional[str] = None*, *eos_token: Optional[str] = None*, *embeddings: Optional[str] = None*, *embeddings_format: str = 'glove'*, *embeddings_binary: bool = False*, *unk_init_all: bool = False*, *drop_unknown: bool = False*)
Bases: *flambe.field.Field*

Featurize raw text inputs

This class performs tokenization and numericalization, as well as decorating the input sequences with optional start and end tokens.

When a vocabulary is passed during initialiazation, it is used to map the the words to indices. However, the vocabulary can also be generated from input data, through the *setup* method. Once a vocabulary has been built, this object can also be used to load external pretrained embeddings.

The pad, unk, sos and eos tokens, when given, are assigned the first indices in the vocabulary, in that order. This means, that whenever a pad token is specified, it will always use the 0 index.

vocab_size :int
Get the vocabulary length.

Returns The length of the vocabulary

Return type int

setup(*self*, **data: np.ndarray*)
Build the vocabulary and sets embeddings.

Parameters **data** (*Iterable[str]*) – List of input strings.

process(*self*, *example: str*)
Process an example, and create a Tensor.

Parameters **example** (*str*) – The example to process, as a single string

Returns The processed example, tokenized and numericalized

Return type torch.Tensor

23.2 Package Contents

class flambe.field.**Field**
Bases: flambe.Component

Base Field interface.

A field processes raw examples and produces Tensors.

setup(*self*, **data: np.ndarray*)
Setup the field.

This method will be called with all the data in the dataset and it can be used to compute aggregated information (for example, vocabulary in Fields that process text).

ATTENTION: this method could be called multiple times in case the same field is used in different datasets. Take this into account and build a stateful implementation.

Parameters ***data** (*np.ndarray*) – Multiple 2d arrays (ex: train_data, dev_data, test_data). First dimension is for the examples, second dimension for the columns specified for this specific field.

process(*self*, **example: Any*)
Process an example into a Tensor or tuple of Tensor.

This method allows N to M mappings from example columns (N) to tensors (M).

Parameters ***example** (*Any*) – Column values of the example

Returns The processed example, as a tensor or tuple of tensors

Return type Union[torch.Tensor, Tuple[torch.Tensor, ..]]

class flambe.field.**TextField**(*tokenizer: Optional[Tokenizer] = None*, *lower: bool = False*, *pad_token: Optional[str] = '<pad>'*, *unk_token: Optional[str] = '<unk>'*, *sos_token: Optional[str] = None*, *eos_token: Optional[str] = None*, *embeddings: Optional[str] = None*, *embeddings_format: str = 'glove'*, *embeddings_binary: bool = False*, *unk_init_all: bool = False*, *drop_unknown: bool = False*)
Bases: *flambe.field.Field*

Featurize raw text inputs

This class performs tokenization and numericalization, as well as decorating the input sequences with optional start and end tokens.

When a vocabulary is passed during initialiazation, it is used to map the the words to indices. However, the vocabulary can also be generated from input data, through the *setup* method. Once a vocabulary has been built, this object can also be used to load external pretrained embeddings.

The pad, unk, sos and eos tokens, when given, are assigned the first indices in the vocabulary, in that order. This means, that whenever a pad token is specified, it will always use the 0 index.

vocab_size :int
Get the vocabulary length.

Returns The length of the vocabulary

Return type int

setup (*self*, **data: np.ndarray*)
Build the vocabulary and sets embeddings.

Parameters **data** (*Iterable[str]*) – List of input strings.

process (*self*, *example: str*)
Process an example, and create a Tensor.

Parameters **example** (*str*) – The example to process, as a single string

Returns The processed example, tokenized and numericalized

Return type torch.Tensor

class flambe.field.**BoWField**(*tokenizer: Optional[Tokenizer] = None*, *lower: bool = False*, *unk_token: str = '<unk>'*, *min_freq: int = 5*, *normalize: bool = False*, *scale_factor: float = None*)
Bases: *flambe.field.Field*

Featurize raw text inputs using bag of words (BoW)

This class performs tokenization and numericalization.

The pad, unk, when given, are assigned the first indices in the vocabulary, in that order. This means, that whenever a pad token is specified, it will always use the 0 index.

Examples

```
>>> f = BoWField(min_freq=2, normalize=True)
>>> f.setup(['thank you', 'thank you very much', 'thanks a lot'])
>>> f._vocab.keys()
['thank', you']
```

Note that 'thank' and 'you' are the only ones that appear twice.

```
>>> f.process("thank you really. You help was awesome")
tensor([1, 2])
```

vocab_size :int
Get the vocabulary length.

Returns The length of the vocabulary

Return type int

process (*self*, *example*)

setup(*self*, **data*)

class flambe.field.**LabelField**(*one_hot: bool = False*, *multilabel_sep: Optional[str] = None*, *labels: Optional[Sequence[str]] = None*)

Bases: *flambe.field.field.Field*

Featurizes input labels.

The class also handles multilabel inputs and one hot encoding.

vocab_size :int
Get the vocabulary length.

Returns The length of the vocabulary

Return type int

label_count :torch.Tensor
Get the label count.

Returns Tensor containing the count for each label, indexed by the id of the label in the vocabulary.

Return type torch.Tensor

label_freq :torch.Tensor
Get the frequency of each label.

Returns Tensor containing the frequency of each label, indexed by the id of the label in the vocabulary.

Return type torch.Tensor

label_inv_freq :torch.Tensor
Get the inverse frequency for each label.

Returns Tensor containing the inverse frequency of each label, indexed by the id of the label in the vocabulary.

Return type torch.Tensor

setup(*self*, **data: np.ndarray*)
Build the vocabulary.

Parameters **data** (*Iterable[str]*) – List of input strings.

process(*self*, *example*)
Featurize a single example.

Parameters **example** (*str*) – The input label

Returns A list of integer tokens

Return type torch.Tensor

CHAPTER 24

flambe.learn

24.1 Submodules

24.1.1 flambe.learn.distillation

Module Contents

class flambe.learn.distillation.**DistillationTrainer**(*dataset: Dataset*, *train_sampler: Sampler*, *val_sampler: Sampler*, *teacher_model: Module*, *student_model: Module*, *loss_fn: Metric*, *metric_fn: Metric*, *optimizer: Optimizer*, *scheduler: Optional[_LRScheduler] = None*, *iter_scheduler: Optional[_LRScheduler] = None*, *device: Optional[str] = None*, *max_steps: int = 10*, *epoch_per_step: float = 1.0*, *iter_per_step: Optional[int] = None*, *batches_per_iter: int = 1*, *lower_is_better: bool = False*, *max_grad_norm: Optional[float] = None*, *max_grad_abs_val: Optional[float] = None*, *extra_validation_metrics: Optional[List[Metric]] = None*, *teacher_columns: Optional[Tuple[int, ...]] = None*, *student_columns: Optional[Tuple[int, ...]] = None*, *alpha_kl: float = 0.5*, *temperature: int = 1*, *unlabel_dataset: Optional[Dataset] = None*, *unlabel_sampler: Optional[Sampler] = None*)

Bases: *flambe.learn.Trainer*

Implement a Distillation Trainer.

Perform knowledge distillation between a teacher and a student model. Note that the model outputs are expected to be raw logits. Make sure that you are not applying a softmax after the decoder. You can replace the traditional Decoder with a MLPEncoder.

_compute_loss(*self, batch: Tuple[torch.Tensor, ...]*)

Compute the loss for a single batch

Important: the student and teacher output predictions must be the raw logits, so ensure that your decoder object is step with *take_log=False*.

Parameters **batch** (*Tuple[torch.Tensor, ...]*) – The batch to train on

Returns The computed loss

Return type torch.Tensor

_aggregate_preds(*self*, *data_iterator*)

Aggregate the predicitons and targets for the dataset.

Parameters **data_iterator** (*Iterator*) – Batches of data

Returns The predictions, and targets

Return type Tuple[torch.tensor, torch.tensor]

24.1.2 flambe.learn.eval

Module Contents

class flambe.learn.eval.**Evaluator**(*dataset: Dataset*, *model: Module*, *metric_fn: Metric*, *eval_sampler: Optional[Sampler] = None*, *eval_data: str = 'test'*, *device: Optional[str] = None*)

Bases: *flambe.compile.Component*

Implement an Evaluator block.

An *Evaluator* takes as input data, and a model and executes the evaluation. This is a single step *Component* object.

run(*self*, *block_name: str = None*)

Run the evaluation.

Returns Whether the component should continue running.

Return type bool

metric(*self*)

Override this method to enable scheduling.

Returns The metric to compare computable varients

Return type float

24.1.3 flambe.learn.script

Module Contents

class flambe.learn.script.**Script**(*script: str, args: Dict[str, Any], output_dir_arg: Optional[str] = None*)

Bases: *flambe.compile.Component*

Implement a Script computable.

The obejct can be used to turn any script into a Flambé computable. This is useful when you want to rapidly integrate code. Note however that this computable does not enable checkpointing or linking to internal components as it does not have any attributes.

To use this object, your script needs to be in a pip installable, containing all dependencies. The script is run with the following command:

```
python -m script.py --arg1 value1 --arg2 value2
```

run(*self*)

Run the evaluation.

Returns Report dictionary to use for logging

Return type Dict[str, float]

24.1.4 flambe.learn.train

Module Contents

class flambe.learn.train.**Trainer**(*dataset: Dataset*, *train_sampler: Sampler*, *val_sampler: Sampler*, *model: Module*, *loss_fn: Metric*, *metric_fn: Metric*, *optimizer: Optimizer*, *scheduler: Optional[_LRScheduler] = None*, *iter_scheduler: Optional[_LRScheduler] = None*, *device: Optional[str] = None*, *max_steps: int = 10*, *epoch_per_step: float = 1.0*, *iter_per_step: Optional[int] = None*, *batches_per_iter: int = 1*, *lower_is_better: bool = False*, *max_grad_norm: Optional[float] = None*, *max_grad_abs_val: Optional[float] = None*, *extra_validation_metrics: Optional[List[Metric]] = None*)

Bases: *flambe.compile.Component*

Implement a Trainer block.

A *Trainer* takes as input data, model and optimizer, and executes training incrementally in *run*.

Note that it is important that a trainer run be long enough to not increase overhead, so at least a few seconds, and ideally multiple minutes.

_create_train_iterator(*self*)

_batch_to_device(*self, batch: Tuple[torch.Tensor, ...]*)
Move the current batch on the correct device.

Can be overriden if a batch doesn't follow the expected structure. For example if the batch is a dictionary.

Parameters **batch** (*Tuple[torch.Tensor, ..]*) – The batch to train on.

_compute_loss(*self, batch: Tuple[torch.Tensor, ...]*)
Compute the loss given a single batch

Parameters **batch** (*Tuple[torch.Tensor, ..]*) – The batch to train on.

_train_step(*self*)
Run a training step over the training data.

_aggregate_preds(*self*, *data_iterator: Iterator*)
Aggregate the predicitons and targets for the dataset.

Parameters **data_iterator** (*Iterator*) – Batches of data.

Returns The predictions and targets.

Return type Tuple[torch.tensor, torch.tensor]

_eval_step(*self*)
Run an evaluation step over the validation data.

run(*self*)
Evaluate and then train until the next checkpoint

Returns Whether the component should continue running.

Return type bool

metric(*self*)
Override this method to enable scheduling.

Returns The metric to compare computable variants.

Return type float

_state(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any]*)

_load_state(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any], strict: bool, missing_keys: List[Any], unexpected_keys: List[Any], error_msgs: List[Any]*)

classmethod precompile(*cls*, ***kwargs*)

Override initialization.

Ensure that the model is compiled and pushed to the right device before its parameters are passed to the optimizer.

24.1.5 flambe.learn.utils

Module Contents

flambe.learn.utils.**select_device**(*device: Optional[str]*) → str

Chooses the torch device to run in.

Parameters

device: Union[torch.device, str] A device or a string representing a device, such as 'cpu'

str the passed-as-parameter device if any, otherwise cuda if available. Last option is cpu.

24.2 Package Contents

class flambe.learn.**Trainer**(*dataset: Dataset, train_sampler: Sampler, val_sampler: Sampler, model: Module, loss_fn: Metric, metric_fn: Metric, optimizer: Optimizer, scheduler: Optional[_LRScheduler] = None, iter_scheduler: Optional[_LRScheduler] = None, device: Optional[str] = None, max_steps: int = 10, epoch_per_step: float = 1.0, iter_per_step: Optional[int] = None, batches_per_iter: int = 1, lower_is_better: bool = False, max_grad_norm: Optional[float] = None, max_grad_abs_val: Optional[float] = None, extra_validation_metrics: Optional[List[Metric]] = None*)

Bases: *flambe.compile.Component*

Implement a Trainer block.

A *Trainer* takes as input data, model and optimizer, and executes training incrementally in *run*.

Note that it is important that a trainer run be long enough to not increase overhead, so at least a few seconds, and ideally multiple minutes.

_create_train_iterator(*self*)

_batch_to_device(*self, batch: Tuple[torch.Tensor, ...]*)

Move the current batch on the correct device.

Can be overriden if a batch doesn't follow the expected structure. For example if the batch is a dictionary.

Parameters **batch** (*Tuple[torch.Tensor, ...]*) – The batch to train on.

_compute_loss(*self, batch: Tuple[torch.Tensor, ...]*)

Compute the loss given a single batch

Parameters **batch** (*Tuple[torch.Tensor, ...]*) – The batch to train on.

_train_step(*self*)

Run a training step over the training data.

_aggregate_preds(*self*, *data_iterator: Iterator*)
Aggregate the predicitons and targets for the dataset.

Parameters **data_iterator** (*Iterator*) – Batches of data.

Returns The predictions and targets.

Return type Tuple[torch.tensor, torch.tensor]

_eval_step(*self*)
Run an evaluation step over the validation data.

run(*self*)
Evaluate and then train until the next checkpoint

Returns Whether the component should continue running.

Return type bool

metric(*self*)
Override this method to enable scheduling.

Returns The metric to compare computable variants.

Return type float

_state(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any]*)

_load_state(*self, state_dict: State, prefix: str, local_metadata: Dict[str, Any], strict: bool, missing_keys: List[Any], unexpected_keys: List[Any], error_msgs: List[Any]*)

classmethod precompile(*cls*, ***kwargs*)
Override initialization.

Ensure that the model is compiled and pushed to the right device before its parameters are passed to the optimizer.

class flambe.learn.**Evaluator**(*dataset: Dataset*, *model: Module*, *metric_fn: Metric*, *eval_sampler: Optional[Sampler] = None*, *eval_data: str = 'test'*, *device: Optional[str] = None*)
Bases: *flambe.compile.Component*

Implement an Evaluator block.

An *Evaluator* takes as input data, and a model and executes the evaluation. This is a single step *Component* object.

run(*self*, *block_name: str = None*)
Run the evaluation.

Returns Whether the component should continue running.

Return type bool

metric(*self*)
Override this method to enable scheduling.

Returns The metric to compare computable varients

Return type float

class flambe.learn.**Script**(*script: str, args: Dict[str, Any], output_dir_arg: Optional[str] = None*)
Bases: *flambe.compile.Component*

Implement a Script computable.

The obejct can be used to turn any script into a Flambé computable. This is useful when you want to rapidly integrate code. Note however that this computable does not enable checkpointing or linking to internal components as it does not have any attributes.

To use this object, your script needs to be in a pip installable, containing all dependencies. The script is run with the following command:

```
python -m script.py --arg1 value1 --arg2 value2
```

run (*self*)
Run the evaluation.

Returns Report dictionary to use for logging

Return type Dict[str, float]

class flambe.learn.**DistillationTrainer**(*dataset: Dataset*, *train_sampler: Sampler*, *val_sampler: Sampler*, *teacher_model: Module*, *student_model: Module*, *loss_fn: Metric*, *metric_fn: Metric*, *optimizer: Optimizer*, *scheduler: Optional[_LRScheduler] = None*, *iter_scheduler: Optional[_LRScheduler] = None*, *device: Optional[str] = None*, *max_steps: int = 10*, *epoch_per_step: float = 1.0*, *iter_per_step: Optional[int] = None*, *batches_per_iter: int = 1*, *lower_is_better: bool = False*, *max_grad_norm: Optional[float] = None*, *max_grad_abs_val: Optional[float] = None*, *extra_validation_metrics: Optional[List[Metric]] = None*, *teacher_columns: Optional[Tuple[int, ...]] = None*, *student_columns: Optional[Tuple[int, ...]] = None*, *alpha_kl: float = 0.5*, *temperature: int = 1*, *unlabel_dataset: Optional[Dataset] = None*, *unlabel_sampler: Optional[Sampler] = None*)

Bases: *flambe.learn.Trainer*

Implement a Distillation Trainer.

Perform knowledge distillation between a teacher and a student model. Note that the model outputs are expected to be raw logits. Make sure that you are not applying a softmax after the decoder. You can replace the traditional Decoder with a MLPEncoder.

_compute_loss (*self, batch: Tuple[torch.Tensor, ...]*)
Compute the loss for a single batch

Important: the student and teacher output predictions must be the raw logits, so ensure that your decoder object is step with *take_log=False*.

Parameters **batch** (*Tuple[torch.Tensor, ...]*) – The batch to train on

Returns The computed loss

Return type torch.Tensor

_aggregate_preds (*self*, *data_iterator*)
Aggregate the predicitons and targets for the dataset.

Parameters **data_iterator** (*Iterator*) – Batches of data

Returns The predictions, and targets

Return type Tuple[torch.tensor, torch.tensor]

CHAPTER 25

flambe.logging

25.1 Subpackages

25.1.1 flambe.logging.handler

Submodules

flambe.logging.handler.contextual_file

Module Contents

class flambe.logging.handler.contextual_file.**ContextualFileHandler**(*canonical_name: str*, *mode: str = 'a'*, *encoding=None*)

Bases: logging.FileHandler

Uses the record *current_log_dir* value to customize file path

Uses the LogRecord object's current_log_dir value to dynamically determine a path for the output file name. Functions the same as parent *logging.FileHandler* but always writes to the file given by *current_log_dir* + *canonical_name*.

Parameters

- **canonical_name** (*str*) – Common name for each file
- **mode** (*str*) – See built-in *open* description of *mode*
- **encoding** (*type*) – See built-in *open* description of *encoding*

current_log_dir
Most recently used prefix in an incoming *LogRecord*

Type str

canonical_name
Common name for each file

Type str

mode
See built-in *open* description of *mode*

Type str

delay
If true will delay opening of file until first use

Type type

stream
Currently open file stream for writing logs - should match the file indicated by *base_path* + *current_prefix* + *canonical_name*

Type type

encoding
See built-in *open* description of *encoding*

baseFilename :str
Output filename; Override parent property to use prefix

emit(*self*, *record: logging.LogRecord*)
Emit a record

If the stream is invalidated by a new record *prefix* value it will be closed and set to *None* before calling the super *emit* which will handle opening a new stream to *baseFilename*

Parameters **record** (*logging.LogRecord*) – Record to be saved at *_console_log_dir*

Returns

Return type None

flambe.logging.handler.tensorboard

Module Contents

class flambe.logging.handler.**TensorboardXHandler**(**args: Any*, ***kwargs: Any*)

Bases: logging.Handler

Implements Tensorboard message logging via TensorboardX

Parameters

- **writer** (*SummaryWriter*) – Initialized TensorboardX Writer
- ***args** (*Any*) – Other positional args for *logging.Handler*
- ****kwargs** (*Any*) – Other kwargs for *logging.Handler*

writer
Initialized TensorboardX Writer

Type SummaryWriter

emit (*self*, *record: logging.LogRecord*)
Save to tensorboard logging directory

Overrides *logging.Handler.emit*

Parameters **record** (*logging.LogRecord*) – LogRecord with data relevant to Tensorboard

Returns

Return type None

close (*self*)
Teardown writers and teardown super

Returns

Return type None

flush (*self*)
Call flush on the Tensorboard writer

Returns

Return type None

25.2 Submodules

25.2.1 flambe.logging.datatypes

Module Contents

class flambe.logging.datatypes.**ScalarT**
Bases: typing.NamedTuple

A single scalar value

Supported by TensorboardX

Parameters

- **tag** (*str*) – Data identifier
- **scalar_value** (*float*) – The scalar value
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float = time.time()*) – Wall clock time associated with this value

tag :str

scalar_value :float

global_step :int

walltime :float

__repr__ (*self*)

class flambe.logging.datatypes.**ScalarsT**
Bases: typing.NamedTuple

A dictionary mapping tag keys to scalar values

Supported by TensorboardX

Parameters

- **main_tag** (*str*) – Parent name for all the children tags
- **tag_scalar_dict** (*Dict[str, float]*) – Mapping from scalar tags to their values
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float = time.time()*) – Wall clock time associated with this value

main_tag :str

tag_scalar_dict :Dict[str, float]

global_step :int

walltime :float

__repr__(*self*)

class flambe.logging.datatypes.**HistogramT**
Bases: typing.NamedTuple

A histogram with an array of values

Supported by TensorboardX

Parameters

- **tag** (*str*) – Data identifier
- **values** (*Union[torch.Tensor, numpy.array]*) – Values to build histogram
- **global_step** (*int*) – Iteration associated with this value
- **bins** (*str*) – Determines how bins are made
- **walltime** (*float = time.time()*) – Wall clock time associated with this value

tag :str

values :Union[torch.Tensor, numpy.array]

global_step :int

bins :str

walltime :float

__repr__(*self*)

class flambe.logging.datatypes.**ImageT**
Bases: typing.NamedTuple

Image message

Supported by TensorboardX

Parameters

- **tag** (*str*) – Data identifier
- **img_tensor** (*Union*) – Image tensor to record
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float*) – Wall clock time associated with this value

tag :str

img_tensor :Union[torch.Tensor, numpy.array]

global_step :int

walltime :float

__repr__(*self*)

class flambe.logging.datatypes.**TextT**
Bases: typing.NamedTuple

Text message

Supported by TensorboardX

Parameters

- **tag** (*str*) – Data identifier
- **text_string** (*str*) – String to record
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float*) – Wall clock time associated with this value

tag :str

text_string :str

global_step :int

walltime :float

__repr__(*self*)

class flambe.logging.datatypes.**PRCurveT**
Bases: typing.NamedTuple

PRCurve message

Supported by TensorboardX

Parameters

- **tag** (*str*) – Data identifier
- **labels** (*Union[torch.Tensor, numpy.array]*) – Containing 0, 1 values
- **predictions** (*Union[torch.Tensor, numpy.array]*) – Containing 0<=x<=1 values. Needs to match labels size
- **num_thresholds** (*int = 127*) – The number of thresholds to evaluate. Max value allowed 127.
- **weights** (*Optional[float] = None*) – No description provided.
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float*) – Wall clock time associated with this value

tag :str

labels :Union[torch.Tensor, numpy.array]

predictions :Union[torch.Tensor, numpy.array]

global_step :int

num_thresholds :int = 127

weights :Optional[float]

walltime :float

__repr__(*self*)

class flambe.logging.datatypes.**EmbeddingT**
Bases: typing.NamedTuple

Embedding data, including array of vaues and metadata

Supported by TensorboardX

Parameters

- **mat** ((N, D)) – A matrix where each row is the feature vector of a data point
- **metadata** (*Sequence[str]*) – A list of labels; each element will be converted to string
- **label_img** ((N, C, H, W)) – Images corresponding to each data point
- **global_step** (*int*) – Iteration associated with this value
- **tag** (*str*) – Data identifier
- **metadata_header** (*Sequence[str]*) –
- **Shape** –
- **-----** –
- **mat** – where N is number of data and D is feature dimension
- **label_img** –

mat :Union[torch.Tensor, numpy.array]

metadata :Sequence[str]

label_img :torch.Tensor

global_step :int

tag :str

metadata_header :Sequence[str]

__repr__(*self*)

class flambe.logging.datatypes.**GraphT**
Bases: typing.NamedTuple

PyTorch Model with input and other keyword args

Supported by ModelSave NOT YET Supported by TensorboardX

model
PyTorch Model (should have *forward* and *state_dict* methods)

Type torch.nn.Module

input_to_model
Input to the model *forward* call

Type torch.autograd.Variable

verbose
Include extra detail

Type bool = False

kwargs
Other kwargs for model recording

Type Dict[str, Any] = {}

model :torch.nn.Module

input_to_model :torch.autograd.Variable

verbose :bool = False

kwargs :Dict[str, Any]

flambe.logging.datatypes.**DATA_TYPES**

class flambe.logging.datatypes.**DataLoggingFilter**(*default: bool = True*, *level: int = logging.NOTSET*, *dont_include: Optional[Tuple[type, ...]] = None*, ***kwargs: Any*)

Bases: logging.Filter

Filters on *DATA_TYPES* otherwise returns *default*

filter returns *self.default* if record is not a *DATA_TYPES* type; True if message is a *DATA_TYPES* type not in *dont_include* and high enough level; otherwise False

Parameters

- **default** (*bool*) – Returned when record is not one *DATA_TYPES*
- **level** (*int*) – Minimum level of records that are *DATA_TYPES* to be accepted
- **dont_include** (*Sequence[Type[Any]]*) – Types from *DATA_TYPES* to be excluded
- ****kwargs** (*Any*) – Additional kwargs to pass to *logging.Filter*

default
Returned when record is not one *DATA_TYPES*

Type bool

level
Minimum level of records that are *DATA_TYPES* to be accepted

Type int

dont_include
Types from *DATA_TYPES* to be excluded

Type Tuple[Type[Any]]

filter(*self*, *record: logging.LogRecord*)
Return True iff record should be accepted

Parameters **record** (*logging.LogRecord*) – logging record to be filtered

Returns True iff record should be accepted. *self.default* if record is not a *DATA_TYPES* type; True if message is a *DATA_TYPES* type not in *dont_include* and high enough level; otherwise False

Return type bool

25.2.2 flambe.logging.logging

Module Contents

flambe.logging.logging.**MB**

flambe.logging.logging.**setup_global_logging**(*console_log_level: int = logging.NOTSET*) → None

Set up flambe logging with a Stream handler and a Rotating File handler.

This method should be set before consuming any logger as it sets the basic configuration for all future logs.

After executing this method, all loggers will have the following handlers: * Stream handler: prints to std output all logs that above The console_log_level * Rotating File hanlder: 10MB log file located in Flambe global folder. Configured to store all logs (min level DEBUG)

Parameters **console_log_level** (*int*) – The minimum log level for the Stream handler

class flambe.logging.logging.**FlambeFilter**

Bases: logging.Filter

Filter all log records that don't come from flambe or main.

filter(*self*, *record: logging.LogRecord*)

class flambe.logging.logging.**TrialLogging**(*log_dir: str*, *verbose: bool = False*, *root_log_level: Optional[int] = None*, *capture_warnings: bool = True*, *console_prefix: Optional[str] = None*, *hyper_params: Optional[Dict] = None*)

__enter__(*self*)

__exit__(*self*, *exc_type: Any*, *exc_value: Any*, *traceback: Any*)

Close the listener and restore original logging config

class flambe.logging.logging.**ContextInjection**(***attrs*)

Add specified attributes to all log records

Parameters ****attrs** (*Any*) – Attributes that should be added to all log records, for use in downstream handlers

filter(*self*, *record: logging.LogRecord*)

__call__(*self*, *record: logging.LogRecord*)

class flambe.logging.logging.**TqdmFileWrapper**(*file: Any*)

Dummy file-like that will write to tqdm

Based on canoncial tqdm example

write(*self*, *x: AnyStr*)

flush(*self*)

flambe.logging.logging.**colorize_exceptions**() → None

Colorizes the system stderr ouput using pygments if installed

25.2.3 flambe.logging.utils

Module Contents

flambe.logging.utils.**ValueT**

flambe.logging.utils.**d :Dict[str, Callable]**

flambe.logging.utils.**coloredlogs**

flambe.logging.utils.**_get_context_logger**() → logging.Logger
Return the appropriate logger related to the module that logs.

flambe.logging.utils.**get_trial_dir**() → str
Get the output path used by the currently active trial.

Returns The output path

Return type str

flambe.logging.utils.**log**(*tag: str*, *data: ValueT*, *global_step: int*, *walltime: Optional[float] = None*) → None
Log data to tensorboard and console (convenience function)

Inspects type of data and uses the appropriate wrapper for tensorboard to consume the data. Supports floats (scalar), dictionary mapping tags to gloats (scalars), and strings (text).

Parameters

- **tag** (*str*) – Name of data, used as the tensorboard tag
- **data** (*ValueT*) – The scalar or text to log
- **global_step** (*int*) – Iteration number associated with data
- **walltime** (*Optional[float]*) – Walltime for data (the default is None).

Examples

Normally you would have to do the following to log a scalar >>> import logging; from flambe.logging import ScalarT >>> logger = logging.getLogger(__name__) >>> logger.info(ScalarT(tag, data, step, walltime)) But this method allows you to write a more concise statement with a common interface >>> from flambe.logging import log >>> log(tag, data, step)

flambe.logging.utils.**log_scalar**(*tag: str*, *data: float*, *global_step: int*, *walltime: Optional[float] = None*, *logger: Optional[logging.Logger] = None*) → None
Log tensorboard compatible scalar value with common interface

Parameters

- **tag** (*str*) – Tensorboard tag associated with scalar data
- **data** (*float*) – Scalar float value
- **global_step** (*int*) – The global step or iteration number
- **walltime** (*Optional[float]*) – Current walltime, for example from *time.time()*
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.utils.**log_scalars**(*tag: str, data: Dict[str, float], global_step: int, walltime: Optional[float] = None, logger: Optional[logging.Logger] = None*) → None
Log tensorboard compatible scalar values with common interface

Parameters

- **tag** (*str*) – Main tensorboard tag associated with all data
- **data** (*Dict[str, float]*) – Scalar float value
- **global_step** (*int*) – The global step or iteration number
- **walltime** (*Optional[float]*) – Current walltime, for example from *time.time()*
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.utils.**log_text**(*tag: str*, *data: str*, *global_step: int*, *walltime: Optional[float] = None*, *logger: Optional[logging.Logger] = None*) → None

Log tensorboard compatible text value with common interface

Parameters

- **tag** (*str*) – Tensorboard tag associated with data
- **data** (*str*) – Scalar float value
- **global_step** (*int*) – The global step or iteration number
- **walltime** (*Optional[float]*) – Current walltime, for example from *time.time()*
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.utils.**log_image**(*tag: str*, *data: str*, *global_step: int*, *walltime: Optional[float] = None*, *logger: Optional[logging.Logger] = None*) → None

Log tensorboard compatible image value with common interface

Parameters

- **tag** (*str*) – Tensorboard tag associated with data
- **data** (*str*) – Scalar float value
- **global_step** (*int*) – The global step or iteration number
- **walltime** (*Optional[float]*) – Current walltime, for example from *time.time()*
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.utils.**log_pr_curve**(*tag: str, labels: Union[torch.Tensor, numpy.array], predictions: Union[torch.Tensor, numpy.array], global_step: int, num_thresholds: int = 127, walltime: Optional[float] = None, logger: Optional[logging.Logger] = None*) → None

Log tensorboard compatible image value with common interface

Parameters

- **tag** (*str*) – Data identifier
- **labels** (*Union[torch.Tensor, numpy.array]*) – Containing 0, 1 values
- **predictions** (*Union[torch.Tensor, numpy.array]*) – Containing 0<=x<=1 values. Needs to match labels size
- **num_thresholds** (*int = 127*) – The number of thresholds to evaluate. Max value allowed 127.
- **weights** (*Optional[float] = None*) – No description provided.
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float*) – Wall clock time associated with this value
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.utils.**log_histogram**(*tag: str*, *data: str*, *global_step: int*, *bins: str = 'auto'*, *walltime: Optional[float] = None*, *logger: Optional[logging.Logger] = None*) → None

Log tensorboard compatible image value with common interface

Parameters

- **tag** (*str*) – Tensorboard tag associated with data
- **data** (*str*) – Scalar float value
- **global_step** (*int*) – The global step or iteration number
- **walltime** (*Optional[float]*) – Current walltime, for example from *time.time()*
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

25.3 Package Contents

class flambe.logging.**TrialLogging**(*log_dir: str*, *verbose: bool = False*, *root_log_level: Optional[int] = None*, *capture_warnings: bool = True*, *console_prefix: Optional[str] = None*, *hyper_params: Optional[Dict] = None*)

__enter__(*self*)

__exit__(*self*, *exc_type: Any*, *exc_value: Any*, *traceback: Any*)

Close the listener and restore original logging config

flambe.logging.**setup_global_logging**(*console_log_level: int = logging.NOTSET*) → None

Set up flambe logging with a Stream handler and a Rotating File handler.

This method should be set before consuming any logger as it sets the basic configuration for all future logs.

After executing this method, all loggers will have the following handlers: * Stream handler: prints to std output all logs that above The console_log_level * Rotating File hanlder: 10MB log file located in Flambe global folder. Configured to store all logs (min level DEBUG)

Parameters **console_log_level** (*int*) – The minimum log level for the Stream handler

class flambe.logging.**ScalarT**

Bases: typing.NamedTuple

A single scalar value

Supported by TensorboardX

Parameters

- **tag** (*str*) – Data identifier
- **scalar_value** (*float*) – The scalar value
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float = time.time()*) – Wall clock time associated with this value

tag :str

scalar_value :float

global_step :int

walltime :float

__repr__(*self*)

class flambe.logging.**ScalarsT**
Bases: typing.NamedTuple

A dictionary mapping tag keys to scalar values

Supported by TensorboardX

Parameters

- **main_tag** (*str*) – Parent name for all the children tags
- **tag_scalar_dict** (*Dict[str, float]*) – Mapping from scalar tags to their values
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float = time.time()*) – Wall clock time associated with this value

main_tag :str

tag_scalar_dict :Dict[str, float]

global_step :int

walltime :float

__repr__(*self*)

class flambe.logging.**HistogramT**
Bases: typing.NamedTuple

A histogram with an array of values

Supported by TensorboardX

Parameters

- **tag** (*str*) – Data identifier
- **values** (*Union[torch.Tensor, numpy.array]*) – Values to build histogram
- **global_step** (*int*) – Iteration associated with this value
- **bins** (*str*) – Determines how bins are made
- **walltime** (*float = time.time()*) – Wall clock time associated with this value

tag :str

values :Union[torch.Tensor, numpy.array]

global_step :int

bins :str

walltime :float

__repr__(*self*)

class flambe.logging.**TextT**
Bases: typing.NamedTuple

Text message

Supported by TensorboardX

Parameters

- **tag** (*str*) – Data identifier

- **text_string** (*str*) – String to record
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float*) – Wall clock time associated with this value

tag :str

text_string :str

global_step :int

walltime :float

__repr__(*self*)

class flambe.logging.**ImageT**
Bases: typing.NamedTuple

Image message

Supported by TensorboardX

Parameters

- **tag** (*str*) – Data identifier
- **img_tensor** (*Union*) – Image tensor to record
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float*) – Wall clock time associated with this value

tag :str

img_tensor :Union[torch.Tensor, numpy.array]

global_step :int

walltime :float

__repr__(*self*)

class flambe.logging.**PRCurveT**
Bases: typing.NamedTuple

PRCurve message

Supported by TensorboardX

Parameters

- **tag** (*str*) – Data identifier
- **labels** (*Union[torch.Tensor, numpy.array]*) – Containing 0, 1 values
- **predictions** (*Union[torch.Tensor, numpy.array]*) – Containing 0<=x<=1 values. Needs to match labels size
- **num_thresholds** (*int = 127*) – The number of thresholds to evaluate. Max value allowed 127.
- **weights** (*Optional[float] = None*) – No description provided.
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float*) – Wall clock time associated with this value

tag :str

labels :Union[torch.Tensor, numpy.array]

predictions :Union[torch.Tensor, numpy.array]

global_step :int

num_thresholds :int = 127

weights :Optional[float]

walltime :float

__repr__(*self*)

class flambe.logging.**EmbeddingT**

Bases: typing.NamedTuple

Embedding data, including array of vaues and metadata

Supported by TensorboardX

Parameters

- **mat** ((N, D)) – A matrix where each row is the feature vector of a data point
- **metadata** (*Sequence[str]*) – A list of labels; each element will be converted to string
- **label_img** ((N, C, H, W)) – Images corresponding to each data point
- **global_step** (*int*) – Iteration associated with this value
- **tag** (*str*) – Data identifier
- **metadata_header** (*Sequence[str]*) –
- **Shape** –
- **-----** –
- **mat** – where N is number of data and D is feature dimension
- **label_img** –

mat :Union[torch.Tensor, numpy.array]

metadata :Sequence[str]

label_img :torch.Tensor

global_step :int

tag :str

metadata_header :Sequence[str]

__repr__(*self*)

class flambe.logging.**GraphT**

Bases: typing.NamedTuple

PyTorch Model with input and other keyword args

Supported by ModelSave NOT YET Supported by TensorboardX

model

PyTorch Model (should have *forward* and *state_dict* methods)

Type torch.nn.Module

input_to_model
Input to the model *forward* call

Type torch.autograd.Variable

verbose
Include extra detail

Type bool = False

kwargs
Other kwargs for model recording

Type Dict[str, Any] = {}

model :torch.nn.Module

input_to_model :torch.autograd.Variable

verbose :bool = False

kwargs :Dict[str, Any]

flambe.logging.**log**(*tag: str*, *data: ValueT*, *global_step: int*, *walltime: Optional[float] = None*) → None
Log data to tensorboard and console (convenience function)

Inspects type of data and uses the appropriate wrapper for tensorboard to consume the data. Supports floats (scalar), dictionary mapping tags to gloats (scalars), and strings (text).

Parameters

- **tag** (*str*) – Name of data, used as the tensorboard tag
- **data** (*ValueT*) – The scalar or text to log
- **global_step** (*int*) – Iteration number associated with data
- **walltime** (*Optional[float]*) – Walltime for data (the default is None).

Examples

Normally you would have to do the following to log a scalar >>> import logging; from flambe.logging import ScalarT >>> logger = logging.getLogger(__name__) >>> logger.info(ScalarT(tag, data, step, walltime)) But this method allows you to write a more concise statement with a common interface >>> from flambe.logging import log >>> log(tag, data, step)

flambe.logging.**coloredlogs**

flambe.logging.**log_scalar**(*tag: str*, *data: float*, *global_step: int*, *walltime: Optional[float] = None*, *logger: Optional[logging.Logger] = None*) → None
Log tensorboard compatible scalar value with common interface

Parameters

- **tag** (*str*) – Tensorboard tag associated with scalar data
- **data** (*float*) – Scalar float value
- **global_step** (*int*) – The global step or iteration number
- **walltime** (*Optional[float]*) – Current walltime, for example from *time.time()*
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.**log_scalars**(*tag: str, data: Dict[str, float], global_step: int, walltime: Optional[float] = None, logger: Optional[logging.Logger] = None*) → None

Log tensorboard compatible scalar values with common interface

Parameters

- **tag** (*str*) – Main tensorboard tag associated with all data
- **data** (*Dict[str, float]*) – Scalar float value
- **global_step** (*int*) – The global step or iteration number
- **walltime** (*Optional[float]*) – Current walltime, for example from *time.time()*
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.**log_text**(*tag: str, data: str, global_step: int, walltime: Optional[float] = None, logger: Optional[logging.Logger] = None*) → None

Log tensorboard compatible text value with common interface

Parameters

- **tag** (*str*) – Tensorboard tag associated with data
- **data** (*str*) – Scalar float value
- **global_step** (*int*) – The global step or iteration number
- **walltime** (*Optional[float]*) – Current walltime, for example from *time.time()*
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.**log_image**(*tag: str, data: str, global_step: int, walltime: Optional[float] = None, logger: Optional[logging.Logger] = None*) → None

Log tensorboard compatible image value with common interface

Parameters

- **tag** (*str*) – Tensorboard tag associated with data
- **data** (*str*) – Scalar float value
- **global_step** (*int*) – The global step or iteration number
- **walltime** (*Optional[float]*) – Current walltime, for example from *time.time()*
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.**log_histogram**(*tag: str, data: str, global_step: int, bins: str = 'auto', walltime: Optional[float] = None, logger: Optional[logging.Logger] = None*) → None

Log tensorboard compatible image value with common interface

Parameters

- **tag** (*str*) – Tensorboard tag associated with data
- **data** (*str*) – Scalar float value
- **global_step** (*int*) – The global step or iteration number
- **walltime** (*Optional[float]*) – Current walltime, for example from *time.time()*
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.**log_pr_curve**(*tag: str, labels: Union[torch.Tensor, numpy.array], predictions: Union[torch.Tensor, numpy.array], global_step: int, num_thresholds: int = 127, walltime: Optional[float] = None, logger: Optional[logging.Logger] = None*) → None

Log tensorboard compatible image value with common interface

Parameters

- **tag** (*str*) – Data identifier
- **labels** (*Union[torch.Tensor, numpy.array]*) – Containing 0, 1 values
- **predictions** (*Union[torch.Tensor, numpy.array]*) – Containing 0<=x<=1 values. Needs to match labels size
- **num_thresholds** (*int = 127*) – The number of thresholds to evaluate. Max value allowed 127.
- **weights** (*Optional[float] = None*) – No description provided.
- **global_step** (*int*) – Iteration associated with this value
- **walltime** (*float*) – Wall clock time associated with this value
- **logger** (*Optional[logging.Logger]*) – logger to use for logging the scalar

flambe.logging.**get_trial_dir**() → str

Get the output path used by the currently active trial.

Returns The output path

Return type str

CHAPTER 26

flambe.metric

26.1 Subpackages

26.1.1 flambe.metric.dev

Submodules

flambe.metric.dev.accuracy

Module Contents

class flambe.metric.dev.accuracy.**Accuracy**
Bases: *flambe.metric.metric.Metric*

compute (*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Computes the loss.

Parameters

- **pred** (*Tensor*) – input logits of shape (B x N)
- **target** (*LontTensor*) – target tensor of shape (B) or (B x N)

Returns **accuracy** – single label accuracy, of shape (B)

Return type torch.Tensor

flambe.metric.dev.auc

Module Contents

class flambe.metric.dev.auc.**AUC**(*max_fpr=1.0*)
Bases: *flambe.metric.metric.Metric*

compute (*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute AUC at the given max false positive rate.

Parameters

- **pred** (*torch.Tensor*) – The model predictions
- **target** (*torch.Tensor*) – The binary targets

Returns The computed AUC

Return type torch.Tensor

flambe.metric.dev.binary

Module Contents

class flambe.metric.dev.binary.**BinaryMetric**(*threshold: float = 0.5*)
Bases: *flambe.metric.metric.Metric*

compute (*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute the metric given predictions and targets

Parameters

- **pred** (*Tensor*) – The model predictions
- **target** (*Tensor*) – The binary targets

Returns The computed binary metric

Return type float

compute_binary (*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute a binary-input metric.

Parameters

- **pred** (*torch.Tensor*) – Predictions made by the model. It should be a probability 0 <= p <= 1 for each sample, 1 being the positive class.
- **target** (*torch.Tensor*) – Ground truth. Each label should be either 0 or 1.

Returns The computed binary metric

Return type torch.float

class flambe.metric.dev.binary.**BinaryAccuracy**
Bases: *flambe.metric.dev.binary.BinaryMetric*

Compute binary accuracy.

```
` |True positives + True negatives| / N `
```

compute_binary (*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute binary accuracy.

Parameters

- **pred** (*torch.Tensor*) – Predictions made by the model. It should be a probability 0 <= p <= 1 for each sample, 1 being the positive class.
- **target** (*torch.Tensor*) – Ground truth. Each label should be either 0 or 1.

Returns The computed binary metric

Return type torch.float

class flambe.metric.dev.binary.**BinaryPrecision**(*threshold: float = 0.5*, *positive_label: int = 1*)

Bases: *flambe.metric.dev.binary.BinaryMetric*

Compute Binary Precision.

An example is considered negative when its score is below the specified threshold. Binary precition is computed as follows:

` |True positives| / |True Positives| + |False Positives| `

compute_binary(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)

Compute binary precision.

Parameters

- **pred** (*torch.Tensor*) – Predictions made by the model. It should be a probability 0 <= p <= 1 for each sample, 1 being the positive class.
- **target** (*torch.Tensor*) – Ground truth. Each label should be either 0 or 1.

Returns The computed binary metric

Return type torch.float

__str__(*self*)

Return the name of the Metric (for use in logging).

class flambe.metric.dev.binary.**BinaryRecall**(*threshold: float = 0.5*, *positive_label: int = 1*)

Bases: *flambe.metric.dev.binary.BinaryMetric*

Compute binary recall.

An example is considered negative when its score is below the specified threshold. Binary precition is computed as follows:

` |True positives| / |True Positives| + |False Negatives| `

compute_binary(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)

Compute binary recall.

Parameters

- **pred** (*torch.Tensor*) – Predictions made by the model. It should be a probability 0 <= p <= 1 for each sample, 1 being the positive class.
- **target** (*torch.Tensor*) – Ground truth. Each label should be either 0 or 1.

Returns The computed binary metric

Return type torch.float

__str__(*self*)

Return the name of the Metric (for use in logging).

class flambe.metric.dev.binary.**F1**(*threshold: float = 0.5*, *positive_label: int = 1*, *eps: float = 1e-08*)

Bases: *flambe.metric.dev.binary.BinaryMetric*

compute_binary(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)

Compute F1. Score, the harmonic mean between precision and recall.

Parameters

- **pred** (*torch.Tensor*) – Predictions made by the model. It should be a probability 0 <= p <= 1 for each sample, 1 being the positive class.
- **target** (*torch.Tensor*) – Ground truth. Each label should be either 0 or 1.

Returns The computed binary metric

Return type torch.float

flambe.metric.dev.bpc

Module Contents

class flambe.metric.dev.bpc.**BPC**
Bases: *flambe.metric.Metric*

Bits per character. Computed as log_2(perplexity)

compute(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute the bits per character given the input and target.

Parameters

- **pred** (*torch.Tensor*) – input logits of shape (B x N)
- **target** (*torch.LontTensor*) – target tensor of shape (B)

Returns Output perplexity

Return type torch.float

flambe.metric.dev.perplexity

Module Contents

class flambe.metric.dev.perplexity.**Perplexity**
Bases: *flambe.metric.Metric*

Token level perplexity, computed a exp(cross_entropy).

compute(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute the preplexity given the input and target.

Parameters

- **pred** (*torch.Tensor*) – input logits of shape (B x N)
- **target** (*torch.LontTensor*) – target tensor of shape (B)

Returns Output perplexity

Return type torch.float

26.1.2 flambe.metric.loss

Submodules

flambe.metric.loss.cross_entropy

Module Contents

class flambe.metric.loss.cross_entropy.**MultiLabelCrossEntropy**(*weight: Optional[torch.Tensor] = None, ignore_index: Optional[int] = None, reduction: str = 'mean'*)

Bases: *flambe.metric.metric.Metric*

compute(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)

Computes the multilabel cross entropy loss.

Parameters

- **pred** (*torch.Tensor*) – input logits of shape (B x N)
- **target** (*torch.LontTensor*) – target tensor of shape (B x N)

Returns **loss** – Multi label cross-entropy loss, of shape (B)

Return type torch.Tensor

flambe.metric.loss.nll_loss

Module Contents

class flambe.metric.loss.nll_loss.**MultiLabelNLLLoss**(*weight: Optional[torch.Tensor] = None, ignore_index: Optional[int] = None, reduction: str = 'mean'*)

Bases: *flambe.metric.metric.Metric*

compute(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)

Computes the Negative log likelihood loss for multilabel.

Parameters

- **pred** (*torch.Tensor*) – input logits of shape (B x N)
- **target** (*torch.LontTensor*) – target tensor of shape (B x N)

Returns **loss** – Multi label negative log likelihood loss, of shape (B)

Return type torch.float

26.2 Submodules

26.2.1 flambe.metric.metric

Module Contents

class flambe.metric.metric.**Metric**

Bases: *flambe.compile.Component*

Base Metric interface.

Objects implementing this interface should take in a sequence of examples and provide as output a processd list of the same size.

compute (*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Computes the metric over the given prediction and target.

Parameters

- **pred** (*torch.Tensor*) – The model predictions
- **target** (*torch.Tensor*) – The ground truth targets

Returns The computed metric

Return type torch.Tensor

__call__ (*self*, **args*, ***kwargs*)
Makes Featurizer a callable.

__str__ (*self*)
Return the name of the Metric (for use in logging).

26.3 Package Contents

class flambe.metric.**Metric**
Bases: *flambe.compile.Component*

Base Metric interface.

Objects implementing this interface should take in a sequence of examples and provide as output a processd list of the same size.

compute (*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Computes the metric over the given prediction and target.

Parameters

- **pred** (*torch.Tensor*) – The model predictions
- **target** (*torch.Tensor*) – The ground truth targets

Returns The computed metric

Return type torch.Tensor

__call__ (*self*, **args*, ***kwargs*)
Makes Featurizer a callable.

__str__ (*self*)
Return the name of the Metric (for use in logging).

class flambe.metric.**MultiLabelCrossEntropy** (*weight: Optional[torch.Tensor] = None*, *ignore_index: Optional[int] = None*, *reduction: str = 'mean'*)
Bases: *flambe.metric.metric.Metric*

compute (*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Computes the multilabel cross entropy loss.

Parameters

- **pred** (*torch.Tensor*) – input logits of shape (B x N)
- **target** (*torch.LontTensor*) – target tensor of shape (B x N)

Returns **loss** – Multi label cross-entropy loss, of shape (B)

Return type torch.Tensor

class flambe.metric.**MultiLabelNLLLoss**(*weight: Optional[torch.Tensor] = None*, *ignore_index: Optional[int] = None*, *reduction: str = 'mean'*)

Bases: *flambe.metric.metric.Metric*

compute(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)

Computes the Negative log likelihood loss for multilabel.

Parameters

- **pred** (*torch.Tensor*) – input logits of shape (B x N)
- **target** (*torch.LontTensor*) – target tensor of shape (B x N)

Returns **loss** – Multi label negative log likelihood loss, of shape (B)

Return type torch.float

class flambe.metric.**Accuracy**

Bases: *flambe.metric.metric.Metric*

compute(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)

Computes the loss.

Parameters

- **pred** (*Tensor*) – input logits of shape (B x N)
- **target** (*LontTensor*) – target tensor of shape (B) or (B x N)

Returns **accuracy** – single label accuracy, of shape (B)

Return type torch.Tensor

class flambe.metric.**Perplexity**

Bases: *flambe.metric.Metric*

Token level perplexity, computed a exp(cross_entropy).

compute(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)

Compute the preplexity given the input and target.

Parameters

- **pred** (*torch.Tensor*) – input logits of shape (B x N)
- **target** (*torch.LontTensor*) – target tensor of shape (B)

Returns Output perplexity

Return type torch.float

class flambe.metric.**BPC**

Bases: *flambe.metric.Metric*

Bits per character. Computed as log_2(perplexity)

compute(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)

Compute the bits per character given the input and target.

Parameters

- **pred** (*torch.Tensor*) – input logits of shape (B x N)
- **target** (*torch.LontTensor*) – target tensor of shape (B)

Returns Output perplexity

Return type torch.float

class flambe.metric.**AUC**(*max_fpr=1.0*)
Bases: *flambe.metric.metric.Metric*

compute(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute AUC at the given max false positive rate.

Parameters

- **pred** (*torch.Tensor*) – The model predictions
- **target** (*torch.Tensor*) – The binary targets

Returns The computed AUC

Return type torch.Tensor

class flambe.metric.**BinaryPrecision**(*threshold: float = 0.5*, *positive_label: int = 1*)
Bases: *flambe.metric.dev.binary.BinaryMetric*

Compute Binary Precision.

An example is considered negative when its score is below the specified threshold. Binary precition is computed as follows:

` |True positives| / |True Positives| + |False Positives| `

compute_binary(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute binary precision.

Parameters

- **pred** (*torch.Tensor*) – Predictions made by the model. It should be a probability 0 <= p <= 1 for each sample, 1 being the positive class.
- **target** (*torch.Tensor*) – Ground truth. Each label should be either 0 or 1.

Returns The computed binary metric

Return type torch.float

__str__(*self*)
Return the name of the Metric (for use in logging).

class flambe.metric.**BinaryRecall**(*threshold: float = 0.5*, *positive_label: int = 1*)
Bases: *flambe.metric.dev.binary.BinaryMetric*

Compute binary recall.

An example is considered negative when its score is below the specified threshold. Binary precition is computed as follows:

` |True positives| / |True Positives| + |False Negatives| `

compute_binary(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute binary recall.

Parameters

- **pred** (*torch.Tensor*) – Predictions made by the model. It should be a probability 0 <= p <= 1 for each sample, 1 being the positive class.
- **target** (*torch.Tensor*) – Ground truth. Each label should be either 0 or 1.

Returns The computed binary metric

Return type torch.float

__str__(*self*)
Return the name of the Metric (for use in logging).

class flambe.metric.**BinaryAccuracy**
Bases: *flambe.metric.dev.binary.BinaryMetric*

Compute binary accuracy.

` |True positives + True negatives| / N `

compute_binary(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute binary accuracy.

Parameters

- **pred** (*torch.Tensor*) – Predictions made by the model. It should be a probability 0 <= p <= 1 for each sample, 1 being the positive class.
- **target** (*torch.Tensor*) – Ground truth. Each label should be either 0 or 1.

Returns The computed binary metric

Return type torch.float

class flambe.metric.**F1**(*threshold: float = 0.5*, *positive_label: int = 1*, *eps: float = 1e-08*)
Bases: *flambe.metric.dev.binary.BinaryMetric*

compute_binary(*self*, *pred: torch.Tensor*, *target: torch.Tensor*)
Compute F1. Score, the harmonic mean between precision and recall.

Parameters

- **pred** (*torch.Tensor*) – Predictions made by the model. It should be a probability 0 <= p <= 1 for each sample, 1 being the positive class.
- **target** (*torch.Tensor*) – Ground truth. Each label should be either 0 or 1.

Returns The computed binary metric

Return type torch.float

CHAPTER 27

flambe.model

27.1 Submodules

27.1.1 flambe.model.logistic_regression

Module Contents

class flambe.model.logistic_regression.**LogisticRegression**(*input_size: int*)
Bases: *flambe.nn.module.Module*

Logistic regression model given an input vector v the forward calculation is sigmoid(Wv+b), where W is a weight vector and b a bias term. The result is then passed to a sigmoid function, which maps it as a real number in [0,1]. This is typically interpreted in classification settings as the probability of belonging to a given class.

input_size
Dimension (number of features) of the input vector.

Type int

forward(*self*, *data: Tensor*, *target: Optional[Tensor] = None*)
Forward pass that encodes data :param data: input data to encode :type data: Tensor :param target: target value, will be casted to a float tensor. :type target: Optional[Tensor]

27.2 Package Contents

class flambe.model.**LogisticRegression**(*input_size: int*)
Bases: *flambe.nn.module.Module*

Logistic regression model given an input vector v the forward calculation is sigmoid(Wv+b), where W is a weight vector and b a bias term. The result is then passed to a sigmoid function, which maps it as a real number in [0,1]. This is typically interpreted in classification settings as the probability of belonging to a given class.

input_size
Dimension (number of features) of the input vector.

Type int

forward(*self*, *data: Tensor*, *target: Optional[Tensor] = None*)
Forward pass that encodes data :param data: input data to encode :type data: Tensor :param target: target value, will be casted to a float tensor. :type target: Optional[Tensor]

CHAPTER 28

flambe.nlp

28.1 Subpackages

28.1.1 flambe.nlp.classification

Submodules

flambe.nlp.classification.datasets

Module Contents

class flambe.nlp.classification.datasets.**SSTDataset**(*binary: bool = True, phrases: bool = False, cache: bool = True, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official SST-1 dataset.

URL = https://raw.githubusercontent.com/harvardnlp/sent-conv-torch/master/data/

classmethod _load_file(*cls, path: str, sep: Optional[str] = 't', header: Optional[str] = None, columns: Optional[Union[List[str], List[int]]] = None, encoding: Optional[str] = 'utf-8'*)

Load data from the given path.

class flambe.nlp.classification.datasets.**TRECDataset**(*cache: bool = True, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official TREC dataset.

URL = https://raw.githubusercontent.com/harvardnlp/sent-conv-torch/master/data/

classmethod _load_file(*cls, path: str, sep: Optional[str] = 't', header: Optional[str] = None, columns: Optional[Union[List[str], List[int]]] = None, encoding: Optional[str] = 'latin-1'*)

Load data from the given path.

class flambe.nlp.classification.datasets.**NewsGroupDataset**(*cache: bool = False, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official 20 news group dataset.

flambe.nlp.classification.model

Module Contents

class flambe.nlp.classification.model.**TextClassifier**(*embedder: Embedder, output_layer: Module, dropout: float = 0*)

Bases: *flambe.nn.Module*

Implements a standard classifier.

The classifier is composed of an encoder module, followed by a fully connected output layer, with a dropout layer in between.

embedder
The embedder layer

Type *Embedder*

output_layer
The output layer, yields a probability distribution over targets

Type *Module*

drop
the dropout layer

Type nn.Dropout

loss
the loss function to optimize the model with

Type *Metric*

metric
the dev metric to evaluate the model on

Type *Metric*

forward(*self, data: Tensor, target: Optional[Tensor] = None*)
Run a forward pass through the network.

Parameters

- **data** (*Tensor*) – The input data
- **target** (*Tensor, optional*) – The input targets, optional

Returns The output predictions, and optionally the targets

Return type Union[Tensor, Tuple[Tensor, Tensor]

Package Contents

class flambe.nlp.classification.**SSTDataset**(*binary: bool = True, phrases: bool = False, cache: bool = True, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official SST-1 dataset.

URL = https://raw.githubusercontent.com/harvardnlp/sent-conv-torch/master/data/

classmethod _load_file(*cls, path: str, sep: Optional[str] = 't', header: Optional[str] = None, columns: Optional[Union[List[str], List[int]]] = None, encoding: Optional[str] = 'utf-8'*)

Load data from the given path.

class flambe.nlp.classification.**TRECDataset**(*cache: bool = True, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official TREC dataset.

URL = https://raw.githubusercontent.com/harvardnlp/sent-conv-torch/master/data/

classmethod _load_file(*cls, path: str, sep: Optional[str] = 't', header: Optional[str] = None, columns: Optional[Union[List[str], List[int]]] = None, encoding: Optional[str] = 'latin-1'*)

Load data from the given path.

class flambe.nlp.classification.**NewsGroupDataset**(*cache: bool = False, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official 20 news group dataset.

class flambe.nlp.classification.**TextClassifier**(*embedder: Embedder, output_layer: Module, dropout: float = 0*)

Bases: *flambe.nn.Module*

Implements a standard classifier.

The classifier is composed of an encoder module, followed by a fully connected output layer, with a dropout layer in between.

embedder
The embedder layer

Type *Embedder*

output_layer
The output layer, yields a probability distribution over targets

Type *Module*

drop
the dropout layer

Type nn.Dropout

loss
the loss function to optimize the model with

Type *Metric*

metric
the dev metric to evaluate the model on

Type *Metric*

forward(*self*, *data: Tensor*, *target: Optional[Tensor] = None*)
Run a forward pass through the network.

Parameters

- **data** (*Tensor*) – The input data
- **target** (*Tensor, optional*) – The input targets, optional

Returns The output predictions, and optionally the targets

Return type Union[Tensor, Tuple[Tensor, Tensor]

28.1.2 flambe.nlp.fewshot

Submodules

flambe.nlp.fewshot.model

Module Contents

class flambe.nlp.fewshot.model.**PrototypicalTextClassifier**(*embedder: Embedder*, *distance: str = 'euclidean'*, *detach_mean: bool = False*)

Bases: *flambe.nn.Module*

Implements a standard classifier.

The classifier is composed of an encoder module, followed by a fully connected output layer, with a dropout layer in between.

encoder
the encoder object

Type *Module*

decoder
the decoder layer

Type Decoder

drop
the dropout layer

Type nn.Dropout

loss
the loss function to optimize the model with

Type *Metric*

metric
the dev metric to evaluate the model on

Type *Metric*

compute_prototypes(*self*, *support: Tensor*, *label: Tensor*)
Set the current prototypes used for classification.

Parameters

- **data** (*torch.Tensor*) – Input encodings
- **label** (*torch.Tensor*) – Corresponding labels

forward(*self*, *query: Tensor*, *query_label: Optional[Tensor] = None*, *support: Optional[Tensor] = None*, *support_label: Optional[Tensor] = None*, *prototypes: Optional[Tensor] = None*)
Run a forward pass through the network.

Parameters **data** (*Tensor*) – The input data

Returns The output predictions

Return type Union[Tensor, Tuple[Tensor, Tensor]]

Package Contents

class flambe.nlp.fewshot.**PrototypicalTextClassifier**(*embedder: Embedder*, *distance: str = 'euclidean'*, *detach_mean: bool = False*)
Bases: *flambe.nn.Module*

Implements a standard classifier.

The classifier is composed of an encoder module, followed by a fully connected output layer, with a dropout layer in between.

encoder
the encoder object

Type *Module*

decoder
the decoder layer

Type Decoder

drop
the dropout layer

Type nn.Dropout

loss
the loss function to optimize the model with

Type *Metric*

metric
the dev metric to evaluate the model on

Type *Metric*

compute_prototypes(*self*, *support: Tensor*, *label: Tensor*)
Set the current prototypes used for classification.

Parameters

- **data** (*torch.Tensor*) – Input encodings
- **label** (*torch.Tensor*) – Corresponding labels

forward(*self*, *query: Tensor*, *query_label: Optional[Tensor] = None*, *support: Optional[Tensor] = None*, *support_label: Optional[Tensor] = None*, *prototypes: Optional[Tensor] = None*)
Run a forward pass through the network.

Parameters **data** (*Tensor*) – The input data

Returns The output predictions

Return type Union[Tensor, Tuple[Tensor, Tensor]]

28.1.3 flambe.nlp.language_modeling

Submodules

flambe.nlp.language_modeling.datasets

Module Contents

class flambe.nlp.language_modeling.datasets.**PTBDataset**(*split_by_sentence: bool = False*, *end_of_line_token: Optional[str] = '<eol>'*, *cache: bool = False*, *transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official PTB dataset.

PTB_URL = https://raw.githubusercontent.com/yoonkim/lstm-char-cnn/master/data/ptb/

_process(*self*, *file: bytes*)
Process the input file.

Parameters **field** (*str*) – The input file, as bytes

Returns List of examples, where each example is a single element tuple containing the text.

Return type List[Tuple[str]]

class flambe.nlp.language_modeling.datasets.**Wiki103**(*split_by_sentence: bool = False*, *end_of_line_token: Optional[str] = '<eol>'*, *cache: bool = False*, *transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official WikiText103 dataset.

WIKI_URL = https://s3.amazonaws.com/research.metamind.io/wikitext/wikitext-103-v1.zip

_process(*self*, *file: bytes*)
Process the input file.

Parameters **file** (*bytes*) – The input file, as a byte string

Returns List of examples, where each example is a single element tuple containing the text.

Return type List[Tuple[str]]

class flambe.nlp.language_modeling.datasets.**Enwiki8**(*num_eval_symbols: int = 5000000, remove_end_of_line: bool = False, cache: bool = False, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official WikiText103 dataset.

ENWIKI_URL = http://mattmahoney.net/dc/enwik8.zip

_process(*self*, *file: bytes*)

Process the input file.

Parameters **file** (*bytes*) – The input file, as a byte string

Returns List of examples, where each example is a single element tuple containing the text.

Return type List[Tuple[str]]

flambe.nlp.language_modeling.fields

Module Contents

class flambe.nlp.language_modeling.fields.**LMField**(***kwargs*)

Bases: *flambe.field.TextField*

Language Model field.

Generates the original tensor alongside its shifted version.

process(*self*, *example: str*)

Process an example and create 2 Tensors.

Parameters **example** (*str*) – The example to process, as a single string

Returns The processed example, tokenized and numericalized

Return type Tuple[torch.Tensor, ..]

flambe.nlp.language_modeling.model

Module Contents

class flambe.nlp.language_modeling.model.**LanguageModel**(*embedder: Embedder, output_layer: Module, dropout: float = 0, pad_index: int = 0, tie_weights: bool = False, tie_weight_attr: str = 'embedding'*)

Bases: *flambe.nn.Module*

Implement an LanguageModel model for sequential classification.

This model can be used to language modeling, as well as other sequential classification tasks. The full sequence predictions are produced by the model, effectively making the number of examples the batch size multiplied by the sequence length.

forward (*self*, *data: Tensor*, *target: Optional[Tensor] = None*)
Run a forward pass through the network.

Parameters **data** (*Tensor*) – The input data

Returns The output predictions of shape seq_len x batch_size x n_out

Return type Union[Tensor, Tuple[Tensor, Tensor]]

flambe.nlp.language_modeling.sampler

Module Contents

class flambe.nlp.language_modeling.sampler.**CorpusSampler**(*batch_size: int = 128*, *unroll_size: int = 128*, *n_workers: int = 0*, *pin_memory: bool = False*, *downsample: Optional[float] = None*, *drop_last: bool = True*)

Bases: *flambe.sampler.sampler.Sampler*

Implement a CorpusSampler object.

This object is useful for iteration over a large corpus of text in an ordered way. It takes as input a dataset with a single example containing the sequence of tokens and will yield batches that contain both source sequences of tensors corresponding to the Corpus's text, and these same sequences shifted by one as the target.

static collate_fn (*data: Sequence[Tuple[Tensor, Tensor]]*)
Create a batch from data.

Parameters **data** (*Sequence[Tuple[Tensor, Tensor]]*) – List of (source, target) tuples.

Returns Source and target Tensors.

Return type Tuple[Tensor, Tensor]

sample (*self, data: Sequence[Sequence[Tensor]], n_epochs: int = 1*)
Sample from the list of features and yields batches.

Parameters

- **data** (*Sequence[Sequence[Tensor, ..]]*) – The input data to sample from
- **n_epochs** (*int, optional*) – The number of epochs to run in the output iterator. Use -1 to run infinitely.

Yields *Iterator[Tuple[Tensor]]* – A batch of data, as a tuple of Tensors

length (*self, data: Sequence[Sequence[torch.Tensor]]*)
Return the number of batches in the sampler.

Parameters **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from

Returns The number of batches that would be created per epoch

Return type int

Package Contents

class flambe.nlp.language_modeling.**PTBDataset**(*split_by_sentence: bool = False, end_of_line_token: Optional[str] = '<eol>', cache: bool = False, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official PTB dataset.

PTB_URL = https://raw.githubusercontent.com/yoonkim/lstm-char-cnn/master/data/ptb/

_process(*self*, *file: bytes*)

Process the input file.

Parameters **field** (*str*) – The input file, as bytes

Returns List of examples, where each example is a single element tuple containing the text.

Return type List[Tuple[str]]

class flambe.nlp.language_modeling.**Wiki103**(*split_by_sentence: bool = False, end_of_line_token: Optional[str] = '<eol>', cache: bool = False, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official WikiText103 dataset.

WIKI_URL = https://s3.amazonaws.com/research.metamind.io/wikitext/wikitext-103-v1.zip

_process(*self*, *file: bytes*)

Process the input file.

Parameters **file** (*bytes*) – The input file, as a byte string

Returns List of examples, where each example is a single element tuple containing the text.

Return type List[Tuple[str]]

class flambe.nlp.language_modeling.**Enwiki8**(*num_eval_symbols: int = 5000000, remove_end_of_line: bool = False, cache: bool = False, transform: Dict[str, Union[Field, Dict]] = None*)

Bases: *flambe.dataset.TabularDataset*

The official WikiText103 dataset.

ENWIKI_URL = http://mattmahoney.net/dc/enwik8.zip

_process(*self*, *file: bytes*)

Process the input file.

Parameters **file** (*bytes*) – The input file, as a byte string

Returns List of examples, where each example is a single element tuple containing the text.

Return type List[Tuple[str]]

class flambe.nlp.language_modeling.**LMField**(***kwargs*)

Bases: *flambe.field.TextField*

Language Model field.

Generates the original tensor alongside its shifted version.

process (*self*, *example: str*)
Process an example and create 2 Tensors.

Parameters **example** (*str*) – The example to process, as a single string

Returns The processed example, tokenized and numericalized

Return type Tuple[torch.Tensor, ..]

class flambe.nlp.language_modeling.**LanguageModel** (*embedder: Embedder*, *output_layer: Module*, *dropout: float = 0*, *pad_index: int = 0*, *tie_weights: bool = False*, *tie_weight_attr: str = 'embedding'*)
Bases: *flambe.nn.Module*

Implement an LanguageModel model for sequential classification.

This model can be used to language modeling, as well as other sequential classification tasks. The full sequence predictions are produced by the model, effectively making the number of examples the batch size multiplied by the sequence length.

forward (*self*, *data: Tensor*, *target: Optional[Tensor] = None*)
Run a forward pass through the network.

Parameters **data** (*Tensor*) – The input data

Returns The output predictions of shape seq_len x batch_size x n_out

Return type Union[Tensor, Tuple[Tensor, Tensor]]

class flambe.nlp.language_modeling.**CorpusSampler** (*batch_size: int = 128*, *unroll_size: int = 128*, *n_workers: int = 0*, *pin_memory: bool = False*, *downsample: Optional[float] = None*, *drop_last: bool = True*)
Bases: *flambe.sampler.sampler.Sampler*

Implement a CorpusSampler object.

This object is useful for iteration over a large corpus of text in an ordered way. It takes as input a dataset with a single example containing the sequence of tokens and will yield batches that contain both source sequences of tensors corresponding to the Corpus's text, and these same sequences shifted by one as the target.

static collate_fn (*data: Sequence[Tuple[Tensor, Tensor]]*)
Create a batch from data.

Parameters **data** (*Sequence[Tuple[Tensor, Tensor]]*) – List of (source, target) tuples.

Returns Source and target Tensors.

Return type Tuple[Tensor, Tensor]

sample (*self, data: Sequence[Sequence[Tensor]], n_epochs: int = 1*)
Sample from the list of features and yields batches.

Parameters

- **data** (*Sequence[Sequence[Tensor, ..]]*) – The input data to sample from
- **n_epochs** (*int, optional*) – The number of epochs to run in the output iterator. Use -1 to run infinitely.

Yields *Iterator[Tuple[Tensor]]* – A batch of data, as a tuple of Tensors

length (*self, data: Sequence[Sequence[torch.Tensor]]*)
Return the number of batches in the sampler.

Parameters **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from

Returns The number of batches that would be created per epoch

Return type int

28.1.4 flambe.nlp.transformers

Submodules

flambe.nlp.transformers.field

Module Contents

class flambe.nlp.transformers.field.**PretrainedTransformerField**(*alias: str, cache_dir: Optional[str] = None, max_len_truncate: Optional[int] = None, add_special_tokens: bool = True, **kwargs*)

Bases: *flambe.field.Field*

Field intergation of the transformers library.

Instantiate this object using any alias available in the *transformers* library. More information can be found here:

https://huggingface.co/transformers/

padding_idx :int
Get the padding index.

Returns The padding index in the vocabulary

Return type int

vocab_size :int
Get the vocabulary length.

Returns The length of the vocabulary

Return type int

process (*self, example: str*)
Process an example, and create a Tensor.

Parameters **example** (*str*) – The example to process, as a single string

Returns The processed example, tokenized and numericalized

Return type torch.Tensor

flambe.nlp.transformers.model

Module Contents

class flambe.nlp.transformers.model.**PretrainedTransformerEmbedder**(*alias: str*, *cache_dir: Optional[str] = None*, *padding_idx: Optional[int] = None*, *pool: bool = False*, ***kwargs*)

Bases: *flambe.nn.Module*

Embedder intergation of the transformers library.

Instantiate this object using any alias available in the *transformers* library. More information can be found here:

https://huggingface.co/transformers/

forward(*self*, *data: torch.Tensor*, *token_type_ids: Optional[torch.Tensor] = None*, *attention_mask: Optional[torch.Tensor] = None*, *position_ids: Optional[torch.Tensor] = None*, *head_mask: Optional[torch.Tensor] = None*)

Perform a forward pass through the network.

If pool was provided, will only return the pooled output of shape [B x H]. Otherwise, returns the full sequence encoding of shape [S x B x H].

Parameters

- **data** (*torch.Tensor*) – The input data of shape [B x S]
- **token_type_ids** (*Optional[torch.Tensor], optional*) – Segment token indices to indicate first and second portions of the inputs. Indices are selected in `[0, 1]`: `0` corresponds to a *sentence A* token, `1` corresponds to a *sentence B* token. Has shape [B x S]
- **attention_mask** (*Optional[torch.Tensor], optional*) – FloatTensor of shape [B x S]. Masked values should be 0 for padding tokens, 1 otherwise.
- **position_ids** (*Optional[torch.Tensor], optional*) – Indices of positions of each input sequence tokens in the position embedding. Defaults to the order given in the input. Has shape [B x S].
- **head_mask** (*Optional[torch.Tensor], optional*) – Mask to nullify selected heads of the self-attention modules. Should be 0 for heads to mask, 1 otherwise. Has shape [num_layers x num_heads]

Returns If pool is True, returns a tneosr of shape [B x H], else returns an encoding for each token in the sequence of shape [B x S x H].

Return type torch.Tensor

__getattr__(*self*, *name: str*)

Override getattr to inspect config.

Parameters **name** (*str*) – The attribute to fetch

Returns The attribute

Return type Any

Package Contents

class flambe.nlp.transformers.**PretrainedTransformerField**(*alias: str*, *cache_dir: Optional[str] = None*, *max_len_truncate: Optional[int] = None*, *add_special_tokens: bool = True*, ***kwargs*)

Bases: *flambe.field.Field*

Field intergation of the transformers library.

Instantiate this object using any alias available in the *transformers* library. More information can be found here:

https://huggingface.co/transformers/

padding_idx :int
Get the padding index.

Returns The padding index in the vocabulary

Return type int

vocab_size :int
Get the vocabulary length.

Returns The length of the vocabulary

Return type int

process(*self*, *example: str*)
Process an example, and create a Tensor.

Parameters **example** (*str*) – The example to process, as a single string

Returns The processed example, tokenized and numericalized

Return type torch.Tensor

class flambe.nlp.transformers.**PretrainedTransformerEmbedder**(*alias: str*, *cache_dir: Optional[str] = None*, *padding_idx: Optional[int] = None*, *pool: bool = False*, ***kwargs*)

Bases: *flambe.nn.Module*

Embedder intergation of the transformers library.

Instantiate this object using any alias available in the *transformers* library. More information can be found here:

https://huggingface.co/transformers/

forward(*self*, *data: torch.Tensor*, *token_type_ids: Optional[torch.Tensor] = None*, *attention_mask: Optional[torch.Tensor] = None*, *position_ids: Optional[torch.Tensor] = None*, *head_mask: Optional[torch.Tensor] = None*)
Perform a forward pass through the network.

If pool was provided, will only return the pooled output of shape [B x H]. Otherwise, returns the full sequence encoding of shape [S x B x H].

Parameters

- **data** (*torch.Tensor*) – The input data of shape [B x S]
- **token_type_ids** (*Optional[torch.Tensor], optional*) – Segment token indices to indicate first and second portions of the inputs. Indices are selected in `[0, 1]`: `0` corresponds to a *sentence A* token, `1` corresponds to a *sentence B* token. Has shape [B x S]
- **attention_mask** (*Optional[torch.Tensor], optional*) – FloatTensor of shape [B x S]. Masked values should be 0 for padding tokens, 1 otherwise.
- **position_ids** (*Optional[torch.Tensor], optional*) – Indices of positions of each input sequence tokens in the position embedding. Defaults to the order given in the input. Has shape [B x S].
- **head_mask** (*Optional[torch.Tensor], optional*) – Mask to nullify selected heads of the self-attention modules. Should be 0 for heads to mask, 1 otherwise. Has shape [num_layers x num_heads]

Returns If pool is True, returns a tneosr of shape [B x H], else returns an encoding for each token in the sequence of shape [B x S x H].

Return type torch.Tensor

__getattr__(*self*, *name: str*)

Override getattr to inspect config.

Parameters **name** (*str*) – The attribute to fetch

Returns The attribute

Return type Any

CHAPTER 29

flambe.nn

29.1 Subpackages

29.1.1 flambe.nn.distance

Submodules

flambe.nn.distance.cosine

Module Contents

class flambe.nn.distance.cosine.**CosineDistance**(*eps: float = 1e-08*)

Bases: *flambe.nn.distance.DistanceModule*

Implement a CosineDistance object.

forward(*self*, *mat_1: Tensor*, *mat_2: Tensor*)

Returns the cosine distance between each element in mat_1 and each element in mat_2.

Parameters

- **mat_1** (*torch.Tensor*) – matrix of shape (n_1, n_features)
- **mat_2** (*torch.Tensor*) – matrix of shape (n_2, n_features)

Returns **dist** – distance matrix of shape (n_1, n_2)

Return type torch.Tensor

class flambe.nn.distance.cosine.**CosineMean**

Bases: *flambe.nn.distance.MeanModule*

Implement a CosineMean object.

forward(*self*, *data: Tensor*)

Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type torch.Tensor

flambe.nn.distance.distance

Module Contents

class flambe.nn.distance.distance.**DistanceModule**
Bases: *flambe.nn.module.Module*

Implement a DistanceModule object.

forward(*self*, *mat_1: Tensor*, *mat_2: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type torch.Tensor

class flambe.nn.distance.distance.**MeanModule**(*detach_mean: bool = False*)
Bases: *flambe.nn.module.Module*

Implement a MeanModule object.

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type torch.Tensor

flambe.nn.distance.euclidean

Module Contents

class flambe.nn.distance.euclidean.**EuclideanDistance**
Bases: *flambe.nn.distance.distance.DistanceModule*

Implement a EuclideanDistance object.

forward(*self*, *mat_1: Tensor*, *mat_2: Tensor*)
Returns the squared euclidean distance between each element in mat_1 and each element in mat_2.

Parameters

- **mat_1** (*torch.Tensor*) – matrix of shape (n_1, n_features)
- **mat_2** (*torch.Tensor*) – matrix of shape (n_2, n_features)

Returns **dist** – distance matrix of shape (n_1, n_2)

Return type torch.Tensor

class flambe.nn.distance.euclidean.**EuclideanMean**
Bases: *flambe.nn.distance.distance.MeanModule*

Implement a EuclideanMean object.

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type torch.Tensor

flambe.nn.distance.hyperbolic

Module Contents

flambe.nn.distance.hyperbolic.**EPSILON = 1e-05**

flambe.nn.distance.hyperbolic.**arccosh**(*x*)
Compute the arcosh, numerically stable.

flambe.nn.distance.hyperbolic.**mdot**(*x*, *y*)
Compute the inner product.

flambe.nn.distance.hyperbolic.**dist**(*x*, *y*)
Get the hyperbolic distance between x and y.

flambe.nn.distance.hyperbolic.**project**(*x*)
Project onto the hyeprboloid embedded in in n+1 dimensions.

flambe.nn.distance.hyperbolic.**log_map**(*x*, *y*)
Perform the log step.

flambe.nn.distance.hyperbolic.**norm**(*x*)
Compute the norm

flambe.nn.distance.hyperbolic.**exp_map**(*x*, *y*)
Perform the exp step.

flambe.nn.distance.hyperbolic.**loss**(*x*, *y*)
Get the loss for the optimizer.

class flambe.nn.distance.hyperbolic.**HyperbolicDistance**
Bases: *flambe.nn.distance.distance.DistanceModule*

Implement a HyperbolicDistance object.

forward(*self*, *mat_1: Tensor*, *mat_2: Tensor*)
Returns the squared euclidean distance between each element in mat_1 and each element in mat_2.

Parameters

- **mat_1** (*torch.Tensor*) – matrix of shape (n_1, n_features)
- **mat_2** (*torch.Tensor*) – matrix of shape (n_2, n_features)

Returns **dist** – distance matrix of shape (n_1, n_2)

Return type torch.Tensor

class flambe.nn.distance.hyperbolic.**HyperbolicMean**
Bases: *flambe.nn.distance.distance.MeanModule*

Compute the mean point in the hyperboloid model.

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type torch.Tensor

Package Contents

class flambe.nn.distance.**DistanceModule**
Bases: *flambe.nn.module.Module*

Implement a DistanceModule object.

forward(*self*, *mat_1: Tensor*, *mat_2: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type torch.Tensor

class flambe.nn.distance.**MeanModule**(*detach_mean: bool = False*)
Bases: *flambe.nn.module.Module*

Implement a MeanModule object.

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type torch.Tensor

class flambe.nn.distance.**EuclideanDistance**
Bases: *flambe.nn.distance.distance.DistanceModule*

Implement a EuclideanDistance object.

forward(*self*, *mat_1: Tensor*, *mat_2: Tensor*)
Returns the squared euclidean distance between each element in mat_1 and each element in mat_2.

Parameters

- **mat_1** (*torch.Tensor*) – matrix of shape (n_1, n_features)
- **mat_2** (*torch.Tensor*) – matrix of shape (n_2, n_features)

Returns **dist** – distance matrix of shape (n_1, n_2)

Return type torch.Tensor

class flambe.nn.distance.**EuclideanMean**
Bases: *flambe.nn.distance.distance.MeanModule*

Implement a EuclideanMean object.

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type torch.Tensor

class flambe.nn.distance.**CosineDistance**(*eps: float = 1e-08*)
Bases: *flambe.nn.distance.DistanceModule*

Implement a CosineDistance object.

forward(*self*, *mat_1: Tensor*, *mat_2: Tensor*)
Returns the cosine distance between each element in mat_1 and each element in mat_2.

Parameters

- **mat_1** (*torch.Tensor*) – matrix of shape (n_1, n_features)
- **mat_2** (*torch.Tensor*) – matrix of shape (n_2, n_features)

Returns **dist** – distance matrix of shape (n_1, n_2)

Return type torch.Tensor

class flambe.nn.distance.**CosineMean**
Bases: *flambe.nn.distance.MeanModule*

Implement a CosineMean object.

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type torch.Tensor

class flambe.nn.distance.**HyperbolicDistance**
Bases: *flambe.nn.distance.distance.DistanceModule*

Implement a HyperbolicDistance object.

forward(*self*, *mat_1: Tensor*, *mat_2: Tensor*)
Returns the squared euclidean distance between each element in mat_1 and each element in mat_2.

Parameters

- **mat_1** (*torch.Tensor*) – matrix of shape (n_1, n_features)
- **mat_2** (*torch.Tensor*) – matrix of shape (n_2, n_features)

Returns **dist** – distance matrix of shape (n_1, n_2)

Return type torch.Tensor

class flambe.nn.distance.**HyperbolicMean**
Bases: *flambe.nn.distance.distance.MeanModule*

Compute the mean point in the hyperboloid model.

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type torch.Tensor

flambe.nn.distance.**get_distance_module**(*metric: str*) → DistanceModule
Get the distance module from a string alias.

Currently available: . *euclidean* . *cosine* . *hyperbolic*

Parameters **metric** (*str*) – The distance metric to use

Raises ValueError – Unvalid distance string alias provided

Returns The instantiated distance module

Return type *DistanceModule*

flambe.nn.distance.**get_mean_module**(*metric: str*) → MeanModule
Get the mean module from a string alias.

Currently available: . *euclidean* . *cosine* . *hyperbolic*

Parameters **metric** (*str*) – The distance metric to use

Raises ValueError – Unvalid distance string alias provided

Returns The instantiated distance module

Return type *DistanceModule*

29.2 Submodules

29.2.1 flambe.nn.cnn

Module Contents

flambe.nn.cnn.**conv_block**(*conv_mod: nn.Module*, *activation: nn.Module*, *pooling: nn.Module*, *dropout: float*, *batch_norm: Optional[nn.Module] = None*) → nn.Module
Return a convolutional block.

class flambe.nn.cnn.**CNNEncoder**(*input_channels: int, channels: List[int], conv_dim: int = 2, kernel_size: Union[int, List[Union[Tuple[int, ...], int]]] = 3, activation: nn.Module = None, pooling: nn.Module = None, dropout: float = 0, batch_norm: bool = True, stride: int = 1, padding: int = 0*)
Bases: *flambe.nn.module.Module*

Implements a multi-layer n-dimensional CNN.

This module can be used to create multi-layer CNN models.

cnn
The cnn submodule

Type *nn.Module*

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type Union[Tensor, Tuple[Tensor, ..]]

29.2.2 flambe.nn.embedding

Module Contents

class flambe.nn.embedding.**Embeddings**(*num_embeddings: int*, *embedding_dim: int*, *padding_idx: int = 0*, *max_norm: Optional[float] = None*, *norm_type: float = 2.0*, *scale_grad_by_freq: bool = False*, *sparse: bool = False*, *positional_encoding: bool = False*, *positional_learned: bool = False*, *positonal_max_length: int = 5000*)

Bases: *flambe.nn.module.Module*

Implement an Embeddings module.

This object replicates the usage of nn.Embedding but registers the from_pretrained classmethod to be used inside a Flambé configuration, as this does not happen automatically during the registration of PyTorch objects.

The module also adds optional positional encoding, which can either be sinusoidal or learned during training. For the non-learned positional embeddings, we use sine and cosine functions of different frequencies.

classmethod from_pretrained(*cls*, *embeddings: Tensor*, *freeze: bool = True*, *padding_idx: int = 0*, *max_norm: Optional[float] = None*, *norm_type: float = 2.0*, *scale_grad_by_freq: bool = False*, *sparse: bool = False*, *positional_encoding: bool = False*, *positional_learned: bool = False*, *positonal_max_length: int = 5000*, *positonal_embeddings: Optional[Tensor] = None*, *positonal_freeze: bool = True*)

Create an Embeddings instance from pretrained embeddings.

Parameters

- **embeddings** (*torch.Tensor*) – FloatTensor containing weights for the Embedding. First dimension is being passed to Embedding as num_embeddings, second as embedding_dim.
- **freeze** (*bool*) – If True, the tensor does not get updated in the learning process. Default: True
- **padding_idx** (*int, optional*) – Pads the output with the embedding vector at `padding_idx` (initialized to zeros) whenever it encounters the index, by default 0
- **max_norm** (*Optional[float], optional*) – If given, each embedding vector with norm larger than `max_norm` is normalized to have norm `max_norm`
- **norm_type** (*float, optional*) – The p of the p-norm to compute for the `max_norm` option. Default `2`.
- **scale_grad_by_freq** (*bool, optional*) – If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`.
- **sparse** (*bool, optional*) – If `True`, gradient w.r.t. `weight` matrix will be a sparse tensor. See Notes for more details.
- **positional_encoding** (*bool, optional*) – If True, adds positonal encoding to the token embeddings. By default, the embeddings are frozen sinusodial embeddings. To learn these during training, set positional_learned. Default `False`.

- **positional_learned** (*bool, optional*) – Learns the positional embeddings during training instead of using frozen sinusodial ones. Default `False`.
- **positonal_embeddings** (*torch.Tensor, optional*) – If given, also replaces the positonal embeddings with this matrix. The max length will be ignored and replaced by the dimension of this matrix.
- **positonal_freeze** (*bool, optional*) – Whether the positonal embeddings should be frozen

forward(*self*, *data: Tensor*)
Perform a forward pass.

Parameters **data** (*Tensor*) – The input tensor of shape [S x B]

Returns The output tensor of shape [S x B x E]

Return type Tensor

class flambe.nn.embedding.**Embedder**(*embedding: Module*, *encoder: Module*, *pooling: Optional[Module] = None*, *embedding_dropout: float = 0*, *padding_idx: Optional[int] = 0*)
Bases: *flambe.nn.module.Module*

Implements an Embedder module.

An Embedder takes as input a sequence of index tokens, and computes the corresponding embedded representations, and padding mask. The encoder may be initialized using a pretrained embedding matrix.

embeddings
The embedding module

Type *Module*

encoder
The sub-encoder that this object is wrapping

Type *Module*

pooling
An optional pooling module

Type *Module*

drop
The dropout layer

Type nn.Dropout

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor of shape [S x B]

Returns The encoded output, as a float tensor. May return a state if the encoder is an RNN and no pooling is provided.

Return type Union[Tensor, Tuple[Tensor, Tensor]]

29.2.3 flambe.nn.mlp

Module Contents

class flambe.nn.mlp.**MLPEncoder**(*input_size: int*, *output_size: int*, *n_layers: int = 1*, *dropout: float = 0.0*, *output_activation: Optional[nn.Module] = None*, *hidden_size: Optional[int] = None*, *hidden_activation: Optional[nn.Module] = None*)

Bases: *flambe.nn.module.Module*

Implements a multi layer feed forward network.

This module can be used to create output layers, or more complex multi-layer feed forward networks.

seq
the sequence of layers and activations

Type *nn.Sequential*

forward(*self*, *data: torch.Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – input to the model of shape (batch_size, input_size)

Returns **output** – output of the model of shape (batch_size, output_size)

Return type torch.Tensor

29.2.4 flambe.nn.module

Module Contents

class flambe.nn.module.**Module**

Bases: *flambe.compile.Component*, torch.nn.Module

Base Flambé Module inteface.

Provides the exact same interface as Pytorch's nn.Module, but extends it with a useful set of methods to access and clip parameters, as well as gradients.

This abstraction allows users to convert their modules with a single line change, by importing from Flambé instead. Just like every Pytorch module, a forward method should be implemented.

named_trainable_params :Iterator[Tuple[str, nn.Parameter]]
Get all the named parameters with *requires_grad=True*.

Returns Iterator over the parameters and their name.

Return type Iterator[Tuple[str, nn.Parameter]]

trainable_params :Iterator[nn.Parameter]
Get all the parameters with *requires_grad=True*.

Returns Iterator over the parameters

Return type Iterator[nn.Parameter]

gradient_norm :float
Compute the average gradient norm.

Returns The current average gradient norm

Return type float

parameter_norm :float
Compute the average parameter norm.

Returns The current average parameter norm

Return type float

num_parameters(*self*, *trainable=False*)
Gets the number of parameters in the model.

Returns number of model params

Return type int

clip_params(*self*, *threshold: float*)
Clip the parameters to the given range.

Parameters **float** – Values are clipped between -threshold, threshold

clip_gradient_norm(*self*, *threshold: float*)
Clip the norm of the gradient by the given value.

Parameters **float** – Threshold to clip at

29.2.5 flambe.nn.mos

Module Contents

class flambe.nn.mos.**MixtureOfSoftmax**(*input_size: int*, *output_size: int*, *k: int = 1*, *take_log: bool = True*)
Bases: *flambe.nn.module.Module*

Implement the MixtureOfSoftmax output layer.

pi
softmax layer over the different softmax

Type FullyConnected

layers
list of the k softmax layers

Type [FullyConnected]

forward(*self*, *data: Tensor*)
Implement mixture of softmax for language modeling.

Parameters **data** (*torch.Tensor*) – seq_len x batch_size x hidden_size

Returns **out** – output matrix of shape seq_len x batch_size x out_size

Return type Variable

29.2.6 flambe.nn.pooling

Module Contents

class flambe.nn.pooling.**FirstPooling**
Bases: *flambe.nn.Module*

Get the last hidden state of a sequence.

forward(*self*, *data: torch.Tensor*, *padding_mask: Optional[torch.Tensor] = None*)
Performs a forward pass.

Parameters

- **data** (*torch.Tensor*) – The input data, as a tensor of shape [B x S x H]
- **padding_mask** (*torch.Tensor*) – The input mask, as a tensor of shape [B X S]

Returns The output data, as a tensor of shape [B x H]

Return type torch.Tensor

class flambe.nn.pooling.**LastPooling**
Bases: *flambe.nn.Module*

Get the last hidden state of a sequence.

forward(*self*, *data: torch.Tensor*, *padding_mask: Optional[torch.Tensor] = None*)
Performs a forward pass.

Parameters

- **data** (*torch.Tensor*) – The input data, as a tensor of shape [B x S x H]
- **padding_mask** (*torch.Tensor*) – The input mask, as a tensor of shape [B X S]

Returns The output data, as a tensor of shape [B x H]

Return type torch.Tensor

class flambe.nn.pooling.**SumPooling**
Bases: *flambe.nn.Module*

Get the sum of the hidden state of a sequence.

forward(*self*, *data: torch.Tensor*, *padding_mask: Optional[torch.Tensor] = None*)
Performs a forward pass.

Parameters

- **data** (*torch.Tensor*) – The input data, as a tensor of shape [B x S x H]
- **padding_mask** (*torch.Tensor*) – The input mask, as a tensor of shape [B X S]

Returns The output data, as a tensor of shape [B x H]

Return type torch.Tensor

class flambe.nn.pooling.**AvgPooling**
Bases: *flambe.nn.Module*

Get the average of the hidden state of a sequence.

forward(*self*, *data: torch.Tensor*, *padding_mask: Optional[torch.Tensor] = None*)
Performs a forward pass.

Parameters

- **data** (*torch.Tensor*) – The input data, as a tensor of shape [B x S x H]
- **padding_mask** (*torch.Tensor*) – The input mask, as a tensor of shape [B X S]

Returns The output data, as a tensor of shape [B x H]

Return type torch.Tensor

flambe.nn.pooling.**_default_padding_mask**(*data: torch.Tensor*) → torch.Tensor
Builds a 1s padding mask taking into account initial 2 dimensions of input data.

Parameters **data** (*torch.Tensor*) – The input data, as a tensor of shape [B x S x H]

Returns A padding mask , as a tensor of shape [B x S]

Return type torch.Tensor

flambe.nn.pooling.**_sum_with_padding_mask**(*data: torch.Tensor*, *padding_mask: torch.Tensor*) → torch.Tensor

Applies padding_mask and performs summation over the data

Parameters

- **data** (*torch.Tensor*) – The input data, as a tensor of shape [B x S x H]
- **padding_mask** (*torch.Tensor*) – The input mask, as a tensor of shape [B X S]

Returns The result of the summation, as a tensor of shape [B x H]

Return type torch.Tensor

29.2.7 flambe.nn.rnn

Module Contents

flambe.nn.rnn.**logger**

class flambe.nn.rnn.**RNNEncoder**(*input_size: int*, *hidden_size: int*, *n_layers: int = 1*, *rnn_type: str = 'lstm'*, *dropout: float = 0*, *bidirectional: bool = False*, *layer_norm: bool = False*, *highway_bias: float = 0*, *rescale: bool = True*, *enforce_sorted: bool = False*, ***kwargs*)

Bases: *flambe.nn.module.Module*

Implements a multi-layer RNN.

This module can be used to create multi-layer RNN models, and provides a way to reduce to output of the RNN to a single hidden state by pooling the encoder states either by taking the maximum, average, or by taking the last hidden state before padding.

Padding is delt with by using torch's PackedSequence.

rnn

The rnn submodule

Type *nn.Module*

forward(*self*, *data: Tensor*, *state: Optional[Tensor] = None*, *padding_mask: Optional[Tensor] = None*)

Performs a forward pass through the network.

Parameters

- **data** (*Tensor*) – The input data, as a float tensor of shape [B x S x E]
- **state** (*Tensor*) – An optional previous state of shape [L x B x H]
- **padding_mask** (*Tensor, optional*) – The padding mask of shape [B x S]

Returns

- *Tensor* – The encoded output, as a float tensor of shape [B x S x H]
- *Tensor* – The encoded state, as a float tensor of shape [L x B x H]

class flambe.nn.rnn.**PooledRNNEncoder**(*input_size: int*, *hidden_size: int*, *n_layers: int = 1*, *rnn_type: str = 'lstm'*, *dropout: float = 0*, *bidirectional: bool = False*, *layer_norm: bool = False*, *highway_bias: float = 0*, *rescale: bool = True*, *pooling: str = 'last'*)

Bases: *flambe.nn.module.Module*

Implement an RNNEncoder with additional pooling.

This class can be used to obtan a single encoded output for an input sequence. It also ignores the state of the RNN.

forward(*self*, *data: Tensor*, *state: Optional[Tensor] = None*, *padding_mask: Optional[Tensor] = None*)

Perform a forward pass through the network.

Parameters

- **data** (*torch.Tensor*) – The input data, as a float tensor of shape [B x S x E]
- **state** (*Tensor*) – An optional previous state of shape [L x B x H]
- **padding_mask** (*Tensor, optional*) – The padding mask of shape [B x S]

Returns The encoded output, as a float tensor of shape [B x H]

Return type torch.Tensor

29.2.8 flambe.nn.sequential

Module Contents

class flambe.nn.sequential.**Sequential**(***kwargs: Dict[str, Union[Module, torch.nn.Module]]*)

Bases: *flambe.nn.Module*

Implement a Sequential module.

This class can be used in the same way as torch's nn.Sequential, with the difference that it accepts kwargs arguments.

forward(*self*, *data: torch.Tensor*)

Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – input to the model

Returns **output** – output of the model

Return type torch.Tensor

29.2.9 flambe.nn.softmax

Module Contents

class flambe.nn.softmax.**SoftmaxLayer**(*input_size: int*, *output_size: int*, *mlp_layers: int = 1*, *mlp_dropout: float = 0.0*, *mlp_hidden_activation: Optional[nn.Module] = None*, *take_log: bool = True*)

Bases: *flambe.nn.module.Module*

Implement an SoftmaxLayer module.

Can be used to form a classifier out of any encoder. Note: by default takes the log_softmax so that it can be fed to the NLLLoss module. You can disable this behavior through the *take_log* argument.

forward(*self*, *data: torch.Tensor*)

Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – input to the model of shape (*, input_size)

Returns **output** – output of the model of shape (*, output_size)

Return type torch.Tensor

29.2.10 flambe.nn.transformer

Code taken from the PyTorch source code. Slightly modified to improve the interface to the TransformerEncoder, and TransformerDecoder modules.

Module Contents

class flambe.nn.transformer.**Transformer**(*input_size*, *d_model: int = 512*, *nhead: int = 8*, *num_encoder_layers: int = 6*, *num_decoder_layers: int = 6*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*)

Bases: *flambe.nn.Module*

A Transformer model

User is able to modify the attributes as needed. The architechture is based on the paper "Attention Is All You Need". Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information Processing Systems, pages 6000-6010.

forward(*self*, *src: torch.Tensor*, *tgt: torch.Tensor*, *src_mask: Optional[torch.Tensor] = None*, *tgt_mask: Optional[torch.Tensor] = None*, *memory_mask: Optional[torch.Tensor] = None*, *src_key_padding_mask: Optional[torch.Tensor] = None*, *tgt_key_padding_mask: Optional[torch.Tensor] = None*, *memory_key_padding_mask: Optional[torch.Tensor] = None*)

Take in and process masked source/target sequences.

Parameters

- **src** (*torch.Tensor*) – the sequence to the encoder (required). shape: (N, S, E).
- **tgt** (*torch.Tensor*) – the sequence to the decoder (required). shape: (N, T, E).
- **src_mask** (*torch.Tensor, optional*) – the additive mask for the src sequence (optional). shape: (S, S).
- **tgt_mask** (*torch.Tensor, optional*) – the additive mask for the tgt sequence (optional). shape: (T, T).
- **memory_mask** (*torch.Tensor, optional*) – the additive mask for the encoder output (optional). shape: (T, S).
- **src_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for src keys per batch (optional). shape: (N, S)
- **tgt_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for tgt keys per batch (optional). shape: (N, T).
- **memory_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for memory keys per batch (optional). shape" (N, S).

Returns

- **output** (*torch.Tensor*) – The output sequence, shape: (N, T, E).
- **Note** (*[src/tgt/memory]_mask should be filled with*) – float('-inf') for the masked positions and float(0.0) else. These masks ensure that predictions for position i depend only on the unmasked positions j and are applied identically for each sequence in a batch. [src/tgt/memory]_key_padding_mask should be a ByteTensor where False values are positions that should be masked with float('-inf') and True values will be unchanged. This mask ensures that no information will be taken from position i if it is masked, and has a separate mask for each sequence in a batch.
- **Note** (*Due to the multi-head attention architecture in the*) – transformer model, the output sequence length of a transformer is same as the input sequence (i.e. target) length of the decode.

  where S is the source sequence length, T is the target sequence length, N is the batchsize, E is the feature number

class flambe.nn.transformer.**TransformerEncoder**(*input_size: int = 512*, *d_model: int = 512*, *nhead: int = 8*, *num_layers: int = 6*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*)

Bases: *flambe.nn.Module*

TransformerEncoder is a stack of N encoder layers.

forward(*self*, *src: torch.Tensor*, *memory: Optional[torch.Tensor] = None*, *mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*)

Pass the input through the endocder layers in turn.

Parameters

- **src** (*torch.Tensor*) – The sequence to the encoder (required).
- **memory** (*torch.Tensor, optional*) – Optional memory, unused by default.
- **mask** (*torch.Tensor, optional*) – The mask for the src sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – The mask for the src keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.

_reset_parameters(*self*)

Initiate parameters in the transformer model.

class flambe.nn.transformer.**TransformerDecoder**(*input_size: int*, *d_model: int*, *nhead: int*, *num_layers: int*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*)

Bases: *flambe.nn.Module*

TransformerDecoder is a stack of N decoder layers

forward(*self*, *tgt: torch.Tensor*, *memory: torch.Tensor*, *tgt_mask: Optional[torch.Tensor] = None*, *memory_mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*, *memory_key_padding_mask: Optional[torch.Tensor] = None*)

Pass the inputs (and mask) through the decoder layer in turn.

Parameters

- **tgt** (*torch.Tensor*) – The sequence to the decoder (required).
- **memory** (*torch.Tensor*) – The sequence from the last layer of the encoder (required).

- **tgt_mask** (*torch.Tensor, optional*) – The mask for the tgt sequence (optional).
- **memory_mask** (*torch.Tensor, optional*) – The mask for the memory sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – The mask for the tgt keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.
- **memory_key_padding_mask** (*torch.Tensor, optional*) – The mask for the memory keys per batch (optional).

Returns

Return type torch.Tensor

_reset_parameters(*self*)
Initiate parameters in the transformer model.

class flambe.nn.transformer.**TransformerEncoderLayer**(*d_model: int*, *nhead: int*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*)

Bases: *flambe.nn.Module*

TransformerEncoderLayer is made up of self-attn and feedforward.

This standard encoder layer is based on the paper "Attention Is All You Need". Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information Processing Systems, pages 6000-6010. Users may modify or implement in a different way during application.

forward(*self*, *src: torch.Tensor*, *memory: Optional[torch.Tensor] = None*, *src_mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*)
Pass the input through the endocder layer.

Parameters

- **src** (*torch.Tensor*) – The seqeunce to the encoder layer (required).
- **memory** (*torch.Tensor, optional*) – Optional memory from previous sequence, unused by default.
- **src_mask** (*torch.Tensor, optional*) – The mask for the src sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – The mask for the src keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.

Returns Output tensor of shape [B x S x H]

Return type torch.Tensor

class flambe.nn.transformer.**TransformerDecoderLayer**(*d_model: int*, *nhead: int*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*)

Bases: *flambe.nn.Module*

A TransformerDecoderLayer.

A TransformerDecoderLayer is made up of self-attn, multi-head-attn and feedforward network. This standard decoder layer is based on the paper "Attention Is All You Need". Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all

you need. In Advances in Neural Information Processing Systems, pages 6000-6010. Users may modify or implement in a different way during application.

forward(*self*, *tgt: torch.Tensor*, *memory: torch.Tensor*, *tgt_mask: Optional[torch.Tensor] = None*, *memory_mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*, *memory_key_padding_mask: Optional[torch.Tensor] = None*)

Pass the inputs (and mask) through the decoder layer.

Parameters

- **tgt** (*torch.Tensor*) – The sequence to the decoder layer (required).
- **memory** (*torch.Tensor*) – The sequnce from the last layer of the encoder (required).
- **tgt_mask** (*torch.Tensor, optional*) – The mask for the tgt sequence (optional).
- **memory_mask** (*torch.Tensor, optional*) – the mask for the memory sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – the mask for the tgt keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.
- **memory_key_padding_mask** (*torch.Tensor, optional*) – the mask for the memory keys per batch (optional).

Returns Output tensor of shape [T x B x H]

Return type torch.Tensor

flambe.nn.transformer.**generate_square_subsequent_mask**(*self*, *sz*)

Generate a square mask for the sequence.

The masked positions are filled with float('-inf'). Unmasked positions are filled with float(0.0).

29.2.11 flambe.nn.transformer_sru

Module Contents

class flambe.nn.transformer_sru.**TransformerSRU**(*input_size: int = 512, d_model: int = 512, nhead: int = 8, num_encoder_layers: int = 6, num_decoder_layers: int = 6, dim_feedforward: int = 2048, dropout: float = 0.1, sru_dropout: Optional[float] = None, bidirectional: bool = False, **kwargs: Dict[str, Any]*)

Bases: *flambe.nn.Module*

A Transformer with an SRU replacing the FFN.

forward(*self*, *src: torch.Tensor*, *tgt: torch.Tensor*, *src_mask: Optional[torch.Tensor] = None*, *tgt_mask: Optional[torch.Tensor] = None*, *memory_mask: Optional[torch.Tensor] = None*, *src_key_padding_mask: Optional[torch.Tensor] = None*, *tgt_key_padding_mask: Optional[torch.Tensor] = None*, *memory_key_padding_mask: Optional[torch.Tensor] = None*)

Take in and process masked source/target sequences.

Parameters

- **src** (*torch.Tensor*) – the sequence to the encoder (required). shape: (N, S, E).
- **tgt** (*torch.Tensor*) – the sequence to the decoder (required). shape: (N, T, E).

- **src_mask** (*torch.Tensor, optional*) – the additive mask for the src sequence (optional). shape: (S, S).
- **tgt_mask** (*torch.Tensor, optional*) – the additive mask for the tgt sequence (optional). shape: (T, T).
- **memory_mask** (*torch.Tensor, optional*) – the additive mask for the encoder output (optional). shape: (T, S).
- **src_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for src keys per batch (optional). shape: (N, S).
- **tgt_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for tgt keys per batch (optional). shape: (N, T).
- **memory_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for memory keys per batch (optional). shape" (N, S).

Returns

- **output** (*torch.Tensor*) – The output sequence, shape: (T, N, E).
- **Note** (*[src/tgt/memory]_mask should be filled with*) – float('-inf') for the masked positions and float(0.0) else. These masks ensure that predictions for position i depend only on the unmasked positions j and are applied identically for each sequence in a batch. [src/tgt/memory]_key_padding_mask should be a ByteTensor where False values are positions that should be masked with float('-inf') and True values will be unchanged. This mask ensures that no information will be taken from position i if it is masked, and has a separate mask for each sequence in a batch.
- **Note** (*Due to the multi-head attention architecture in the*) – transformer model, the output sequence length of a transformer is same as the input sequence (i.e. target) length of the decode.

  where S is the source sequence length, T is the target sequence length, N is the batchsize, E is the feature number

class flambe.nn.transformer_sru.**TransformerSRUEncoder**(*input_size: int = 512, d_model: int = 512, nhead: int = 8, num_layers: int = 6, dim_feedforward: int = 2048, dropout: float = 0.1, sru_dropout: Optional[float] = None, bidirectional: bool = False, **kwargs: Dict[str, Any]*)

Bases: *flambe.nn.Module*

A TransformerSRUEncoder with an SRU replacing the FFN.

forward(*self*, *src: torch.Tensor*, *state: Optional[torch.Tensor] = None*, *mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*)

Pass the input through the endocder layers in turn.

Parameters

- **src** (*torch.Tensor*) – The sequnce to the encoder (required).
- **state** (*Optional[torch.Tensor]*) – Optional state from previous sequence encoding. Only passed to the SRU (not used to perform multihead attention).
- **mask** (*torch.Tensor, optional*) – The mask for the src sequence (optional).

- **padding_mask** (*torch.Tensor, optional*) – The mask for the src keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.

_reset_parameters(*self*)
Initiate parameters in the transformer model.

class flambe.nn.transformer_sru.**TransformerSRUDecoder**(*input_size: int = 512, d_model: int = 512, nhead: int = 8, num_layers: int = 6, dim_feedforward: int = 2048, dropout: float = 0.1, sru_dropout: Optional[float] = None, **kwargs: Dict[str, Any]*)

Bases: *flambe.nn.Module*

A TransformerSRUDecoderwith an SRU replacing the FFN.

forward(*self*, *tgt: torch.Tensor*, *memory: torch.Tensor*, *state: Optional[torch.Tensor] = None*, *tgt_mask: Optional[torch.Tensor] = None*, *memory_mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*, *memory_key_padding_mask: Optional[torch.Tensor] = None*)
Pass the inputs (and mask) through the decoder layer in turn.

Parameters

- **tgt** (*torch.Tensor*) – The sequence to the decoder (required).
- **memory** (*torch.Tensor*) – The sequence from the last layer of the encoder (required).
- **state** (*Optional[torch.Tensor]*) – Optional state from previous sequence encoding. Only passed to the SRU (not used to perform multihead attention).
- **tgt_mask** (*torch.Tensor, optional*) – The mask for the tgt sequence (optional).
- **memory_mask** (*torch.Tensor, optional*) – The mask for the memory sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – The mask for the tgt keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.
- **memory_key_padding_mask** (*torch.Tensor, optional*) – The mask for the memory keys per batch (optional).

Returns

Return type torch.Tensor

_reset_parameters(*self*)
Initiate parameters in the transformer model.

class flambe.nn.transformer_sru.**TransformerSRUEncoderLayer**(*d_model: int, nhead: int, dim_feedforward: int = 2048, dropout: float = 0.1, sru_dropout: Optional[float] = None, bidirectional: bool = False, **kwargs: Dict[str, Any]*)

Bases: *flambe.nn.Module*

A TransformerSRUEncoderLayer with an SRU replacing the FFN.

forward(*self*, *src: torch.Tensor*, *state: Optional[torch.Tensor] = None*, *src_mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*)
Pass the input through the endocder layer.

Parameters

- **src** (*torch.Tensor*) – The sequence to the encoder layer (required).
- **state** (*Optional[torch.Tensor]*) – Optional state from previous sequence encoding. Only passed to the SRU (not used to perform multihead attention).
- **src_mask** (*torch.Tensor, optional*) – The mask for the src sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – The mask for the src keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.

Returns

- *torch.Tensor* – Output Tensor of shape [S x B x H]
- *torch.Tensor* – Output state of the SRU of shape [N x B x H]

class flambe.nn.transformer_sru.**TransformerSRUDecoderLayer**(*d_model: int, nhead: int, dim_feedforward: int = 2048, dropout: float = 0.1, sru_dropout: Optional[float] = None, **kwargs: Dict[str, Any]*)

Bases: *flambe.nn.Module*

A TransformerSRUDecoderLayer with an SRU replacing the FFN.

forward(*self*, *tgt: torch.Tensor*, *memory: torch.Tensor*, *state: Optional[torch.Tensor] = None*, *tgt_mask: Optional[torch.Tensor] = None*, *memory_mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*, *memory_key_padding_mask: Optional[torch.Tensor] = None*)
Pass the inputs (and mask) through the decoder layer.

Parameters

- **tgt** (*torch.Tensor*) – The sequence to the decoder layer (required).
- **memory** (*torch.Tensor*) – The sequence from the last layer of the encoder (required).
- **state** (*Optional[torch.Tensor]*) – Optional state from previous sequence encoding. Only passed to the SRU (not used to perform multihead attention).
- **tgt_mask** (*torch.Tensor, optional*) – The mask for the tgt sequence (optional).
- **memory_mask** (*torch.Tensor, optional*) – the mask for the memory sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – the mask for the tgt keys per batch (optional).

- **memory_key_padding_mask** (*torch.Tensor, optional*) – the mask for the memory keys per batch (optional).

Returns Output Tensor of shape [S x B x H]

Return type torch.Tensor

29.3 Package Contents

class flambe.nn.**Module**

Bases: *flambe.compile.Component*, torch.nn.Module

Base Flambé Module inteface.

Provides the exact same interface as Pytorch's nn.Module, but extends it with a useful set of methods to access and clip parameters, as well as gradients.

This abstraction allows users to convert their modules with a single line change, by importing from Flambé instead. Just like every Pytorch module, a forward method should be implemented.

named_trainable_params :Iterator[Tuple[str, nn.Parameter]]

Get all the named parameters with *requires_grad=True*.

Returns Iterator over the parameters and their name.

Return type Iterator[Tuple[str, nn.Parameter]]

trainable_params :Iterator[nn.Parameter]

Get all the parameters with *requires_grad=True*.

Returns Iterator over the parameters

Return type Iterator[nn.Parameter]

gradient_norm :float

Compute the average gradient norm.

Returns The current average gradient norm

Return type float

parameter_norm :float

Compute the average parameter norm.

Returns The current average parameter norm

Return type float

num_parameters(*self*, *trainable=False*)

Gets the number of parameters in the model.

Returns number of model params

Return type int

clip_params(*self*, *threshold: float*)

Clip the parameters to the given range.

Parameters **float** – Values are clipped between -threshold, threshold

clip_gradient_norm(*self*, *threshold: float*)

Clip the norm of the gradient by the given value.

Parameters **float** – Threshold to clip at

class flambe.nn.**SoftmaxLayer**(*input_size: int*, *output_size: int*, *mlp_layers: int = 1*, *mlp_dropout: float = 0.0*, *mlp_hidden_activation: Optional[nn.Module] = None*, *take_log: bool = True*)

Bases: *flambe.nn.module.Module*

Implement an SoftmaxLayer module.

Can be used to form a classifier out of any encoder. Note: by default takes the log_softmax so that it can be fed to the NLLLoss module. You can disable this behavior through the *take_log* argument.

forward(*self*, *data: torch.Tensor*)

Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – input to the model of shape (*, input_size)

Returns **output** – output of the model of shape (*, output_size)

Return type torch.Tensor

class flambe.nn.**MixtureOfSoftmax**(*input_size: int*, *output_size: int*, *k: int = 1*, *take_log: bool = True*)

Bases: *flambe.nn.module.Module*

Implement the MixtureOfSoftmax output layer.

pi

softmax layer over the different softmax

Type FullyConnected

layers

list of the k softmax layers

Type [FullyConnected]

forward(*self*, *data: Tensor*)

Implement mixture of softmax for language modeling.

Parameters **data** (*torch.Tensor*) – seq_len x batch_size x hidden_size

Returns **out** – output matrix of shape seq_len x batch_size x out_size

Return type Variable

class flambe.nn.**Embeddings**(*num_embeddings: int*, *embedding_dim: int*, *padding_idx: int = 0*, *max_norm: Optional[float] = None*, *norm_type: float = 2.0*, *scale_grad_by_freq: bool = False*, *sparse: bool = False*, *positional_encoding: bool = False*, *positional_learned: bool = False*, *positonal_max_length: int = 5000*)

Bases: *flambe.nn.module.Module*

Implement an Embeddings module.

This object replicates the usage of nn.Embedding but registers the from_pretrained classmethod to be used inside a Flambé configuration, as this does not happen automatically during the registration of PyTorch objects.

The module also adds optional positional encoding, which can either be sinusoidal or learned during training. For the non-learned positional embeddings, we use sine and cosine functions of different frequencies.

classmethod from_pretrained(*cls*, *embeddings: Tensor*, *freeze: bool = True*, *padding_idx: int = 0*, *max_norm: Optional[float] = None*, *norm_type: float = 2.0*, *scale_grad_by_freq: bool = False*, *sparse: bool = False*, *positional_encoding: bool = False*, *positional_learned: bool = False*, *positonal_max_length: int = 5000*, *positonal_embeddings: Optional[Tensor] = None*, *positonal_freeze: bool = True*)

Create an Embeddings instance from pretrained embeddings.

Parameters

- **embeddings** (*torch.Tensor*) – FloatTensor containing weights for the Embedding. First dimension is being passed to Embedding as num_embeddings, second as embedding_dim.
- **freeze** (*bool*) – If True, the tensor does not get updated in the learning process. Default: True
- **padding_idx** (*int, optional*) – Pads the output with the embedding vector at `padding_idx` (initialized to zeros) whenever it encounters the index, by default 0
- **max_norm** (*Optional[float], optional*) – If given, each embedding vector with norm larger than `max_norm` is normalized to have norm `max_norm`
- **norm_type** (*float, optional*) – The p of the p-norm to compute for the `max_norm` option. Default `2`.
- **scale_grad_by_freq** (*bool, optional*) – If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`.
- **sparse** (*bool, optional*) – If `True`, gradient w.r.t. `weight` matrix will be a sparse tensor. See Notes for more details.
- **positional_encoding** (*bool, optional*) – If True, adds positonal encoding to the token embeddings. By default, the embeddings are frozen sinusodial embeddings. To learn these during training, set positional_learned. Default `False`.
- **positional_learned** (*bool, optional*) – Learns the positional embeddings during training instead of using frozen sinusodial ones. Default `False`.
- **positonal_embeddings** (*torch.Tensor, optional*) – If given, also replaces the positonal embeddings with this matrix. The max length will be ignored and replaced by the dimension of this matrix.
- **positonal_freeze** (*bool, optional*) – Whether the positonal embeddings should be frozen

forward(*self*, *data: Tensor*)

Perform a forward pass.

Parameters **data** (*Tensor*) – The input tensor of shape [S x B]

Returns The output tensor of shape [S x B x E]

Return type Tensor

class flambe.nn.**Embedder**(*embedding: Module*, *encoder: Module*, *pooling: Optional[Module] = None*, *embedding_dropout: float = 0*, *padding_idx: Optional[int] = 0*)

Bases: *flambe.nn.module.Module*

Implements an Embedder module.

An Embedder takes as input a sequence of index tokens, and computes the corresponding embedded representations, and padding mask. The encoder may be initialized using a pretrained embedding matrix.

embeddings
The embedding module

Type *Module*

encoder
The sub-encoder that this object is wrapping

Type *Module*

pooling
An optional pooling module

Type *Module*

drop
The dropout layer

Type nn.Dropout

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor of shape [S x B]

Returns The encoded output, as a float tensor. May return a state if the encoder is an RNN and no pooling is provided.

Return type Union[Tensor, Tuple[Tensor, Tensor]]

class flambe.nn.**MLPEncoder**(*input_size: int*, *output_size: int*, *n_layers: int = 1*, *dropout: float = 0.0*, *output_activation: Optional[nn.Module] = None*, *hidden_size: Optional[int] = None*, *hidden_activation: Optional[nn.Module] = None*)
Bases: *flambe.nn.module.Module*

Implements a multi layer feed forward network.

This module can be used to create output layers, or more complex multi-layer feed forward networks.

seq
the sequence of layers and activations

Type *nn.Sequential*

forward(*self*, *data: torch.Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – input to the model of shape (batch_size, input_size)

Returns **output** – output of the model of shape (batch_size, output_size)

Return type torch.Tensor

class flambe.nn.**RNNEncoder**(*input_size: int*, *hidden_size: int*, *n_layers: int = 1*, *rnn_type: str = 'lstm'*, *dropout: float = 0*, *bidirectional: bool = False*, *layer_norm: bool = False*, *highway_bias: float = 0*, *rescale: bool = True*, *enforce_sorted: bool = False*, ***kwargs*)
Bases: *flambe.nn.module.Module*

Implements a multi-layer RNN.

This module can be used to create multi-layer RNN models, and provides a way to reduce to output of the RNN to a single hidden state by pooling the encoder states either by taking the maximum, average, or by taking the last hidden state before padding.

Padding is delt with by using torch's PackedSequence.

rnn
The rnn submodule

Type *nn.Module*

forward(*self*, *data: Tensor*, *state: Optional[Tensor] = None*, *padding_mask: Optional[Tensor] = None*)
Performs a forward pass through the network.

Parameters

- **data** (*Tensor*) – The input data, as a float tensor of shape [B x S x E]
- **state** (*Tensor*) – An optional previous state of shape [L x B x H]
- **padding_mask** (*Tensor, optional*) – The padding mask of shape [B x S]

Returns

- *Tensor* – The encoded output, as a float tensor of shape [B x S x H]
- *Tensor* – The encoded state, as a float tensor of shape [L x B x H]

class flambe.nn.**PooledRNNEncoder**(*input_size: int*, *hidden_size: int*, *n_layers: int = 1*, *rnn_type: str = 'lstm'*, *dropout: float = 0*, *bidirectional: bool = False*, *layer_norm: bool = False*, *highway_bias: float = 0*, *rescale: bool = True*, *pooling: str = 'last'*)
Bases: *flambe.nn.module.Module*

Implement an RNNEncoder with additional pooling.

This class can be used to obtan a single encoded output for an input sequence. It also ignores the state of the RNN.

forward(*self*, *data: Tensor*, *state: Optional[Tensor] = None*, *padding_mask: Optional[Tensor] = None*)
Perform a forward pass through the network.

Parameters

- **data** (*torch.Tensor*) – The input data, as a float tensor of shape [B x S x E]
- **state** (*Tensor*) – An optional previous state of shape [L x B x H]
- **padding_mask** (*Tensor, optional*) – The padding mask of shape [B x S]

Returns The encoded output, as a float tensor of shape [B x H]

Return type torch.Tensor

class flambe.nn.**CNNEncoder**(*input_channels: int, channels: List[int], conv_dim: int = 2, kernel_size: Union[int, List[Union[Tuple[int, ...], int]]] = 3, activation: nn.Module = None, pooling: nn.Module = None, dropout: float = 0, batch_norm: bool = True, stride: int = 1, padding: int = 0*)
Bases: *flambe.nn.module.Module*

Implements a multi-layer n-dimensional CNN.

This module can be used to create multi-layer CNN models.

cnn
The cnn submodule

Type *nn.Module*

forward(*self*, *data: Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – The input data, as a float tensor

Returns The encoded output, as a float tensor

Return type Union[Tensor, Tuple[Tensor, ..]]

class flambe.nn.**Sequential**(***kwargs: Dict[str, Union[Module, torch.nn.Module]]*)
Bases: *flambe.nn.Module*

Implement a Sequential module.

This class can be used in the same way as torch's nn.Sequential, with the difference that it accepts kwargs arguments.

forward(*self*, *data: torch.Tensor*)
Performs a forward pass through the network.

Parameters **data** (*torch.Tensor*) – input to the model

Returns **output** – output of the model

Return type torch.Tensor

class flambe.nn.**FirstPooling**
Bases: *flambe.nn.Module*

Get the last hidden state of a sequence.

forward(*self*, *data: torch.Tensor*, *padding_mask: Optional[torch.Tensor] = None*)
Performs a forward pass.

Parameters

- **data** (*torch.Tensor*) – The input data, as a tensor of shape [B x S x H]
- **padding_mask** (*torch.Tensor*) – The input mask, as a tensor of shape [B X S]

Returns The output data, as a tensor of shape [B x H]

Return type torch.Tensor

class flambe.nn.**LastPooling**
Bases: *flambe.nn.Module*

Get the last hidden state of a sequence.

forward(*self*, *data: torch.Tensor*, *padding_mask: Optional[torch.Tensor] = None*)
Performs a forward pass.

Parameters

- **data** (*torch.Tensor*) – The input data, as a tensor of shape [B x S x H]
- **padding_mask** (*torch.Tensor*) – The input mask, as a tensor of shape [B X S]

Returns The output data, as a tensor of shape [B x H]

Return type torch.Tensor

class flambe.nn.**SumPooling**
Bases: *flambe.nn.Module*

Get the sum of the hidden state of a sequence.

forward(*self*, *data: torch.Tensor*, *padding_mask: Optional[torch.Tensor] = None*)
Performs a forward pass.

Parameters

- **data** (*torch.Tensor*) – The input data, as a tensor of shape [B x S x H]
- **padding_mask** (*torch.Tensor*) – The input mask, as a tensor of shape [B X S]

Returns The output data, as a tensor of shape [B x H]

Return type torch.Tensor

class flambe.nn.**AvgPooling**

Bases: *flambe.nn.Module*

Get the average of the hidden state of a sequence.

forward(*self*, *data: torch.Tensor*, *padding_mask: Optional[torch.Tensor] = None*)

Performs a forward pass.

Parameters

- **data** (*torch.Tensor*) – The input data, as a tensor of shape [B x S x H]
- **padding_mask** (*torch.Tensor*) – The input mask, as a tensor of shape [B X S]

Returns The output data, as a tensor of shape [B x H]

Return type torch.Tensor

class flambe.nn.**Transformer**(*input_size*, *d_model: int = 512*, *nhead: int = 8*, *num_encoder_layers: int = 6*, *num_decoder_layers: int = 6*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*)

Bases: *flambe.nn.Module*

A Transformer model

User is able to modify the attributes as needed. The architechture is based on the paper "Attention Is All You Need". Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information Processing Systems, pages 6000-6010.

forward(*self*, *src: torch.Tensor*, *tgt: torch.Tensor*, *src_mask: Optional[torch.Tensor] = None*, *tgt_mask: Optional[torch.Tensor] = None*, *memory_mask: Optional[torch.Tensor] = None*, *src_key_padding_mask: Optional[torch.Tensor] = None*, *tgt_key_padding_mask: Optional[torch.Tensor] = None*, *memory_key_padding_mask: Optional[torch.Tensor] = None*)

Take in and process masked source/target sequences.

Parameters

- **src** (*torch.Tensor*) – the sequence to the encoder (required). shape: (N, S, E).
- **tgt** (*torch.Tensor*) – the sequence to the decoder (required). shape: (N, T, E).
- **src_mask** (*torch.Tensor, optional*) – the additive mask for the src sequence (optional). shape: (S, S).
- **tgt_mask** (*torch.Tensor, optional*) – the additive mask for the tgt sequence (optional). shape: (T, T).
- **memory_mask** (*torch.Tensor, optional*) – the additive mask for the encoder output (optional). shape: (T, S).
- **src_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for src keys per batch (optional). shape: (N, S)
- **tgt_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for tgt keys per batch (optional). shape: (N, T).

- **memory_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for memory keys per batch (optional). shape" (N, S).

Returns

- **output** (*torch.Tensor*) – The output sequence, shape: (N, T, E).
- **Note** (*[src/tgt/memory]_mask should be filled with*) – float('-inf') for the masked positions and float(0.0) else. These masks ensure that predictions for position i depend only on the unmasked positions j and are applied identically for each sequence in a batch. [src/tgt/memory]_key_padding_mask should be a ByteTensor where False values are positions that should be masked with float('-inf') and True values will be unchanged. This mask ensures that no information will be taken from position i if it is masked, and has a separate mask for each sequence in a batch.
- **Note** (*Due to the multi-head attention architecture in the*) – transformer model, the output sequence length of a transformer is same as the input sequence (i.e. target) length of the decode.

  where S is the source sequence length, T is the target sequence length, N is the batchsize, E is the feature number

class flambe.nn.**TransformerEncoder**(*input_size: int = 512*, *d_model: int = 512*, *nhead: int = 8*, *num_layers: int = 6*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*)

Bases: *flambe.nn.Module*

TransformerEncoder is a stack of N encoder layers.

forward(*self*, *src: torch.Tensor*, *memory: Optional[torch.Tensor] = None*, *mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*)

Pass the input through the endocder layers in turn.

Parameters

- **src** (*torch.Tensor*) – The sequence to the encoder (required).
- **memory** (*torch.Tensor, optional*) – Optional memory, unused by default.
- **mask** (*torch.Tensor, optional*) – The mask for the src sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – The mask for the src keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.

_reset_parameters(*self*)

Initiate parameters in the transformer model.

class flambe.nn.**TransformerDecoder**(*input_size: int*, *d_model: int*, *nhead: int*, *num_layers: int*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*)

Bases: *flambe.nn.Module*

TransformerDecoder is a stack of N decoder layers

forward(*self*, *tgt: torch.Tensor*, *memory: torch.Tensor*, *tgt_mask: Optional[torch.Tensor] = None*, *memory_mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*, *memory_key_padding_mask: Optional[torch.Tensor] = None*)

Pass the inputs (and mask) through the decoder layer in turn.

Parameters

- **tgt** (*torch.Tensor*) – The sequence to the decoder (required).
- **memory** (*torch.Tensor*) – The sequence from the last layer of the encoder (required).

- **tgt_mask** (*torch.Tensor, optional*) – The mask for the tgt sequence (optional).
- **memory_mask** (*torch.Tensor, optional*) – The mask for the memory sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – The mask for the tgt keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.
- **memory_key_padding_mask** (*torch.Tensor, optional*) – The mask for the memory keys per batch (optional).

Returns

Return type torch.Tensor

_reset_parameters(*self*)
Initiate parameters in the transformer model.

class flambe.nn.**TransformerSRU**(*input_size: int = 512, d_model: int = 512, nhead: int = 8, num_encoder_layers: int = 6, num_decoder_layers: int = 6, dim_feedforward: int = 2048, dropout: float = 0.1, sru_dropout: Optional[float] = None, bidirectional: bool = False, **kwargs: Dict[str, Any]*)

Bases: *flambe.nn.Module*

A Transformer with an SRU replacing the FFN.

forward(*self*, *src: torch.Tensor*, *tgt: torch.Tensor*, *src_mask: Optional[torch.Tensor] = None*, *tgt_mask: Optional[torch.Tensor] = None*, *memory_mask: Optional[torch.Tensor] = None*, *src_key_padding_mask: Optional[torch.Tensor] = None*, *tgt_key_padding_mask: Optional[torch.Tensor] = None*, *memory_key_padding_mask: Optional[torch.Tensor] = None*)
Take in and process masked source/target sequences.

Parameters

- **src** (*torch.Tensor*) – the sequence to the encoder (required). shape: (N, S, E).
- **tgt** (*torch.Tensor*) – the sequence to the decoder (required). shape: (N, T, E).
- **src_mask** (*torch.Tensor, optional*) – the additive mask for the src sequence (optional). shape: (S, S).
- **tgt_mask** (*torch.Tensor, optional*) – the additive mask for the tgt sequence (optional). shape: (T, T).
- **memory_mask** (*torch.Tensor, optional*) – the additive mask for the encoder output (optional). shape: (T, S).
- **src_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for src keys per batch (optional). shape: (N, S).
- **tgt_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for tgt keys per batch (optional). shape: (N, T).
- **memory_key_padding_mask** (*torch.Tensor, optional*) – the ByteTensor mask for memory keys per batch (optional). shape" (N, S).

Returns

- **output** (*torch.Tensor*) – The output sequence, shape: (T, N, E).
- **Note** (*[src/tgt/memory]_mask should be filled with*) – float('-inf') for the masked positions and float(0.0) else. These masks ensure that predictions for position i depend only

on the unmasked positions j and are applied identically for each sequence in a batch. [src/tgt/memory]_key_padding_mask should be a ByteTensor where False values are positions that should be masked with float('-inf') and True values will be unchanged. This mask ensures that no information will be taken from position i if it is masked, and has a separate mask for each sequence in a batch.

- **Note** (*Due to the multi-head attention architecture in the*) – transformer model, the output sequence length of a transformer is same as the input sequence (i.e. target) length of the decode.

  where S is the source sequence length, T is the target sequence length, N is the batchsize, E is the feature number

class flambe.nn.**TransformerSRUEncoder**(*input_size: int = 512, d_model: int = 512, nhead: int = 8, num_layers: int = 6, dim_feedforward: int = 2048, dropout: float = 0.1, sru_dropout: Optional[float] = None, bidirectional: bool = False, **kwargs: Dict[str, Any]*)

Bases: *flambe.nn.Module*

A TransformerSRUEncoder with an SRU replacing the FFN.

forward(*self*, *src: torch.Tensor*, *state: Optional[torch.Tensor] = None*, *mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*)

Pass the input through the endocder layers in turn.

Parameters

- **src** (*torch.Tensor*) – The sequnce to the encoder (required).
- **state** (*Optional[torch.Tensor]*) – Optional state from previous sequence encoding. Only passed to the SRU (not used to perform multihead attention).
- **mask** (*torch.Tensor, optional*) – The mask for the src sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – The mask for the src keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.

_reset_parameters(*self*)

Initiate parameters in the transformer model.

class flambe.nn.**TransformerSRUDecoder**(*input_size: int = 512, d_model: int = 512, nhead: int = 8, num_layers: int = 6, dim_feedforward: int = 2048, dropout: float = 0.1, sru_dropout: Optional[float] = None, **kwargs: Dict[str, Any]*)

Bases: *flambe.nn.Module*

A TransformerSRUDecoderwith an SRU replacing the FFN.

forward(*self*, *tgt: torch.Tensor*, *memory: torch.Tensor*, *state: Optional[torch.Tensor] = None*, *tgt_mask: Optional[torch.Tensor] = None*, *memory_mask: Optional[torch.Tensor] = None*, *padding_mask: Optional[torch.Tensor] = None*, *memory_key_padding_mask: Optional[torch.Tensor] = None*)

Pass the inputs (and mask) through the decoder layer in turn.

Parameters

- **tgt** (*torch.Tensor*) – The sequence to the decoder (required).
- **memory** (*torch.Tensor*) – The sequence from the last layer of the encoder (required).
- **state** (*Optional[torch.Tensor]*) – Optional state from previous sequence encoding. Only passed to the SRU (not used to perform multihead attention).

- **tgt_mask** (*torch.Tensor, optional*) – The mask for the tgt sequence (optional).
- **memory_mask** (*torch.Tensor, optional*) – The mask for the memory sequence (optional).
- **padding_mask** (*torch.Tensor, optional*) – The mask for the tgt keys per batch (optional). Should be True for tokens to leave untouched, and False for padding tokens.
- **memory_key_padding_mask** (*torch.Tensor, optional*) – The mask for the memory keys per batch (optional).

Returns

Return type torch.Tensor

_reset_parameters(*self*)

Initiate parameters in the transformer model.

CHAPTER 30

flambe.runnable

30.1 Submodules

30.1.1 flambe.runnable.cluster_runnable

Module Contents

class flambe.runnable.cluster_runnable.**ClusterRunnable**(*user_provider: Callable[[], str] = None*, *env: Optional[RemoteEnvironment] = None*, ***kwargs*)

Bases: *flambe.runnable.Runnable*

Base class for all runnables that are able to run on cluster.

This type of Runnables must include logic in the 'run' method to deal with the fact that they could be running in a distributed cluster of machines.

To provide useful information about the cluster, a RemoteEnvironment object will be injected when running remotely.

config
The secrets that the user provides. For example, 'config["AWS"]["ACCESS_KEY"]'

Type configparser.ConfigParser

env
The remote environment has information about the cluster where this ClusterRunnable will be running. IMPORTANT: this object will be available only when the ClusterRunnable is running remotely.

Type *RemoteEnvironment*

user_provider
The logic for specifying the user triggering this Runnable. If not passed, by default it will pick the computer's user.

Type Callable[[], str]

setup (*self, cluster: Cluster, extensions: Dict[str, str], force: bool, **kwargs*)
Setup the cluster.

Parameters

- **cluster** (*Cluster*) – The cluster where this Runnable will be running
- **extensions** (*Dict[str, str]*) – The ClusterRunnable extensions
- **force** (*bool*) – The force value provided to Flambe

setup_inject_env (*self, cluster: Cluster, extensions: Dict[str, str], force: bool, **kwargs*)
Call setup and inject the RemoteEnvironment

Parameters

- **cluster** (*Cluster*) – The cluster where this Runnable will be running
- **extensions** (*Dict[str, str]*) – The ClusterRunnable extensions
- **force** (*bool*) – The force value provided to Flambe

set_serializable_attr (*self*, *attr*, *value*)
Set an attribute while keep supporting serializaton.

30.1.2 flambe.runnable.context

Module Contents

flambe.runnable.context.**logger**

class flambe.runnable.context.**SafeExecutionContext** (*yaml_file: str*)
Context manager handling the experiment's creation and execution.

Parameters **yaml_file** (*str*) – The experiment filename

__enter__ (*self*)
A SafeExecutionContext should be used as a context manager to handle all possible errors in a clear way.

Examples

```
>>> with SafeExecutionContext(...) as ex:
>>>     ...
```

__exit__ (*self, exc_type: Optional[Type[BaseException]], exc_value: Optional[BaseException], tb: Optional[TracebackType]*)
Exit method for the context manager.

This method will catch any exception, and return True. This means that all exceptions produced in a SafeExecutionContext (used with the context manager) will not continue to raise.

Returns True, as an exception should not continue to raise.

Return type Optional[bool]

preprocess (*self*, *secrets: Optional[str] = None*, *download_ext: bool = True*, *install_ext: bool = False*, *import_ext: bool = True*, *check_tags: bool = True*, ***kwargs*)
Preprocess the runnable file.

Looks for syntax errors, import errors, etc. Also injects the secrets into the runnables.

If this method runs and ends without exceptions, then the experiment is ok to be run. If this method raises an Error and the SafeExecutionContext is used as context manager, then the __exit__ method will be executed.

Parameters

- **secrets** (*Optional[str]*) – Optional path to the secrets file
- **install_ext** (*bool*) – Whether to install the extensions or not. This process also downloads the remote extensions. Defaults to False
- **install_ext** – Whether to import the extensions or not. Defaults to True.
- **check_tags** (*bool*) – Whether to check that all tags are valid. Defaults to True.

Returns A tuple containing the compiled Runnable and a dict containing the extensions the Runnable uses.

Return type Tuple[*Runnable*, Dict[str, str]]

Raises `Exception` – Depending on the error.

first_parse(*self*)

Check if valid YAML file and also load config

In this first parse the runnable does not get compiled because it could be a custom Runnable, so it needs the extensions to be imported first.

check_tags(*self*, *content: str*)

Check that all the tags are valid.

Parameters **content** (*str*) – The content of the YAML file

Raises `TagError`

compile_runnable(*self*, *content: str*)

Compiles and returns the Runnable.

IMPORTANT: This method should run after all extensions were registered.

Parameters **content** (*str*) – The runnable, as a YAML string

Returns The compiled experiment.

Return type *Runnable*

30.1.3 flambe.runnable.environment

Module Contents

class flambe.runnable.environment.**RemoteEnvironment**(*key: str, orchestrator_ip: str, factories_ips: List[str], user: str, local_user: str, public_orchestrator_ip: Optional[str] = None, public_factories_ips: Optional[List[str]] = None, **kwargs*)

Bases: *flambe.compile.Registrable*

This objects contains information about the cluster

This object will be available on the remote execution of the ClusterRunnable (as an attribute).

IMPORTANT: this object needs to be serializable, hence it Needs to be created using 'compile' method.

key
The key that communicates the cluster

Type str

orchestrator_ip
The orchestrator visible IP for the factories (usually the private IP)

Type str

factories_ips
The list of factories IPs visible for other factories and orchestrator (usually private IPs)

Type List[str]

user
The username of all machines. This implementations assumes same username for all machines

Type str

local_user
The username of the local process that launched the cluster

Type str

public_orchestrator_ip
The public orchestrator IP, if available.

Type Optional[str]

public_factories_ips
The public factories IPs, if available.

Type Optional[List[str]]

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)
Use representer to create yaml representation of node

classmethod from_yaml(*cls*, *constructor: Any*, *node: Any*, *factory_name: str*)
Use constructor to create an instance of cls

30.1.4 flambe.runnable.error

Module Contents

exception flambe.runnable.error.**ProtocolError**(*message: str*)
Bases: Exception

__repr__(*self*)
Override output message to show ProtocolError

Returns The output message

Return type str

exception flambe.runnable.error.**LinkError**(*block_id: str*, *target_block_id: str*)
Bases: *flambe.runnable.error.ProtocolError*

exception flambe.runnable.error.**SearchComponentError**(*block_id: str*)
Bases: *flambe.runnable.error.ProtocolError*

exception flambe.runnable.error.**UnsuccessfulRunnableError**
Bases: RuntimeError

exception flambe.runnable.error.**RunnableFileError**
Bases: Exception

exception flambe.runnable.error.**ResourceError**
Bases: *flambe.runnable.error.RunnableFileError*

exception flambe.runnable.error.**NonExistentResourceError**
Bases: *flambe.runnable.error.RunnableFileError*

exception flambe.runnable.error.**ExistentResourceError**
Bases: *flambe.runnable.error.RunnableFileError*

exception flambe.runnable.error.**ParsingRunnableError**
Bases: *flambe.runnable.error.RunnableFileError*

exception flambe.runnable.error.**TagError**
Bases: *flambe.runnable.error.RunnableFileError*

exception flambe.runnable.error.**MissingSecretsError**
Bases: Exception

30.1.5 flambe.runnable.runnable

Module Contents

class flambe.runnable.runnable.**Runnable**(*user_provider: Callable[[], str] = None*, ***kwargs*)
Bases: *flambe.compile.MappedRegistrable*

Base class for all runnables.

A runnable contains a single run method that needs to be implemented. It must contain all the logic for the runnable.

Each runnable has also access to the secrets the user provides.

Examples of Runnables: Experiment, Cluster

config
The secrets that the user provides. For example, 'config["AWS"]["ACCESS_KEY"]'

Type configparser.ConfigParser

extensions
The extensions used for this runnable.

Type Dict[str, str]

content
This attribute will hold the YAML representation of the Runnable.

Type Optional[str]

user_provider
The logic for specifying the user triggering this Runnable. If not passed, by default it will pick the computer's user.

Type Callable[[], str]

inject_content (*self*, *content: str*)
Inject the original YAML string that was used to generate this Runnable instance.

Parameters **content** (*str*) – The YAML, as a string

inject_secrets (*self*, *secrets: str*)
Inject the secrets once the Runnable was created.

Parameters **secrets** (*str*) – The filepath to the secrets

inject_extensions (*self, extensions: Dict[str, str]*)
Inject extensions to the Runnable

Parameters **extensions** (*Dict[str, str]*) – The extensions

run (*self*, ***kwargs*)
Run the runnable.

Each implementation will implement its own logic, with the parameters it needs.

parse (*self*)
Parse the runnable to determine if it's able to run. :raises: ParsingExperimentError – In case a parsing error is found.

30.1.6 flambe.runnable.utils

Module Contents

flambe.runnable.utils.**DEFAULT_USER_PROVIDER**

flambe.runnable.utils.**_contains_path** (*nested_dict*) → bool
Whether the nested dict contains any value that could be a path.

Parameters **nested_dict** – The nested dict to evaluate

Returns

Return type bool

flambe.runnable.utils.**is_dev_mode** () → bool
Detects if flambe was installed in editable mode.

For more information: https://pip.pypa.io/en/latest/reference/pip_install/#editable-installs

Returns

Return type bool

flambe.runnable.utils.**get_flambe_repo_location** () → str
Return where flambe repository is located

Returns The local path where flambe is located

Return type str

Raises ValueError – If flambe was not installed in editable mode

flambe.runnable.utils.**get_commit_hash** () → str
Get the commit hash of the current flambe development package.

This will only work if flambe was install from github in dev mode.

Returns The commit hash

Return type str

Raises `Exception` – In case flambe was not installed in dev mode.

`flambe.runnable.utils.`**`rsync_hosts`**(*orch_ip: str, factories_ips: List[str], user: str, folder: str, key: str, exclude: List[str]*) → None

Rsync the hosts in the cluster.

IMPORTANT: this method is intended to be run in the cluster.

Parameters

- **`orch_ip`** (*`str`*) – The Orchestrator's IP that is visible by the factories (usually the private IP)
- **`factories_ips`** (*`List[str]`*) – The factories IPs that are visible by the Orchestrator (usually the private IPs)
- **`user`** (*`str`*) – The username of all machines. IMPORTANT: only machines with same username are supported
- **`key`** (*`str`*) – The key that communicate all machines
- **`exclude`** (*`List[str]`*) – A list of files to be excluded in the rsync

30.2 Package Contents

`class` `flambe.runnable.`**`Runnable`**(*user_provider: Callable[[], str] = None, **kwargs*)

Bases: *`flambe.compile.MappedRegistrable`*

Base class for all runnables.

A runnable contains a single run method that needs to be implemented. It must contain all the logic for the runnable.

Each runnable has also access to the secrets the user provides.

Examples of Runnables: Experiment, Cluster

`config`

The secrets that the user provides. For example, 'config["AWS"]["ACCESS_KEY"]'

Type configparser.ConfigParser

`extensions`

The extensions used for this runnable.

Type Dict[str, str]

`content`

This attribute will hold the YAML representation of the Runnable.

Type Optional[str]

`user_provider`

The logic for specifying the user triggering this Runnable. If not passed, by default it will pick the computer's user.

Type Callable[[], str]

`inject_content`(*self*, *content: str*)

Inject the original YAML string that was used to generate this Runnable instance.

Parameters **`content`** (*`str`*) – The YAML, as a string

inject_secrets (*self*, *secrets: str*)
Inject the secrets once the Runnable was created.

Parameters **secrets** (*str*) – The filepath to the secrets

inject_extensions (*self, extensions: Dict[str, str]*)
Inject extensions to the Runnable

Parameters **extensions** (*Dict[str, str]*) – The extensions

run (*self*, ***kwargs*)
Run the runnable.

Each implementation will implement its own logic, with the parameters it needs.

parse (*self*)
Parse the runnable to determine if it's able to run. :raises: ParsingExperimentError – In case a parsing error is found.

class flambe.runnable.**ClusterRunnable**(*user_provider: Callable[[], str] = None*, *env: Optional[RemoteEnvironment] = None*, ***kwargs*)
Bases: *flambe.runnable.Runnable*

Base class for all runnables that are able to run on cluster.

This type of Runnables must include logic in the 'run' method to deal with the fact that they could be running in a distributed cluster of machines.

To provide useful information about the cluster, a RemoteEnvironment object will be injected when running remotely.

config
The secrets that the user provides. For example, 'config["AWS"]["ACCESS_KEY"]'

Type configparser.ConfigParser

env
The remote environment has information about the cluster where this ClusterRunnable will be running. IMPORTANT: this object will be available only when the ClusterRunnable is running remotely.

Type *RemoteEnvironment*

user_provider
The logic for specifying the user triggering this Runnable. If not passed, by default it will pick the computer's user.

Type Callable[[], str]

setup (*self, cluster: Cluster, extensions: Dict[str, str], force: bool, **kwargs*)
Setup the cluster.

Parameters

- **cluster** (*Cluster*) – The cluster where this Runnable will be running
- **extensions** (*Dict[str, str]*) – The ClusterRunnable extensions
- **force** (*bool*) – The force value provided to Flambe

setup_inject_env (*self, cluster: Cluster, extensions: Dict[str, str], force: bool, **kwargs*)
Call setup and inject the RemoteEnvironment

Parameters

- **cluster** (*Cluster*) – The cluster where this Runnable will be running
- **extensions** (*Dict[str, str]*) – The ClusterRunnable extensions

- **force** (*bool*) – The force value provided to Flambe

set_serializable_attr(*self*, *attr*, *value*)
Set an attribute while keep supporting serializaton.

class flambe.runnable.**SafeExecutionContext**(*yaml_file: str*)
Context manager handling the experiment's creation and execution.

Parameters **yaml_file** (*str*) – The experiment filename

__enter__(*self*)
A SafeExecutionContext should be used as a context manager to handle all possible errors in a clear way.

Examples

```
>>> with SafeExecutionContext(...) as ex:
>>>     ...
```

__exit__(*self, exc_type: Optional[Type[BaseException]], exc_value: Optional[BaseException], tb: Optional[TracebackType]*)
Exit method for the context manager.

This method will catch any exception, and return True. This means that all exceptions produced in a SafeExecutionContext (used with the context manager) will not continue to raise.

Returns True, as an exception should not continue to raise.

Return type Optional[bool]

preprocess(*self*, *secrets: Optional[str] = None*, *download_ext: bool = True*, *install_ext: bool = False*, *import_ext: bool = True*, *check_tags: bool = True*, ***kwargs*)
Preprocess the runnable file.

Looks for syntax errors, import errors, etc. Also injects the secrets into the runnables.

If this method runs and ends without exceptions, then the experiment is ok to be run. If this method raises an Error and the SafeExecutionContext is used as context manager, then the __exit__ method will be executed.

Parameters

- **secrets** (*Optional[str]*) – Optional path to the secrets file
- **install_ext** (*bool*) – Whether to install the extensions or not. This process also downloads the remote extensions. Defaults to False
- **install_ext** – Whether to import the extensions or not. Defaults to True.
- **check_tags** (*bool*) – Whether to check that all tags are valid. Defaults to True.

Returns A tuple containing the compiled Runnable and a dict containing the extensions the Runnable uses.

Return type Tuple[*Runnable*, Dict[str, str]]

Raises Exception – Depending on the error.

first_parse(*self*)
Check if valid YAML file and also load config

In this first parse the runnable does not get compiled because it could be a custom Runnable, so it needs the extensions to be imported first.

check_tags(*self*, *content: str*)
Check that all the tags are valid.

Parameters **content** (*str*) – The content of the YAML file

Raises TagError

compile_runnable(*self*, *content: str*)
Compiles and returns the Runnable.

IMPORTANT: This method should run after all extensions were registered.

Parameters **content** (*str*) – The runnable, as a YAML string

Returns The compiled experiment.

Return type *Runnable*

class flambe.runnable.**RemoteEnvironment**(*key: str, orchestrator_ip: str, factories_ips: List[str], user: str, local_user: str, public_orchestrator_ip: Optional[str] = None, public_factories_ips: Optional[List[str]] = None, **kwargs*)

Bases: *flambe.compile.Registrable*

This objects contains information about the cluster

This object will be available on the remote execution of the ClusterRunnable (as an attribute).

IMPORTANT: this object needs to be serializable, hence it Needs to be created using 'compile' method.

key
The key that communicates the cluster

Type str

orchestrator_ip
The orchestrator visible IP for the factories (usually the private IP)

Type str

factories_ips
The list of factories IPs visible for other factories and orchestrator (usually private IPs)

Type List[str]

user
The username of all machines. This implementations assumes same username for all machines

Type str

local_user
The username of the local process that launched the cluster

Type str

public_orchestrator_ip
The public orchestrator IP, if available.

Type Optional[str]

public_factories_ips
The public factories IPs, if available.

Type Optional[List[str]]

classmethod to_yaml(*cls*, *representer: Any*, *node: Any*, *tag: str*)
Use representer to create yaml representation of node

classmethod from_yaml(*cls*, *constructor: Any*, *node: Any*, *factory_name: str*)
Use constructor to create an instance of cls

CHAPTER 31

flambe.runner

31.1 Submodules

31.1.1 flambe.runner.report_site_run

Script to run the report web site

It takes the *app* defined in *flambe.remote.webapp.app* and runs it.

Module Contents

flambe.runner.report_site_run.**logger**

flambe.runner.report_site_run.**launch_tensorboard**(*tracking_address*) → Optional[str]

flambe.runner.report_site_run.**parser**

31.1.2 flambe.runner.run

Local run script for flambe.

Module Contents

flambe.runner.run.**TORCH_TAG_PREFIX = torch**

flambe.runner.run.**TUNE_TAG_PREFIX = tune**

flambe.runner.run.**main**(*args: argparse.Namespace*) → None
Execute command based on given config

flambe.runner.run.**parser**

31.1.3 flambe.runner.utils

Module Contents

flambe.runner.utils.**logger**

flambe.runner.utils.**MB**

flambe.runner.utils.**WARN_LIMIT_MB = 100**

flambe.runner.utils.**get_size_MB**(*path: str*) → float
Return the size of a file/folder in MB.

Parameters **path** (*str*) – The path to the folder or file

Returns The size in MB

Return type float

flambe.runner.utils.**check_system_reqs**() → None
Run system checks and prepare the system before a run.

This method should:

- Create folders, files that are needed for flambe
- Raise errors in case requirements are not met. This should

run under the SafeExecutionContext, so errors will be handled * Warn the user in case something needs attention.

CHAPTER 32

flambe.sampler

32.1 Submodules

32.1.1 flambe.sampler.base

Module Contents

flambe.sampler.base.**_bfs**(*obs: List*, *obs_idx: int*) → Tuple[Dict[int, List], Set[Tuple[int, ...]]]

Given a single *obs*, itself a nested list, run BFS.

This function enumerates:

1. The lengths of each of the intermediary lists, by depth
2. All paths to the child nodes

Parameters

- **obs** (*List*) – A nested list of lists of arbitrary depth, with the child nodes, i.e. deepest list elements, as ‘torch.Tensor‘s
- **obs_idx** (*int*) – The index of *obs* in the batch.

Returns

- *Set[Tuple[int]]* – A set of all distinct paths to all children
- *Dict[int, List[int]]* – A map containing the lengths of all intermediary lists, by depth

flambe.sampler.base.**_batch_from_nested_col**(*col: Tuple*, *pad: int*) → torch.Tensor

Compose a batch padded to the max-size along each dimension.

Parameters **col** (*List*) – A nested list of lists of arbitrary depth, with the child nodes, i.e. deepest list elements, as ‘torch.Tensor‘s

For example, a *col* might be:

[[torch.Tensor([1, 2]), torch.Tensor([3, 4, 5])], [torch.Tensor([5, 6, 7]), torch.Tensor([4, 5]),
torch.Tensor([5, 6, 7, 8])]
]

Level 1 sizes: [2, 3] Level 2 sizes: [2, 3]; [3, 2, 4]

The max-sizes along each dimension are:

- Dim 1: 3
- Dim 2: 4

As such, since this column contains 2 elements, with max-sizes 3 and 4 along the nested dimensions, our resulting batch would have size (4, 3, 2), and the padded 'Tensor's would be inserted at their respective locations.

Returns A (n+1)-dimensional torch.Tensor, where n is the nesting depth, padded to the max-size along each dimension

Return type torch.Tensor

`flambe.sampler.base.`**`collate_fn`**(*data: List[Tuple[torch.Tensor, ...]], pad: int*) → Tuple[torch.Tensor, ...]

Turn a list of examples into a mini-batch.

Handles padding on the fly on simple sequences, as well as nested sequences.

Parameters

- **data** (*List[Tuple[torch.Tensor, ..]]*) – The list of sampled examples. Each example is a tuple, each dimension representing a column from the original dataset
- **pad** (*int*) – The padding index

Returns The output batch of tensors

Return type Tuple[torch.Tensor, ..]

class `flambe.sampler.base.`**`BaseSampler`**(*batch_size: int = 64, shuffle: bool = True, pad_index: Union[int, Sequence[int]] = 0, n_workers: int = 0, pin_memory: bool = False, seed: Optional[int] = None, downsample: Optional[float] = None, downsample_seed: Optional[int] = None, drop_last: bool = False*)

Bases: *flambe.sampler.sampler.Sampler*

Implements a BaseSampler object.

This is the most basic implementation of a sampler. It uses Pytorch's DataLoader object internally, and offers the possiblity to override the sampling of the examples and how to from a batch from them.

`sample`(*self, data: Sequence[Sequence[torch.Tensor]], n_epochs: int = 1*)

Sample from the list of features and yields batches.

Parameters

- **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from
- **n_epochs** (*int, optional*) – The number of epochs to run in the output iterator. Use -1 to run infinitely.

Yields *Iterator[Tuple[Tensor]]* – A batch of data, as a tuple of Tensors

length (*self, data: Sequence[Sequence[torch.Tensor]]*)
Return the number of batches in the sampler.

Parameters **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from

Returns The number of batches that would be created per epoch

Return type int

32.1.2 flambe.sampler.episodic

Module Contents

class flambe.sampler.episodic.**EpisodicSampler**(*n_support: int, n_query: int, n_episodes: int, n_classes: int = None, pad_index: int = 0, balance_query: bool = False*)

Bases: *flambe.sampler.Sampler*

Implement an EpisodicSample object.

Currently only supports sequence inputs.

sample (*self, data: Sequence[Sequence[torch.Tensor]], n_epochs: int = 1*)
Sample from the list of features and yields batches.

Parameters

- **data** (*Sequence[Sequence[torch.Tensor, torch.Tensor]]*) – The input data as a list of (source, target) pairs
- **n_epochs** (*int, optional*) – The number of epochs to run in the output iterator. For this object, the total number of batches will be (n_episodes * n_epochs)

Yields *Iterator[Tuple[Tensor, Tensor, Tensor, Tensor]]* – In order: the query_source, the query_target the support_source, and the support_target tensors. For sequences, the batch is used as first dimension.

length (*self, data: Sequence[Sequence[torch.Tensor]]*)
Return the number of batches in the sampler.

Parameters **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from

Returns The number of batches that would be created per epoch

Return type int

32.1.3 flambe.sampler.sampler

Module Contents

class flambe.sampler.sampler.**Sampler**
Bases: *flambe.compile.Component*

Base Sampler interface.

Objects implementing this interface should implement two methods:

- *sample*: takes a set of data and returns an iterator

- ***lenght*: takes a set of data and return the length of the** iterator that would be given by the sample method

Sampler objects are used inside the Trainer to provide the data to the models. Note that pushing the data to the appropriate device is usually done inside the Trainer.

sample (*self, data: Sequence[Sequence[torch.Tensor]], n_epochs: int = 1*)
Sample from the list of features and yields batches.

Parameters

- **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from
- **n_epochs** (*int, optional*) – The number of epochs to run in the output iterator.

Yields *Iterator[Tuple[Tensor]]* – A batch of data, as a tuple of Tensors

length (*self, data: Sequence[Sequence[torch.Tensor]]*)
Return the number of batches in the sampler.

Parameters **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from

Returns The number of batches that would be created per epoch

Return type int

32.2 Package Contents

class flambe.sampler.**Sampler**
Bases: *flambe.compile.Component*

Base Sampler interface.

Objects implementing this interface should implement two methods:

- *sample*: takes a set of data and returns an iterator
- ***lenght*: takes a set of data and return the length of the** iterator that would be given by the sample method

Sampler objects are used inside the Trainer to provide the data to the models. Note that pushing the data to the appropriate device is usually done inside the Trainer.

sample (*self, data: Sequence[Sequence[torch.Tensor]], n_epochs: int = 1*)
Sample from the list of features and yields batches.

Parameters

- **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from
- **n_epochs** (*int, optional*) – The number of epochs to run in the output iterator.

Yields *Iterator[Tuple[Tensor]]* – A batch of data, as a tuple of Tensors

length (*self, data: Sequence[Sequence[torch.Tensor]]*)
Return the number of batches in the sampler.

Parameters **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from

Returns The number of batches that would be created per epoch

Return type int

class flambe.sampler.**BaseSampler**(*batch_size: int = 64*, *shuffle: bool = True*, *pad_index: Union[int, Sequence[int]] = 0*, *n_workers: int = 0*, *pin_memory: bool = False*, *seed: Optional[int] = None*, *downsample: Optional[float] = None*, *downsample_seed: Optional[int] = None*, *drop_last: bool = False*)

Bases: *flambe.sampler.sampler.Sampler*

Implements a BaseSampler object.

This is the most basic implementation of a sampler. It uses Pytorch's DataLoader object internally, and offers the possiblity to override the sampling of the examples and how to from a batch from them.

sample(*self, data: Sequence[Sequence[torch.Tensor]], n_epochs: int = 1*)

Sample from the list of features and yields batches.

Parameters

- **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from
- **n_epochs** (*int, optional*) – The number of epochs to run in the output iterator. Use -1 to run infinitely.

Yields *Iterator[Tuple[Tensor]]* – A batch of data, as a tuple of Tensors

length(*self, data: Sequence[Sequence[torch.Tensor]]*)

Return the number of batches in the sampler.

Parameters **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from

Returns The number of batches that would be created per epoch

Return type int

class flambe.sampler.**EpisodicSampler**(*n_support: int*, *n_query: int*, *n_episodes: int*, *n_classes: int = None*, *pad_index: int = 0*, *balance_query: bool = False*)

Bases: *flambe.sampler.Sampler*

Implement an EpisodicSample object.

Currently only supports sequence inputs.

sample(*self, data: Sequence[Sequence[torch.Tensor]], n_epochs: int = 1*)

Sample from the list of features and yields batches.

Parameters

- **data** (*Sequence[Sequence[torch.Tensor, torch.Tensor]]*) – The input data as a list of (source, target) pairs
- **n_epochs** (*int, optional*) – The number of epochs to run in the output iterator. For this object, the total number of batches will be (n_episodes * n_epochs)

Yields *Iterator[Tuple[Tensor, Tensor, Tensor, Tensor]]* – In order: the query_source, the query_target the support_source, and the support_target tensors. For sequences, the batch is used as first dimension.

length(*self, data: Sequence[Sequence[torch.Tensor]]*)

Return the number of batches in the sampler.

Parameters **data** (*Sequence[Sequence[torch.Tensor, ..]]*) – The input data to sample from

Returns The number of batches that would be created per epoch

Return type int

CHAPTER 33

flambe.tokenizer

33.1 Submodules

33.1.1 flambe.tokenizer.char

Module Contents

class flambe.tokenizer.char.**CharTokenizer**
Bases: *flambe.tokenizer.Tokenizer*

Implement a character level tokenizer.

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output charachter tokens, as a list of strings

Return type List[str]

33.1.2 flambe.tokenizer.label

Module Contents

class flambe.tokenizer.label.**LabelTokenizer**(*multilabel_sep: Optional[str] = None*)
Bases: *flambe.tokenizer.Tokenizer*

Base label tokenizer.

This object tokenizes string labels into a list of a single or multiple elements, depending on the provided separator.

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output tokens, as a list of strings

Return type List[str]

33.1.3 flambe.tokenizer.subword

Module Contents

class flambe.tokenizer.subword.**BPETokenizer**(*codes_path: str*)
Bases: *flambe.tokenizer.Tokenizer*

Implement a subword level tokenizer using byte pair encoding. Tokenization is done using fastBPE (https://github.com/glample/fastBPE) and requires a fastBPE codes file.

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output subword tokens, as a list of strings

Return type List[str]

33.1.4 flambe.tokenizer.tokenizer

Module Contents

class flambe.tokenizer.tokenizer.**Tokenizer**
Bases: flambe.Component

Base interface to a Tokenizer object.

Tokenizers implement the *tokenize* method, which takes a string as input and produces a list of strings as output.

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output tokens, as a list of strings

Return type List[str]

__call__(*self*, *example: str*)
Make a tokenizer callable.

33.1.5 flambe.tokenizer.word

Module Contents

class flambe.tokenizer.word.**WordTokenizer**
Bases: *flambe.tokenizer.Tokenizer*

Implement a word level tokenizer using nltk.tokenize.word_tokenize

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output word tokens, as a list of strings

Return type List[str]

class flambe.tokenizer.word.**NLTKWordTokenizer**(***kwargs*)
Bases: *flambe.tokenizer.Tokenizer*

Implement a word level tokenizer using nltk.tokenize.word_tokenize

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output word tokens, as a list of strings

Return type List[str]

class flambe.tokenizer.word.**NGramsTokenizer**(*ngrams: Union[int, List[int]] = 1*, *exclude_stopwords: bool = False*, *stop_words: Optional[List] = None*)
Bases: *flambe.tokenizer.Tokenizer*

Implement a n-gram tokenizer

Examples

```
>>> t = NGramsTokenizer(ngrams=2).tokenize("hi how are you?")
['hi, how', 'how are', 'are you?']
```

```
>>> t = NGramsTokenizer(ngrams=[1,2]).tokenize("hi how are you?")
['hi,', 'how', 'are', 'you?', 'hi, how', 'how are', 'are you?']
```

Parameters

- **ngrams** (*Union[int, List[int]]*) – An int or a list of ints. If it's a list of ints, all n-grams (for each int) will be considered in the tokenizer.
- **exclude_stopwords** (*bool*) – Whether to exlude stopword or not. See the related param stop_words
- **stop_words** (*Optional[List]*) – List of stop words to exclude when exclude_stopwords is True. If None set to nltk.corpus.stopwords.

static _tokenize(*example: str*, *n: int*)
Tokenize an input example using ngrams.

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string.

Returns The output word tokens, as a list of strings

Return type List[str]

33.2 Package Contents

class flambe.tokenizer.**Tokenizer**
Bases: flambe.Component

Base interface to a Tokenizer object.

Tokenizers implement the *tokenize* method, which takes a string as input and produces a list of strings as output.

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output tokens, as a list of strings

Return type List[str]

__call__(*self*, *example: str*)
Make a tokenizer callable.

class flambe.tokenizer.**CharTokenizer**
Bases: *flambe.tokenizer.Tokenizer*

Implement a character level tokenizer.

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output charachter tokens, as a list of strings

Return type List[str]

class flambe.tokenizer.**WordTokenizer**
Bases: *flambe.tokenizer.Tokenizer*

Implement a word level tokenizer using nltk.tokenize.word_tokenize

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output word tokens, as a list of strings

Return type List[str]

class flambe.tokenizer.**NLTKWordTokenizer**(***kwargs*)
Bases: *flambe.tokenizer.Tokenizer*

Implement a word level tokenizer using nltk.tokenize.word_tokenize

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output word tokens, as a list of strings

Return type List[str]

class flambe.tokenizer.**NGramsTokenizer**(*ngrams: Union[int, List[int]] = 1, exclude_stopwords: bool = False, stop_words: Optional[List] = None*)

Bases: *flambe.tokenizer.Tokenizer*

Implement a n-gram tokenizer

Examples

```
>>> t = NGramsTokenizer(ngrams=2).tokenize("hi how are you?")
['hi, how', 'how are', 'are you?']
```

```
>>> t = NGramsTokenizer(ngrams=[1,2]).tokenize("hi how are you?")
['hi,', 'how', 'are', 'you?', 'hi, how', 'how are', 'are you?']
```

Parameters

- **ngrams** (*Union[int, List[int]]*) – An int or a list of ints. If it's a list of ints, all n-grams (for each int) will be considered in the tokenizer.
- **exclude_stopwords** (*bool*) – Whether to exlude stopword or not. See the related param stop_words
- **stop_words** (*Optional[List]*) – List of stop words to exclude when exclude_stopwords is True. If None set to nltk.corpus.stopwords.

static _tokenize(*example: str, n: int*)

Tokenize an input example using ngrams.

tokenize(*self, example: str*)

Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string.

Returns The output word tokens, as a list of strings

Return type List[str]

class flambe.tokenizer.**BPETokenizer**(*codes_path: str*)

Bases: *flambe.tokenizer.Tokenizer*

Implement a subword level tokenizer using byte pair encoding. Tokenization is done using fastBPE (https://github.com/glample/fastBPE) and requires a fastBPE codes file.

tokenize(*self, example: str*)

Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output subword tokens, as a list of strings

Return type List[str]

class flambe.tokenizer.**LabelTokenizer**(*multilabel_sep: Optional[str] = None*)

Bases: *flambe.tokenizer.Tokenizer*

Base label tokenizer.

This object tokenizes string labels into a list of a single or multiple elements, depending on the provided separator.

tokenize(*self*, *example: str*)
Tokenize an input example.

Parameters **example** (*str*) – The input example, as a string

Returns The output tokens, as a list of strings

Return type List[str]

CHAPTER 34

flambe.vision

34.1 Subpackages

34.1.1 flambe.vision.classification

Submodules

flambe.vision.classification.datasets

Module Contents

class flambe.vision.classification.datasets.**MNISTDataset**(*train_images: np.ndarray = None, train_labels: np.ndarray = None, test_images: np.ndarray = None, test_labels: np.ndarray = None, val_ratio: Optional[float] = 0.2, seed: Optional[int] = None*)

Bases: *flambe.dataset.Dataset*

The official MNIST dataset.

data_type

URL = http://yann.lecun.com/exdb/mnist/

train :List[Tuple[torch.Tensor, torch.Tensor]]
Returns the training data

val :List[Tuple[torch.Tensor, torch.Tensor]]
Returns the validation data

test :List[Tuple[torch.Tensor, torch.Tensor]]
Returns the test data

classmethod from_path(*cls*, *train_images_path: str*, *train_labels_path: str*, *test_images_path: str*, *test_labels_path: str*, *val_ratio: Optional[float] = 0.2*, *seed: Optional[int] = None*)
Initialize the MNISTDataset from local files.

Parameters

- **train_images_path** (*str*) – path to the train images file in the idx format
- **train_labels_path** (*str*) – path to the train labels file in the idx format
- **test_images_path** (*str*) – path to the test images file in the idx format
- **test_labels_path** (*str*) – path to the test labels file in the idx format
- **val_ratio** (*Optional[float]*) – validation set ratio. Default 0.2
- **seed** (*Optional[int]*) – random seed for the validation set split

classmethod _parse_local_gzipped_idx(*cls*, *path: str*)
Parse a local gzipped idx file

classmethod _parse_downloaded_idx(*cls*, *url: str*)
Parse a downloaded idx file

classmethod _parse_idx(*cls*, *data: bytes*)
Parse an idx filie

flambe.vision.classification.datasets.**get_dataset**(*images: np.ndarray*, *labels: np.ndarray*) → List[Tuple[torch.Tensor, torch.Tensor]]

flambe.vision.classification.model

Module Contents

class flambe.vision.classification.model.**ImageClassifier**(*encoder: Module*, *output_layer: Module*)
Bases: *flambe.nn.Module*

Implements a simple image classifier.

This classifier consists of an encocder module, followed by a fully connected output layer that outputs a probability distribution.

encoder
The encoder layer

Type Moodule

output_layer
The output layer, yields a probability distribution over targets

Type *Module*

forward(*self*, *data: Tensor*, *target: Optional[Tensor] = None*)
Run a forward pass through the network.

Parameters

- **data** (*Tensor*) – The input data
- **target** (*Tensor, optional*) – The input targets, optional

Returns The output predictions, and optionally the targets

Return type Union[Tensor, Tuple[Tensor, Tensor]

Package Contents

class flambe.vision.classification.**MNISTDataset**(*train_images: np.ndarray = None*, *train_labels: np.ndarray = None*, *test_images: np.ndarray = None*, *test_labels: np.ndarray = None*, *val_ratio: Optional[float] = 0.2*, *seed: Optional[int] = None*)

Bases: *flambe.dataset.Dataset*

The official MNIST dataset.

data_type

URL = http://yann.lecun.com/exdb/mnist/

train :List[Tuple[torch.Tensor, torch.Tensor]]
Returns the training data

val :List[Tuple[torch.Tensor, torch.Tensor]]
Returns the validation data

test :List[Tuple[torch.Tensor, torch.Tensor]]
Returns the test data

classmethod from_path(*cls*, *train_images_path: str*, *train_labels_path: str*, *test_images_path: str*, *test_labels_path: str*, *val_ratio: Optional[float] = 0.2*, *seed: Optional[int] = None*)
Initialize the MNISTDataset from local files.

Parameters

- **train_images_path** (*str*) – path to the train images file in the idx format
- **train_labels_path** (*str*) – path to the train labels file in the idx format
- **test_images_path** (*str*) – path to the test images file in the idx format
- **test_labels_path** (*str*) – path to the test labels file in the idx format
- **val_ratio** (*Optional[float]*) – validation set ratio. Default 0.2
- **seed** (*Optional[int]*) – random seed for the validation set split

classmethod _parse_local_gzipped_idx(*cls*, *path: str*)
Parse a local gzipped idx file

classmethod _parse_downloaded_idx(*cls*, *url: str*)
Parse a downloaded idx file

classmethod _parse_idx(*cls*, *data: bytes*)
Parse an idx filie

class flambe.vision.classification.**ImageClassifier**(*encoder: Module*, *output_layer: Module*)

Bases: *flambe.nn.Module*

Implements a simple image classifier.

This classifier consists of an encocder module, followed by a fully connected output layer that outputs a probability distribution.

encoder
The encoder layer

Type Moodule

output_layer
The output layer, yields a probability distribution over targets

Type *Module*

forward(*self*, *data: Tensor*, *target: Optional[Tensor] = None*)
Run a forward pass through the network.

Parameters

- **data** (*Tensor*) – The input data
- **target** (*Tensor, optional*) – The input targets, optional

Returns The output predictions, and optionally the targets

Return type Union[Tensor, Tuple[Tensor, Tensor]

Python Module Index

f

flambe.cluster, 103
flambe.cluster.aws, 119
flambe.cluster.cluster, 124
flambe.cluster.const, 130
flambe.cluster.errors, 130
flambe.cluster.instance, 103
flambe.cluster.instance.errors, 103
flambe.cluster.instance.instance, 103
flambe.cluster.ssh, 130
flambe.cluster.utils, 131
flambe.compile, 143
flambe.compile.component, 143
flambe.compile.const, 152
flambe.compile.downloader, 152
flambe.compile.extensions, 154
flambe.compile.registrable, 155
flambe.compile.serialization, 157
flambe.compile.utils, 160
flambe.dataset, 95
flambe.dataset.dataset, 95
flambe.dataset.tabular, 96
flambe.experiment, 173
flambe.experiment.experiment, 174
flambe.experiment.options, 176
flambe.experiment.progress, 177
flambe.experiment.tune_adapter, 178
flambe.experiment.utils, 178
flambe.experiment.webapp, 173
flambe.experiment.webapp.app, 173
flambe.experiment.wording, 182
flambe.export, 187
flambe.export.builder, 187
flambe.export.exporter, 188
flambe.field, 191
flambe.field.bow, 191
flambe.field.field, 192
flambe.field.label, 192
flambe.field.text, 193
flambe.learn, 197
flambe.learn.distillation, 197
flambe.learn.eval, 199
flambe.learn.script, 199
flambe.learn.train, 199
flambe.learn.utils, 201
flambe.logging, 205
flambe.logging.datatypes, 207
flambe.logging.handler, 205
flambe.logging.handler.contextual_file, 205
flambe.logging.handler.tensorboard, 206
flambe.logging.logging, 212
flambe.logging.utils, 212
flambe.metric, 223
flambe.metric.dev, 223
flambe.metric.dev.accuracy, 223
flambe.metric.dev.auc, 223
flambe.metric.dev.binary, 224
flambe.metric.dev.bpc, 226
flambe.metric.dev.perplexity, 226
flambe.metric.loss, 226
flambe.metric.loss.cross_entropy, 226
flambe.metric.loss.nll_loss, 227
flambe.metric.metric, 227
flambe.model, 233
flambe.model.logistic_regression, 233
flambe.nlp, 235
flambe.nlp.classification, 235
flambe.nlp.classification.datasets, 235
flambe.nlp.classification.model, 236
flambe.nlp.fewshot, 238
flambe.nlp.fewshot.model, 238
flambe.nlp.language_modeling, 240
flambe.nlp.language_modeling.datasets, 240
flambe.nlp.language_modeling.fields, 241
flambe.nlp.language_modeling.model, 241
flambe.nlp.language_modeling.sampler, 242

flambe.nlp.transformers, 245
flambe.nlp.transformers.field, 245
flambe.nlp.transformers.model, 246
flambe.nn, 249
flambe.nn.cnn, 254
flambe.nn.distance, 249
flambe.nn.distance.cosine, 249
flambe.nn.distance.distance, 250
flambe.nn.distance.euclidean, 250
flambe.nn.distance.hyperbolic, 251
flambe.nn.embedding, 255
flambe.nn.mlp, 256
flambe.nn.module, 257
flambe.nn.mos, 258
flambe.nn.pooling, 258
flambe.nn.rnn, 260
flambe.nn.sequential, 261
flambe.nn.softmax, 261
flambe.nn.transformer, 262
flambe.nn.transformer_sru, 265
flambe.runnable, 281
flambe.runnable.cluster_runnable, 281
flambe.runnable.context, 282
flambe.runnable.environment, 283
flambe.runnable.error, 284
flambe.runnable.runnable, 285
flambe.runnable.utils, 286
flambe.runner, 293
flambe.runner.report_site_run, 293
flambe.runner.run, 293
flambe.runner.utils, 294
flambe.sampler, 295
flambe.sampler.base, 295
flambe.sampler.episodic, 297
flambe.sampler.sampler, 297
flambe.tokenizer, 301
flambe.tokenizer.char, 301
flambe.tokenizer.label, 301
flambe.tokenizer.subword, 302
flambe.tokenizer.tokenizer, 302
flambe.tokenizer.word, 302
flambe.vision, 307
flambe.vision.classification, 307
flambe.vision.classification.datasets, 307
flambe.vision.classification.model, 308

Index

Symbols

_EMPTY (*in module flambe.compile.component*), 143
__call__() (*flambe.compile.Link method*), 168
__call__() (*flambe.compile.Schema method*), 163
__call__() (*flambe.compile.component.FunctionCallLink method*), 147
__call__() (*flambe.compile.component.Link method*), 147
__call__() (*flambe.compile.component.PickledDataLink method*), 145
__call__() (*flambe.compile.component.Schema method*), 144
__call__() (*flambe.compile.registrable.registration_context method*), 156
__call__() (*flambe.compile.registration_context method*), 162
__call__() (*flambe.logging.logging.ContextInjection method*), 212
__call__() (*flambe.metric.Metric method*), 228
__call__() (*flambe.metric.metric.Metric method*), 228
__call__() (*flambe.tokenizer.Tokenizer method*), 304
__call__() (*flambe.tokenizer.tokenizer.Tokenizer method*), 302
__delitem__() (*flambe.compile.Schema method*), 163
__delitem__() (*flambe.compile.component.Schema method*), 144
__delitem__() (*flambe.dataset.Dataset method*), 99
__delitem__() (*flambe.dataset.dataset.Dataset method*), 95
__delitem__() (*flambe.dataset.tabular.DataView method*), 96
__enter__() (*flambe.cluster.Cluster method*), 132
__enter__() (*flambe.cluster.cluster.Cluster method*), 125
__enter__() (*flambe.cluster.instance.Instance method*), 112
__enter__() (*flambe.cluster.instance.instance.Instance method*), 104
__enter__() (*flambe.compile.component.contextualized_linking method*), 145
__enter__() (*flambe.compile.registrable.registration_context method*), 155
__enter__() (*flambe.compile.registration_context method*), 162
__enter__() (*flambe.logging.TrialLogging method*), 215
__enter__() (*flambe.logging.logging.TrialLogging method*), 212
__enter__() (*flambe.runnable.SafeExecutionContext method*), 289
__enter__() (*flambe.runnable.context.SafeExecutionContext method*), 282
__exit__() (*flambe.cluster.Cluster method*), 132
__exit__() (*flambe.cluster.cluster.Cluster method*), 125
__exit__() (*flambe.cluster.instance.Instance method*), 112
__exit__() (*flambe.cluster.instance.instance.Instance method*), 104
__exit__() (*flambe.compile.component.contextualized_linking method*), 145
__exit__() (*flambe.compile.registrable.registration_context method*), 156
__exit__() (*flambe.compile.registration_context method*), 162
__exit__() (*flambe.logging.TrialLogging method*), 215
__exit__() (*flambe.logging.logging.TrialLogging method*), 212
__exit__() (*flambe.runnable.SafeExecutionContext method*), 289
__exit__() (*flambe.runnable.context.SafeExecutionContext method*), 282
__getattr__() (*flambe.compile.Schema method*), 163
__getattr__() (*flambe.compile.component.Schema method*), 144

__getattr__() (*flambe.nlp.transformers.PretrainedTransformerEmbedder method*), 248
__getattr__() (*flambe.nlp.transformers.model.PretrainedTransformerEmbedder method*), 246
__getitem__() (*flambe.compile.Schema method*), 163
__getitem__() (*flambe.compile.component.Schema method*), 144
__getitem__() (*flambe.dataset.TabularDataset method*), 101
__getitem__() (*flambe.dataset.tabular.DataView method*), 96
__getitem__() (*flambe.dataset.tabular.TabularDataset method*), 98
__getitem__() (*flambe.experiment.GridSearchOptions method*), 185
__getitem__() (*flambe.experiment.SampledUniformSearchOptions method*), 185
__getitem__() (*flambe.experiment.options.GridSearchOptions method*), 177
__getitem__() (*flambe.experiment.options.SampledUniformSearchOptions method*), 177
__init_subclass__() (*flambe.compile.Registrable class method*), 161
__init_subclass__() (*flambe.compile.registrable.Registrable class method*), 156
__iter__() (*flambe.compile.Schema method*), 163
__iter__() (*flambe.compile.component.Schema method*), 144
__iter__() (*flambe.dataset.TabularDataset method*), 101
__iter__() (*flambe.dataset.tabular.TabularDataset method*), 98
__len__() (*flambe.compile.Schema method*), 163
__len__() (*flambe.compile.component.Schema method*), 144
__len__() (*flambe.dataset.TabularDataset method*), 101
__len__() (*flambe.dataset.tabular.DataView method*), 96
__len__() (*flambe.dataset.tabular.TabularDataset method*), 98
__len__() (*flambe.experiment.GridSearchOptions method*), 185
__len__() (*flambe.experiment.SampledUniformSearchOptions method*), 185
__len__() (*flambe.experiment.options.GridSearchOptions method*), 177
__len__() (*flambe.experiment.options.SampledUniformSearchOptions method*), 177
__repr__() (*flambe.compile.Link method*), 168
__repr__() (*flambe.compile.Schema method*), 163
__repr__() (*flambe.compile.component.Link method*), 167
__repr__() (*flambe.compile.component.Schema method*), 144
__repr__() (*flambe.dataset.Dataset method*), 99
__repr__() (*flambe.experiment.GridSearchOptions method*), 185
__repr__() (*flambe.experiment.SampledUniformSearchOptions method*), 185
__repr__() (*flambe.experiment.options.GridSearchOptions method*), 177
__repr__() (*flambe.experiment.options.SampledUniformSearchOptions method*), 177
__repr__() (*flambe.logging.EmbeddingT method*), 218
__repr__() (*flambe.logging.HistogramT method*), 216
__repr__() (*flambe.logging.ImageT method*), 217
__repr__() (*flambe.logging.PRCurveT method*), 218
__repr__() (*flambe.logging.ScalarT method*), 215
__repr__() (*flambe.logging.ScalarsT method*), 216
__repr__() (*flambe.logging.TextT method*), 217
__repr__() (*flambe.logging.datatypes.EmbeddingT method*), 210
__repr__() (*flambe.logging.datatypes.HistogramT method*), 208
__repr__() (*flambe.logging.datatypes.ImageT method*), 209
__repr__() (*flambe.logging.datatypes.PRCurveT method*), 210
__repr__() (*flambe.logging.datatypes.ScalarT method*), 207
__repr__() (*flambe.logging.datatypes.ScalarsT method*), 208
__repr__() (*flambe.logging.datatypes.TextT method*), 209
__repr__() (*flambe.runnable.error.ProtocolError method*), 284
__set_name__() (*flambe.compile.registrable.registrable_factory method*), 157
__set_name__() (*flambe.compile.registrable_factory method*), 162
__setattr__() (*flambe.compile.Schema method*), 163
__setattr__() (*flambe.compile.component.Schema method*), 144
__setitem__() (*flambe.compile.Schema method*), 163
__setitem__() (*flambe.compile.component.Schema method*), 144
__setitem__() (*flambe.dataset.Dataset method*), 99
__setitem__() (*flambe.dataset.dataset.Dataset method*), 95
__setitem__() (*flambe.dataset.tabular.DataView method*), 96
__str__() (*flambe.metric.BinaryPrecision method*), 230

__str__() (*flambe.metric.BinaryRecall method*), 231
__str__() (*flambe.metric.Metric method*), 228
__str__() (*flambe.metric.dev.binary.BinaryPrecision method*), 225
__str__() (*flambe.metric.dev.binary.BinaryRecall method*), 225
__str__() (*flambe.metric.metric.Metric method*), 228
_add_registered_attrs() (*flambe.compile.Component method*), 165
_add_registered_attrs() (*flambe.compile.component.Component method*), 149
_aggregate_preds() (*flambe.learn.DistillationTrainer method*), 203
_aggregate_preds() (*flambe.learn.Trainer method*), 201
_aggregate_preds() (*flambe.learn.distillation.DistillationTrainer method*), 198
_aggregate_preds() (*flambe.learn.train.Trainer method*), 200
_batch_from_nested_col() (*in module flambe.sampler.base*), 295
_batch_to_device() (*flambe.learn.Trainer method*), 201
_batch_to_device() (*flambe.learn.train.Trainer method*), 200
_bfs() (*in module flambe.sampler.base*), 295
_compute_loss() (*flambe.learn.DistillationTrainer method*), 203
_compute_loss() (*flambe.learn.Trainer method*), 201
_compute_loss() (*flambe.learn.distillation.DistillationTrainer method*), 198
_compute_loss() (*flambe.learn.train.Trainer method*), 200
_config_str (*flambe.compile.Component attribute*), 164
_config_str (*flambe.compile.component.Component attribute*), 148
_contains_path() (*in module flambe.runnable.utils*), 286
_convert_to_tree() (*in module flambe.compile.serialization*), 157
_create_cloudwatch_event() (*flambe.cluster.AWSCluster method*), 141
_create_cloudwatch_event() (*flambe.cluster.aws.AWSCluster method*), 123
_create_factories() (*flambe.cluster.AWSCluster method*), 139
_create_factories() (*flambe.cluster.aws.AWSCluster method*), 121
_create_orchestrator() (*flambe.cluster.AWSCluster method*), 138
_create_orchestrator() (*flambe.cluster.aws.AWSCluster method*), 121
_create_train_iterator() (*flambe.learn.Trainer method*), 201
_create_train_iterator() (*flambe.learn.train.Trainer method*), 200
_default_padding_mask() (*in module flambe.nn.pooling*), 259
_delete_cloudwatch_event() (*flambe.cluster.AWSCluster method*), 140
_delete_cloudwatch_event() (*flambe.cluster.aws.AWSCluster method*), 123
_download_remote_extension() (*in module flambe.compile.extensions*), 154
_download_svn() (*in module flambe.compile.extensions*), 154
_dump_experiment_file() (*flambe.experiment.Experiment method*), 184
_dump_experiment_file() (*flambe.experiment.experiment.Experiment method*), 176
_eval_step() (*flambe.learn.Trainer method*), 202
_eval_step() (*flambe.learn.train.Trainer method*), 200
_existing_cluster() (*flambe.cluster.AWSCluster method*), 138
_existing_cluster() (*flambe.cluster.aws.AWSCluster method*), 120
_extract_prefix() (*in module flambe.compile.serialization*), 158
_find_default_ami() (*flambe.cluster.AWSCluster method*), 141
_find_default_ami() (*flambe.cluster.aws.AWSCluster method*), 124
_flambe_version (*flambe.compile.Component attribute*), 164
_flambe_version (*flambe.compile.component.Component attribute*), 148
_generic_launch_instances() (*flambe.cluster.AWSCluster method*), 139
_generic_launch_instances() (*flambe.cluster.aws.AWSCluster method*), 121
_get_alarm_name() (*flambe.cluster.AWSCluster method*), 140
_get_alarm_name()

(*flambe.cluster.aws.AWSCluster method*), 123
_get_all_hosts() (*flambe.cluster.Cluster method*), 132
_get_all_hosts() (*flambe.cluster.cluster.Cluster method*), 126
_get_all_tags() (*flambe.cluster.AWSCluster method*), 138
_get_all_tags() (*flambe.cluster.aws.AWSCluster method*), 121
_get_ami() (*flambe.cluster.AWSCluster method*), 141
_get_ami() (*flambe.cluster.aws.AWSCluster method*), 124
_get_boto_instance_by_host() (*flambe.cluster.AWSCluster method*), 140
_get_boto_instance_by_host() (*flambe.cluster.aws.AWSCluster method*), 122
_get_boto_private_host() (*flambe.cluster.AWSCluster method*), 139
_get_boto_private_host() (*flambe.cluster.aws.AWSCluster method*), 122
_get_boto_public_host() (*flambe.cluster.AWSCluster method*), 139
_get_boto_public_host() (*flambe.cluster.aws.AWSCluster method*), 122
_get_boto_session() (*flambe.cluster.AWSCluster method*), 137
_get_boto_session() (*flambe.cluster.aws.AWSCluster method*), 120
_get_cli() (*flambe.cluster.instance.Instance method*), 112
_get_cli() (*flambe.cluster.instance.instance.Instance method*), 105
_get_context_logger() (*in module flambe.logging.utils*), 213
_get_existing_tags() (*flambe.cluster.AWSCluster method*), 138
_get_existing_tags() (*flambe.cluster.aws.AWSCluster method*), 120
_get_images() (*flambe.cluster.AWSCluster method*), 141
_get_images() (*flambe.cluster.aws.AWSCluster method*), 124
_get_instance_id_by_host() (*flambe.cluster.AWSCluster method*), 140
_get_instance_id_by_host() (*flambe.cluster.aws.AWSCluster method*), 123
_has_svn() (*in module flambe.compile.extensions*), 154
_is_url() (*in module flambe.compile.utils*), 160
_link_context_active (*in module flambe.compile.component*), 145
_link_obj_stash (*in module flambe.compile.component*), 145
_link_root_obj (*in module flambe.compile.component*), 145
_load_file() (*flambe.dataset.TabularDataset class method*), 101
_load_file() (*flambe.dataset.tabular.TabularDataset class method*), 98
_load_file() (*flambe.nlp.classification.SSTDataset class method*), 237
_load_file() (*flambe.nlp.classification.TRECDataset class method*), 237
_load_file() (*flambe.nlp.classification.datasets.SSTDataset class method*), 235
_load_file() (*flambe.nlp.classification.datasets.TRECDataset class method*), 235
_load_registered_attrs() (*flambe.compile.Component method*), 166
_load_registered_attrs() (*flambe.compile.component.Component method*), 150
_load_state() (*flambe.compile.Component method*), 166
_load_state() (*flambe.compile.component.Component method*), 150
_load_state() (*flambe.learn.Trainer method*), 202
_load_state() (*flambe.learn.train.Trainer method*), 200
_load_state_dict_hook() (*flambe.compile.Component method*), 166
_load_state_dict_hook() (*flambe.compile.component.Component method*), 149
_metadata (*flambe.compile.State attribute*), 171
_metadata (*flambe.compile.serialization.State attribute*), 157
_parse_downloaded_idx() (*flambe.vision.classification.MNISTDataset class method*), 309
_parse_downloaded_idx() (*flambe.vision.classification.datasets.MNISTDataset class method*), 308
_parse_idx() (*flambe.vision.classification.MNISTDataset class method*), 309
_parse_idx() (*flambe.vision.classification.datasets.MNISTDataset class method*), 308
_parse_local_gzipped_idx() (*flambe.vision.classification.MNISTDataset class method*), 309
_parse_local_gzipped_idx()

(*flambe.vision.classification.datasets.MNISTDataset class method*), 308
_prefix_keys() (*in module flambe.compile.serialization*), 158
_process() (*flambe.nlp.language_modeling.Enwiki8 method*), 243
_process() (*flambe.nlp.language_modeling.PTBDataset method*), 243
_process() (*flambe.nlp.language_modeling.Wiki103 method*), 243
_process() (*flambe.nlp.language_modeling.datasets.Enwiki8 method*), 241
_process() (*flambe.nlp.language_modeling.datasets.PTBDataset method*), 240
_process() (*flambe.nlp.language_modeling.datasets.Wiki103 method*), 240
_put_fake_cloudwatch_data() (*flambe.cluster.AWSCluster method*), 140
_put_fake_cloudwatch_data() (*flambe.cluster.aws.AWSCluster method*), 123
_reg_prefix (*in module flambe.compile.registrable*), 155
_remote_script() (*flambe.cluster.instance.Instance method*), 113
_remote_script() (*flambe.cluster.instance.instance.Instance method*), 106
_reset_parameters() (*flambe.nn.TransformerDecoder method*), 277
_reset_parameters() (*flambe.nn.TransformerEncoder method*), 276
_reset_parameters() (*flambe.nn.TransformerSRUDecoder method*), 279
_reset_parameters() (*flambe.nn.TransformerSRUEncoder method*), 278
_reset_parameters() (*flambe.nn.transformer.TransformerDecoder method*), 264
_reset_parameters() (*flambe.nn.transformer.TransformerEncoder method*), 263
_reset_parameters() (*flambe.nn.transformer_sru.TransformerSRUDecoder method*), 267
_reset_parameters() (*flambe.nn.transformer_sru.TransformerSRUEncoder method*), 267
_restore() (*flambe.experiment.TuneAdapter method*), 185
_restore() (*flambe.experiment.tune_adapter.TuneAdapter method*), 178
_run_cmd() (*flambe.cluster.instance.Instance method*), 112
_run_cmd() (*flambe.cluster.instance.instance.Instance method*), 105
_run_script() (*flambe.cluster.instance.Instance method*), 113
_run_script() (*flambe.cluster.instance.instance.Instance method*), 106
_save() (*flambe.experiment.ProgressState method*), 184
_save() (*flambe.experiment.TuneAdapter method*), 185
_save() (*flambe.experiment.progress.ProgressState method*), 177
_save() (*flambe.experiment.tune_adapter.TuneAdapter method*), 178
_set_transforms() (*flambe.dataset.TabularDataset method*), 100
_set_transforms() (*flambe.dataset.tabular.TabularDataset method*), 97
_setup() (*flambe.experiment.TuneAdapter method*), 184
_setup() (*flambe.experiment.tune_adapter.TuneAdapter method*), 178
_state() (*flambe.compile.Component method*), 165
_state() (*flambe.compile.component.Component method*), 149
_state() (*flambe.learn.Trainer method*), 202
_state() (*flambe.learn.train.Trainer method*), 200
_state_dict_hook() (*flambe.compile.Component static method*), 164
_state_dict_hook() (*flambe.compile.component.Component static method*), 148
_stop() (*flambe.experiment.TuneAdapter method*), 185
_stop() (*flambe.experiment.tune_adapter.TuneAdapter method*), 178
_sum_with_padding_mask() (*in module flambe.nn.pooling*), 260
_tokenize() (*flambe.tokenizer.NGramsTokenizer static method*), 305
_tokenize() (*flambe.tokenizer.word.NGramsTokenizer static method*), 303
_train() (*flambe.experiment.TuneAdapter method*), 184
_train() (*flambe.experiment.tune_adapter.TuneAdapter method*), 178
_train_step() (*flambe.learn.Trainer method*), 201
_train_step() (*flambe.learn.train.Trainer method*), 200
_traverse_all_nodes() (*in module flambe.compile.serialization*), 157
_update_link_refs() (*in module*

flambe.compile.serialization), 158
_update_save_tree() (*in module flambe.compile.serialization*), 157
_update_tags() (*flambe.cluster.AWSCluster method*), 138
_update_tags() (*flambe.cluster.aws.AWSCluster method*), 121
_yaml_registered_factories (*flambe.compile.Registrable attribute*), 161
_yaml_registered_factories (*flambe.compile.registrable.Registrable attribute*), 156
_yaml_tag_namespace (*flambe.compile.Registrable attribute*), 161
_yaml_tag_namespace (*flambe.compile.registrable.Registrable attribute*), 156
_yaml_tags (*flambe.compile.Registrable attribute*), 161
_yaml_tags (*flambe.compile.registrable.Registrable attribute*), 156

A

A (*in module flambe.compile.component*), 143
A (*in module flambe.compile.registrable*), 155
Accuracy (*class in flambe.metric*), 229
Accuracy (*class in flambe.metric.dev.accuracy*), 223
add_extensions_metadata() (*flambe.compile.component.Schema method*), 144
add_extensions_metadata() (*flambe.compile.Schema method*), 163
aggregate_extensions_metadata() (*flambe.compile.Component method*), 167
aggregate_extensions_metadata() (*flambe.compile.component.Component method*), 151
aggregate_extensions_metadata() (*flambe.compile.component.Schema method*), 144
aggregate_extensions_metadata() (*flambe.compile.Schema method*), 163
alias() (*in module flambe.compile*), 161
alias() (*in module flambe.compile.registrable*), 156
all_subclasses() (*in module flambe.compile*), 169
all_subclasses() (*in module flambe.compile.utils*), 160
analyze_download_params() (*in module flambe.experiment.webapp.app*), 173
app (*in module flambe.experiment.webapp.app*), 173
arccosh() (*in module flambe.nn.distance.hyperbolic*), 251
AUC (*class in flambe.metric*), 230
AUC (*class in flambe.metric.dev.auc*), 223
autogen() (*flambe.dataset.tabular.TabularDataset class method*), 97
autogen() (*flambe.dataset.TabularDataset class method*), 100
AvgPooling (*class in flambe.nn*), 275
AvgPooling (*class in flambe.nn.pooling*), 259
AWS_FLAMBE_ACCOUNT (*in module flambe.cluster.const*), 130
AWSCluster (*class in flambe.cluster*), 136
AWSCluster (*class in flambe.cluster.aws*), 119

B

baseFilename (*flambe.logging.handler.contextual_file.ContextualFileHa attribute*), 206
BaseSampler (*class in flambe.sampler*), 299
BaseSampler (*class in flambe.sampler.base*), 296
BinaryAccuracy (*class in flambe.metric*), 231
BinaryAccuracy (*class in flambe.metric.dev.binary*), 224
BinaryMetric (*class in flambe.metric.dev.binary*), 224
BinaryPrecision (*class in flambe.metric*), 230
BinaryPrecision (*class in flambe.metric.dev.binary*), 225
BinaryRecall (*class in flambe.metric*), 230
BinaryRecall (*class in flambe.metric.dev.binary*), 225
bins (*flambe.logging.datatypes.HistogramT attribute*), 208
bins (*flambe.logging.HistogramT attribute*), 216
BoWField (*class in flambe.field*), 195
BoWField (*class in flambe.field.bow*), 191
BPC (*class in flambe.metric*), 229
BPC (*class in flambe.metric.dev.bpc*), 226
BPETokenizer (*class in flambe.tokenizer*), 305
BPETokenizer (*class in flambe.tokenizer.subword*), 302
Builder (*class in flambe.export*), 188
Builder (*class in flambe.export.builder*), 187

C

C (*in module flambe.compile.component*), 143
canonical_name (*flambe.logging.handler.contextual_file.ContextualFile attribute*), 206
CharTokenizer (*class in flambe.tokenizer*), 304
CharTokenizer (*class in flambe.tokenizer.char*), 301
check_links() (*in module flambe.experiment.utils*), 178
check_ray_cluster() (*flambe.cluster.Cluster method*), 133
check_ray_cluster() (*flambe.cluster.cluster.Cluster method*), 127
check_search() (*in module flambe.experiment.utils*), 178

check_system_reqs() (*in module flambe.runner.utils*), 294
check_tags() (*flambe.runnable.context.SafeExecutionContext method*), 283
check_tags() (*flambe.runnable.SafeExecutionContext method*), 289
checkpoint_end() (*flambe.experiment.progress.ProgressState method*), 177
checkpoint_end() (*flambe.experiment.ProgressState method*), 184
checkpoint_start() (*flambe.experiment.progress.ProgressState method*), 177
checkpoint_start() (*flambe.experiment.ProgressState method*), 184
children (*flambe.compile.serialization.SaveTreeNode attribute*), 157
class_name (*flambe.compile.serialization.SaveTreeNode attribute*), 157
clean_container_by_command() (*flambe.cluster.instance.Instance method*), 114
clean_container_by_command() (*flambe.cluster.instance.instance.Instance method*), 107
clean_container_by_image() (*flambe.cluster.instance.Instance method*), 114
clean_container_by_image() (*flambe.cluster.instance.instance.Instance method*), 107
clean_containers() (*flambe.cluster.instance.Instance method*), 114
clean_containers() (*flambe.cluster.instance.instance.Instance method*), 107
clip_gradient_norm() (*flambe.nn.Module method*), 269
clip_gradient_norm() (*flambe.nn.module.Module method*), 258
clip_params() (*flambe.nn.Module method*), 269
clip_params() (*flambe.nn.module.Module method*), 258
close() (*flambe.logging.handler.tensorboard.TensorboardXHandler method*), 207
Cluster (*class in flambe.cluster*), 131
Cluster (*class in flambe.cluster.cluster*), 125
cluster_has_key() (*flambe.cluster.Cluster method*), 135
cluster_has_key() (*flambe.cluster.cluster.Cluster method*), 129
ClusterConfigurationError, 130
ClusterError, 130
ClusterResource (*class in flambe.experiment.options*), 177
ClusterRunnable (*class in flambe.runnable*), 288
ClusterRunnable (*class in flambe.runnable.cluster_runnable*), 281
cnn (*flambe.nn.cnn.CNNEncoder attribute*), 254
cnn (*flambe.nn.CNNEncoder attribute*), 273
CNNEncoder (*class in flambe.nn*), 273
CNNEncoder (*class in flambe.nn.cnn*), 254
collate_fn() (*flambe.nlp.language_modeling.CorpusSampler static method*), 244
collate_fn() (*flambe.nlp.language_modeling.sampler.CorpusSampler static method*), 242
collate_fn() (*in module flambe.sampler.base*), 296
coloredlogs (*in module flambe.logging*), 219
coloredlogs (*in module flambe.logging.utils*), 213
colorize_exceptions() (*in module flambe.logging.logging*), 212
cols (*flambe.dataset.tabular.TabularDataset attribute*), 97
cols (*flambe.dataset.TabularDataset attribute*), 100
cols() (*flambe.dataset.tabular.DataView method*), 96
CompilationError, 143
compile() (*flambe.compile.Component class method*), 167
compile() (*flambe.compile.component.Component class method*), 151
compile_runnable() (*flambe.runnable.context.SafeExecutionContext method*), 283
compile_runnable() (*flambe.runnable.SafeExecutionContext method*), 290
Component (*class in flambe.compile*), 164
Component (*class in flambe.compile.component*), 147
component_subclass (*flambe.compile.component.Schema attribute*), 144
component_subclass (*flambe.compile.Schema attribute*), 163
compute() (*flambe.metric.Accuracy method*), 229
compute() (*flambe.metric.AUC method*), 230
compute() (*flambe.metric.BPC method*), 229
compute() (*flambe.metric.dev.accuracy.Accuracy method*), 223
compute() (*flambe.metric.dev.auc.AUC method*), 223
compute() (*flambe.metric.dev.binary.BinaryMetric method*), 224
compute() (*flambe.metric.dev.bpc.BPC method*), 226
compute() (*flambe.metric.dev.perplexity.Perplexity method*), 226
compute() (*flambe.metric.loss.cross_entropy.MultiLabelCrossEntropy method*), 227

compute() (*flambe.metric.loss.nll_loss.MultiLabelNLLLoss method*), 227
compute() (*flambe.metric.Metric method*), 228
compute() (*flambe.metric.metric.Metric method*), 228
compute() (*flambe.metric.MultiLabelCrossEntropy method*), 228
compute() (*flambe.metric.MultiLabelNLLLoss method*), 229
compute() (*flambe.metric.Perplexity method*), 229
compute_binary() (*flambe.metric.BinaryAccuracy method*), 231
compute_binary() (*flambe.metric.BinaryPrecision method*), 230
compute_binary() (*flambe.metric.BinaryRecall method*), 230
compute_binary() (*flambe.metric.dev.binary.BinaryAccuracy method*), 224
compute_binary() (*flambe.metric.dev.binary.BinaryMetric method*), 224
compute_binary() (*flambe.metric.dev.binary.BinaryPrecision method*), 225
compute_binary() (*flambe.metric.dev.binary.BinaryRecall method*), 225
compute_binary() (*flambe.metric.dev.binary.F1 method*), 225
compute_binary() (*flambe.metric.F1 method*), 231
compute_prototypes() (*flambe.nlp.fewshot.model.PrototypicalTextClassifier method*), 239
compute_prototypes() (*flambe.nlp.fewshot.PrototypicalTextClassifier method*), 239
config (*flambe.compile.serialization.SaveTreeNode attribute*), 157
config (*flambe.export.Builder attribute*), 188
config (*flambe.export.builder.Builder attribute*), 187
config (*flambe.runnable.cluster_runnable.ClusterRunnable attribute*), 281
config (*flambe.runnable.ClusterRunnable attribute*), 288
config (*flambe.runnable.Runnable attribute*), 287
config (*flambe.runnable.runnable.Runnable attribute*), 285
CONFIG_FILE_NAME (*in module flambe.compile.const*), 152
console_log() (*in module flambe.experiment.webapp.app*), 174
contains() (*flambe.compile.component.Schema method*), 144
contains() (*flambe.compile.Schema method*), 163
contains_gpu() (*flambe.cluster.instance.Instance method*), 116
contains_gpu() (*flambe.cluster.instance.instance.Instance method*), 109
contains_gpu_factories() (*flambe.cluster.Cluster method*), 135
contains_gpu_factories() (*flambe.cluster.cluster.Cluster method*), 129
content (*flambe.runnable.Runnable attribute*), 287
content (*flambe.runnable.runnable.Runnable attribute*), 285
ContextInjection (*class in flambe.logging.logging*), 212
ContextualFileHandler (*class in flambe.logging.handler.contextual_file*), 205
contextualized_linking (*class in flambe.compile.component*), 145
conv_block() (*in module flambe.nn.cnn*), 254
convert() (*flambe.compile.component.Link method*), 147
convert() (*flambe.compile.Link method*), 168
convert() (*flambe.experiment.GridSearchOptions method*), 185
convert() (*flambe.experiment.options.GridSearchOptions method*), 177
convert() (*flambe.experiment.options.Options method*), 176
convert() (*flambe.experiment.options.SampledUniformSearchOptions method*), 177
convert() (*flambe.experiment.SampledUniformSearchOptions method*), 185
convert_tune() (*in module flambe.experiment.utils*), 179
CorpusSampler (*class in flambe.nlp.language_modeling*), 244
CorpusSampler (*class in flambe.nlp.language_modeling.sampler*), 242
CosineDistance (*class in flambe.nn.distance*), 253
CosineDistance (*class in flambe.nn.distance.cosine*), 249
CosineMean (*class in flambe.nn.distance*), 253
CosineMean (*class in flambe.nn.distance.cosine*), 249
CPUFactoryInsT (*in module flambe.cluster.cluster*), 124
CPUFactoryInstance (*class in flambe.cluster.instance*), 116
CPUFactoryInstance (*class in flambe.cluster.instance.instance*), 109
create_cloudwatch_events() (*flambe.cluster.aws.AWSCluster method*), 123
create_cloudwatch_events() (*flambe.cluster.AWSCluster method*), 140
create_dirs() (*flambe.cluster.Cluster method*), 132
create_dirs() (*flambe.cluster.cluster.Cluster method*), 126
create_dirs() (*flambe.cluster.instance.Instance*

method), 116
create_dirs() (*flambe.cluster.instance.instance.Instance method*), 108
create_link_str() (*in module flambe.compile.component*), 146
current_log_dir (*flambe.logging.handler.contextual_file.ContextualFileHandler attribute*), 205

D

d (*in module flambe.logging.utils*), 213
data_type (*flambe.vision.classification.datasets.MNISTDataset attribute*), 307
data_type (*flambe.vision.classification.MNISTDataset attribute*), 309
DATA_TYPES (*in module flambe.logging.datatypes*), 211
DataLoggingFilter (*class in flambe.logging.datatypes*), 211
Dataset (*class in flambe.dataset*), 99
Dataset (*class in flambe.dataset.dataset*), 95
DataView (*class in flambe.dataset.tabular*), 96
decoder (*flambe.nlp.fewshot.model.PrototypicalTextClassifier attribute*), 238
decoder (*flambe.nlp.fewshot.PrototypicalTextClassifier attribute*), 239
default (*flambe.logging.datatypes.DataLoggingFilter attribute*), 211
DEFAULT_PROTOCOL (*in module flambe.compile.const*), 152
DEFAULT_SERIALIZATION_PROTOCOL_VERSION (*in module flambe.compile.const*), 152
DEFAULT_USER_PROVIDER (*in module flambe.runnable.utils*), 286
delay (*flambe.logging.handler.contextual_file.ContextualFileHandler attribute*), 206
deserialize() (*flambe.compile.component.Schema static method*), 145
deserialize() (*flambe.compile.Schema static method*), 164
dist() (*in module flambe.nn.distance.hyperbolic*), 251
DistanceModule (*class in flambe.nn.distance*), 252
DistanceModule (*class in flambe.nn.distance.distance*), 250
DistillationTrainer (*class in flambe.learn*), 203
DistillationTrainer (*class in flambe.learn.distillation*), 198
distribute_keys() (*flambe.cluster.Cluster method*), 135
distribute_keys() (*flambe.cluster.cluster.Cluster method*), 129
divide_nested_grid_search_options() (*in module flambe.experiment.utils*), 180
dont_include (*flambe.logging.datatypes.DataLoggingFilter attribute*), 211
download() (*in module flambe.experiment.webapp.app*), 173
download_extensions() (*in module flambe.compile.extensions*), 154
download_http_file() (*in module flambe.compile.downloader*), 153
download_logs() (*in module flambe.experiment.webapp.app*), 173
download_manager() (*in module flambe.compile.downloader*), 153
download_s3_file() (*in module flambe.compile.downloader*), 153
download_s3_folder() (*in module flambe.compile.downloader*), 153
drop (*flambe.nlp.classification.model.TextClassifier attribute*), 236
drop (*flambe.nlp.classification.TextClassifier attribute*), 237
drop (*flambe.nlp.fewshot.model.PrototypicalTextClassifier attribute*), 238
drop (*flambe.nlp.fewshot.PrototypicalTextClassifier attribute*), 239
drop (*flambe.nn.Embedder attribute*), 272
drop (*flambe.nn.embedding.Embedder attribute*), 256
dynamic_component() (*in module flambe.compile*), 168
dynamic_component() (*in module flambe.compile.component*), 151

E

Embedder (*class in flambe.nn*), 271
Embedder (*class in flambe.nn.embedding*), 256
embedder (*flambe.nlp.classification.model.TextClassifier attribute*), 236
embedder (*flambe.nlp.classification.TextClassifier attribute*), 237
Embeddings (*class in flambe.nn*), 270
Embeddings (*class in flambe.nn.embedding*), 255
embeddings (*flambe.nn.Embedder attribute*), 271
embeddings (*flambe.nn.embedding.Embedder attribute*), 256
EmbeddingT (*class in flambe.logging*), 218
EmbeddingT (*class in flambe.logging.datatypes*), 210
emit() (*flambe.logging.handler.contextual_file.ContextualFileHandler method*), 206
emit() (*flambe.logging.handler.tensorboard.TensorboardXHandler method*), 206
encoder (*flambe.nlp.fewshot.model.PrototypicalTextClassifier attribute*), 238
encoder (*flambe.nlp.fewshot.PrototypicalTextClassifier attribute*), 239
encoder (*flambe.nn.Embedder attribute*), 272
encoder (*flambe.nn.embedding.Embedder attribute*), 256

encoder (*flambe.vision.classification.ImageClassifier attribute*), 310
encoder (*flambe.vision.classification.model.ImageClassifier attribute*), 308
encoding (*flambe.logging.handler.contextual_file.ContextualFileHandler attribute*), 206
env (*flambe.runnable.cluster_runnable.ClusterRunnable attribute*), 281
env (*flambe.runnable.ClusterRunnable attribute*), 288
Enwiki8 (*class in flambe.nlp.language_modeling*), 243
Enwiki8 (*class in flambe.nlp.language_modeling.datasets*), 240
ENWIKI_URL (*flambe.nlp.language_modeling.datasets.Enwiki8 attribute*), 241
ENWIKI_URL (*flambe.nlp.language_modeling.Enwiki8 attribute*), 243
EpisodicSampler (*class in flambe.sampler*), 299
EpisodicSampler (*class in flambe.sampler.episodic*), 297
EPSILON (*in module flambe.nn.distance.hyperbolic*), 251
EuclideanDistance (*class in flambe.nn.distance*), 252
EuclideanDistance (*class in flambe.nn.distance.euclidean*), 250
EuclideanMean (*class in flambe.nn.distance*), 252
EuclideanMean (*class in flambe.nn.distance.euclidean*), 250
Evaluator (*class in flambe.learn*), 202
Evaluator (*class in flambe.learn.eval*), 199
execute() (*flambe.cluster.Cluster method*), 135
execute() (*flambe.cluster.cluster.Cluster method*), 128
ExistentResourceError, 285
existing_dir() (*flambe.cluster.Cluster method*), 134
existing_dir() (*flambe.cluster.cluster.Cluster method*), 127
existing_dir() (*flambe.cluster.instance.Instance method*), 115
existing_dir() (*flambe.cluster.instance.instance.Instance method*), 108
existing_flambe_execution() (*flambe.cluster.Cluster method*), 134
existing_flambe_execution() (*flambe.cluster.cluster.Cluster method*), 127
existing_ray_cluster() (*flambe.cluster.Cluster method*), 134
existing_ray_cluster() (*flambe.cluster.cluster.Cluster method*), 127
existing_tmux_session() (*flambe.cluster.instance.instance.OrchestratorInstance method*), 110
existing_tmux_session() (*flambe.cluster.instance.OrchestratorInstance method*), 118
exp_map() (*in module flambe.nn.distance.hyperbolic*), 251
Experiment (*class in flambe.experiment*), 182
ExperimentHandler (*class in flambe.experiment.experiment*), 174
Exporter (*class in flambe.export*), 188
Exporter (*class in flambe.export.exporter*), 188
extensions (*flambe.runnable.Runnable attribute*), 287
extensions (*flambe.runnable.runnable.Runnable attribute*), 285
extract_dict() (*in module flambe.experiment.utils*), 180
extract_needed_blocks() (*in module flambe.experiment.utils*), 181

F

F1 (*class in flambe.metric*), 231
F1 (*class in flambe.metric.dev.binary*), 225
factories_ips (*flambe.runnable.environment.RemoteEnvironment attribute*), 284
factories_ips (*flambe.runnable.RemoteEnvironment attribute*), 290
FactoryInsT (*in module flambe.cluster.cluster*), 124
FactoryT (*in module flambe.cluster.ssh*), 130
Field (*class in flambe.field*), 194
Field (*class in flambe.field.field*), 192
fill_defaults() (*in module flambe.compile.component*), 147
filter() (*flambe.logging.datatypes.DataLoggingFilter method*), 211
filter() (*flambe.logging.logging.ContextInjection method*), 212
filter() (*flambe.logging.logging.FlambeFilter method*), 212
finish() (*flambe.experiment.progress.ProgressState method*), 177
finish() (*flambe.experiment.ProgressState method*), 184
first_parse() (*flambe.runnable.context.SafeExecutionContext method*), 283
first_parse() (*flambe.runnable.SafeExecutionContext method*), 289
FirstPooling (*class in flambe.nn*), 274
FirstPooling (*class in flambe.nn.pooling*), 258
fix_relpaths_in_config() (*flambe.cluster.instance.Instance method*), 112
fix_relpaths_in_config() (*flambe.cluster.instance.instance.Instance method*), 104
flambe.cluster (*module*), 103
flambe.cluster.aws (*module*), 119

flambe.cluster.cluster (*module*), 124
flambe.cluster.const (*module*), 130
flambe.cluster.errors (*module*), 130
flambe.cluster.instance (*module*), 103
flambe.cluster.instance.errors (*module*), 103
flambe.cluster.instance.instance (*module*), 103
flambe.cluster.ssh (*module*), 130
flambe.cluster.utils (*module*), 131
flambe.compile (*module*), 143
flambe.compile.component (*module*), 143
flambe.compile.const (*module*), 152
flambe.compile.downloader (*module*), 152
flambe.compile.extensions (*module*), 154
flambe.compile.registrable (*module*), 155
flambe.compile.serialization (*module*), 157
flambe.compile.utils (*module*), 160
flambe.dataset (*module*), 95
flambe.dataset.dataset (*module*), 95
flambe.dataset.tabular (*module*), 96
flambe.experiment (*module*), 173
flambe.experiment.experiment (*module*), 174
flambe.experiment.options (*module*), 176
flambe.experiment.progress (*module*), 177
flambe.experiment.tune_adapter (*module*), 178
flambe.experiment.utils (*module*), 178
flambe.experiment.webapp (*module*), 173
flambe.experiment.webapp.app (*module*), 173
flambe.experiment.wording (*module*), 182
flambe.export (*module*), 187
flambe.export.builder (*module*), 187
flambe.export.exporter (*module*), 188
flambe.field (*module*), 191
flambe.field.bow (*module*), 191
flambe.field.field (*module*), 192
flambe.field.label (*module*), 192
flambe.field.text (*module*), 193
flambe.learn (*module*), 197
flambe.learn.distillation (*module*), 197
flambe.learn.eval (*module*), 199
flambe.learn.script (*module*), 199
flambe.learn.train (*module*), 199
flambe.learn.utils (*module*), 201
flambe.logging (*module*), 205
flambe.logging.datatypes (*module*), 207
flambe.logging.handler (*module*), 205
flambe.logging.handler.contextual_file (*module*), 205
flambe.logging.handler.tensorboard (*module*), 206
flambe.logging.logging (*module*), 212
flambe.logging.utils (*module*), 212
flambe.metric (*module*), 223
flambe.metric.dev (*module*), 223
flambe.metric.dev.accuracy (*module*), 223
flambe.metric.dev.auc (*module*), 223
flambe.metric.dev.binary (*module*), 224
flambe.metric.dev.bpc (*module*), 226
flambe.metric.dev.perplexity (*module*), 226
flambe.metric.loss (*module*), 226
flambe.metric.loss.cross_entropy (*module*), 226
flambe.metric.loss.nll_loss (*module*), 227
flambe.metric.metric (*module*), 227
flambe.model (*module*), 233
flambe.model.logistic_regression (*module*), 233
flambe.nlp (*module*), 235
flambe.nlp.classification (*module*), 235
flambe.nlp.classification.datasets (*module*), 235
flambe.nlp.classification.model (*module*), 236
flambe.nlp.fewshot (*module*), 238
flambe.nlp.fewshot.model (*module*), 238
flambe.nlp.language_modeling (*module*), 240
flambe.nlp.language_modeling.datasets (*module*), 240
flambe.nlp.language_modeling.fields (*module*), 241
flambe.nlp.language_modeling.model (*module*), 241
flambe.nlp.language_modeling.sampler (*module*), 242
flambe.nlp.transformers (*module*), 245
flambe.nlp.transformers.field (*module*), 245
flambe.nlp.transformers.model (*module*), 246
flambe.nn (*module*), 249
flambe.nn.cnn (*module*), 254
flambe.nn.distance (*module*), 249
flambe.nn.distance.cosine (*module*), 249
flambe.nn.distance.distance (*module*), 250
flambe.nn.distance.euclidean (*module*), 250
flambe.nn.distance.hyperbolic (*module*), 251
flambe.nn.embedding (*module*), 255
flambe.nn.mlp (*module*), 256
flambe.nn.module (*module*), 257
flambe.nn.mos (*module*), 258
flambe.nn.pooling (*module*), 258
flambe.nn.rnn (*module*), 260
flambe.nn.sequential (*module*), 261
flambe.nn.softmax (*module*), 261
flambe.nn.transformer (*module*), 262

flambe.nn.transformer_sru (*module*), 265
flambe.runnable (*module*), 281
flambe.runnable.cluster_runnable (*module*), 281
flambe.runnable.context (*module*), 282
flambe.runnable.environment (*module*), 283
flambe.runnable.error (*module*), 284
flambe.runnable.runnable (*module*), 285
flambe.runnable.utils (*module*), 286
flambe.runner (*module*), 293
flambe.runner.report_site_run (*module*), 293
flambe.runner.run (*module*), 293
flambe.runner.utils (*module*), 294
flambe.sampler (*module*), 295
flambe.sampler.base (*module*), 295
flambe.sampler.episodic (*module*), 297
flambe.sampler.sampler (*module*), 297
flambe.tokenizer (*module*), 301
flambe.tokenizer.char (*module*), 301
flambe.tokenizer.label (*module*), 301
flambe.tokenizer.subword (*module*), 302
flambe.tokenizer.tokenizer (*module*), 302
flambe.tokenizer.word (*module*), 302
flambe.vision (*module*), 307
flambe.vision.classification (*module*), 307
flambe.vision.classification.datasets (*module*), 307
flambe.vision.classification.model (*module*), 308
FLAMBE_CLASS_KEY (*in module flambe.compile.const*), 152
FLAMBE_CONFIG_KEY (*in module flambe.compile.const*), 152
FLAMBE_DIRECTORIES_KEY (*in module flambe.compile.const*), 152
flambe_own_running_instances() (*flambe.cluster.aws.AWSCluster method*), 121
flambe_own_running_instances() (*flambe.cluster.AWSCluster method*), 138
FLAMBE_SOURCE_KEY (*in module flambe.compile.const*), 152
FLAMBE_STASH_KEY (*in module flambe.compile.const*), 152
FlambeFilter (*class in flambe.logging.logging*), 212
flush() (*flambe.logging.handler.tensorboard.TensorboardXHandler method*), 207
flush() (*flambe.logging.logging.TqdmFileWrapper method*), 212
forward() (*flambe.model.logistic_regression.LogisticRegression method*), 233
forward() (*flambe.model.LogisticRegression method*), 234
forward() (*flambe.nlp.classification.model.TextClassifier method*), 236
forward() (*flambe.nlp.classification.TextClassifier method*), 238
forward() (*flambe.nlp.fewshot.model.PrototypicalTextClassifier method*), 239
forward() (*flambe.nlp.fewshot.PrototypicalTextClassifier method*), 240
forward() (*flambe.nlp.language_modeling.LanguageModel method*), 244
forward() (*flambe.nlp.language_modeling.model.LanguageModel method*), 241
forward() (*flambe.nlp.transformers.model.PretrainedTransformerEmbed method*), 246
forward() (*flambe.nlp.transformers.PretrainedTransformerEmbedder method*), 247
forward() (*flambe.nn.AvgPooling method*), 275
forward() (*flambe.nn.cnn.CNNEncoder method*), 254
forward() (*flambe.nn.CNNEncoder method*), 273
forward() (*flambe.nn.distance.cosine.CosineDistance method*), 249
forward() (*flambe.nn.distance.cosine.CosineMean method*), 249
forward() (*flambe.nn.distance.CosineDistance method*), 253
forward() (*flambe.nn.distance.CosineMean method*), 253
forward() (*flambe.nn.distance.distance.DistanceModule method*), 250
forward() (*flambe.nn.distance.distance.MeanModule method*), 250
forward() (*flambe.nn.distance.DistanceModule method*), 252
forward() (*flambe.nn.distance.euclidean.EuclideanDistance method*), 250
forward() (*flambe.nn.distance.euclidean.EuclideanMean method*), 251
forward() (*flambe.nn.distance.EuclideanDistance method*), 252
forward() (*flambe.nn.distance.EuclideanMean method*), 252
forward() (*flambe.nn.distance.hyperbolic.HyperbolicDistance method*), 251
forward() (*flambe.nn.distance.hyperbolic.HyperbolicMean method*), 252
forward() (*flambe.nn.distance.HyperbolicDistance method*), 253
forward() (*flambe.nn.distance.HyperbolicMean method*), 253
forward() (*flambe.nn.distance.MeanModule method*), 252
forward() (*flambe.nn.Embedder method*), 272
forward() (*flambe.nn.embedding.Embedder method*), 256

forward() (*flambe.nn.embedding.Embeddings method*), 256
forward() (*flambe.nn.Embeddings method*), 271
forward() (*flambe.nn.FirstPooling method*), 274
forward() (*flambe.nn.LastPooling method*), 274
forward() (*flambe.nn.MixtureOfSoftmax method*), 270
forward() (*flambe.nn.mlp.MLPEncoder method*), 257
forward() (*flambe.nn.MLPEncoder method*), 272
forward() (*flambe.nn.mos.MixtureOfSoftmax method*), 258
forward() (*flambe.nn.PooledRNNEncoder method*), 273
forward() (*flambe.nn.pooling.AvgPooling method*), 259
forward() (*flambe.nn.pooling.FirstPooling method*), 258
forward() (*flambe.nn.pooling.LastPooling method*), 259
forward() (*flambe.nn.pooling.SumPooling method*), 259
forward() (*flambe.nn.rnn.PooledRNNEncoder method*), 261
forward() (*flambe.nn.rnn.RNNEncoder method*), 260
forward() (*flambe.nn.RNNEncoder method*), 273
forward() (*flambe.nn.Sequential method*), 274
forward() (*flambe.nn.sequential.Sequential method*), 261
forward() (*flambe.nn.softmax.SoftmaxLayer method*), 262
forward() (*flambe.nn.SoftmaxLayer method*), 270
forward() (*flambe.nn.SumPooling method*), 274
forward() (*flambe.nn.Transformer method*), 275
forward() (*flambe.nn.transformer.Transformer method*), 262
forward() (*flambe.nn.transformer.TransformerDecoder method*), 263
forward() (*flambe.nn.transformer.TransformerDecoderLayer method*), 265
forward() (*flambe.nn.transformer.TransformerEncoder method*), 263
forward() (*flambe.nn.transformer.TransformerEncoderLayer method*), 264
forward() (*flambe.nn.transformer_sru.TransformerSRU method*), 265
forward() (*flambe.nn.transformer_sru.TransformerSRUDecoder method*), 267
forward() (*flambe.nn.transformer_sru.TransformerSRUDecoderLayer method*), 268
forward() (*flambe.nn.transformer_sru.TransformerSRUEncoder method*), 266
forward() (*flambe.nn.transformer_sru.TransformerSRUEncoderLayer method*), 268
forward() (*flambe.nn.TransformerDecoder method*), 276
forward() (*flambe.nn.TransformerEncoder method*), 276
forward() (*flambe.nn.TransformerSRU method*), 277
forward() (*flambe.nn.TransformerSRUDecoder method*), 278
forward() (*flambe.nn.TransformerSRUEncoder method*), 278
forward() (*flambe.vision.classification.ImageClassifier method*), 310
forward() (*flambe.vision.classification.model.ImageClassifier method*), 308
from_path() (*flambe.dataset.tabular.TabularDataset class method*), 97
from_path() (*flambe.dataset.TabularDataset class method*), 100
from_path() (*flambe.vision.classification.datasets.MNISTDataset class method*), 308
from_path() (*flambe.vision.classification.MNISTDataset class method*), 309
from_sequence() (*flambe.experiment.GridSearchOptions class method*), 185
from_sequence() (*flambe.experiment.options.GridSearchOptions class method*), 177
from_sequence() (*flambe.experiment.options.Options class method*), 176
from_sequence() (*flambe.experiment.options.SampledUniformSearchO class method*), 177
from_sequence() (*flambe.experiment.SampledUniformSearchOptions class method*), 185
from_yaml() (*flambe.compile.Component class method*), 167
from_yaml() (*flambe.compile.component.Component class method*), 151
from_yaml() (*flambe.compile.component.Link class method*), 147
from_yaml() (*flambe.compile.component.PickledDataLink class method*), 145
from_yaml() (*flambe.compile.Link class method*), 168
from_yaml() (*flambe.compile.MappedRegistrable class method*), 162
from_yaml() (*flambe.compile.Registrable class method*), 161
from_yaml() (*flambe.compile.registrable.MappedRegistrable class method*), 157
from_yaml() (*flambe.compile.registrable.Registrable class method*), 156
from_yaml() (*flambe.experiment.options.ClusterResource class method*), 177
from_yaml() (*flambe.experiment.options.Options class method*), 176
from_yaml() (*flambe.runnable.environment.RemoteEnvironment class method*), 284
from_yaml() (*flambe.runnable.RemoteEnvironment class method*), 291

FunctionCallLink (*class in flambe.compile.component*), 147

G

generate_square_subsequent_mask() (*in module flambe.nn.transformer*), 265
get_best_trials() (*in module flambe.experiment.utils*), 181
get_boto_session() (*flambe.export.Builder method*), 188
get_boto_session() (*flambe.export.builder.Builder method*), 187
get_commit_hash() (*in module flambe.runnable.utils*), 286
get_dataset() (*in module flambe.vision.classification.datasets*), 308
get_default_tag() (*flambe.compile.Registrable static method*), 161
get_default_tag() (*flambe.compile.registrable.Registrable static method*), 156
get_distance_module() (*in module flambe.nn.distance*), 254
get_factory() (*flambe.cluster.Cluster method*), 133
get_factory() (*flambe.cluster.cluster.Cluster method*), 127
get_factory_basename() (*flambe.cluster.Cluster method*), 132
get_factory_basename() (*flambe.cluster.cluster.Cluster method*), 126
get_flambe_repo_location() (*in module flambe.runnable.utils*), 286
get_gpu_factory() (*flambe.cluster.Cluster method*), 133
get_gpu_factory() (*flambe.cluster.cluster.Cluster method*), 127
get_home_path() (*flambe.cluster.instance.Instance method*), 114
get_home_path() (*flambe.cluster.instance.instance.Instance method*), 107
get_max_resources() (*flambe.cluster.Cluster method*), 135
get_max_resources() (*flambe.cluster.cluster.Cluster method*), 129
get_mean_module() (*in module flambe.nn.distance*), 254
get_nested() (*in module flambe.experiment.utils*), 180
get_non_remote_config() (*in module flambe.experiment.utils*), 181
get_orch_home_path() (*flambe.cluster.Cluster method*), 133
get_orch_home_path() (*flambe.cluster.cluster.Cluster method*), 127
get_orchestrator() (*flambe.cluster.Cluster method*), 133
get_orchestrator() (*flambe.cluster.cluster.Cluster method*), 126
get_orchestrator_name() (*flambe.cluster.Cluster method*), 132
get_orchestrator_name() (*flambe.cluster.cluster.Cluster method*), 125
get_remote_env() (*flambe.cluster.Cluster method*), 136
get_remote_env() (*flambe.cluster.cluster.Cluster method*), 129
get_size_MB() (*in module flambe.runner.utils*), 294
get_state() (*flambe.compile.Component method*), 165
get_state() (*flambe.compile.component.Component method*), 149
get_trial_dir() (*in module flambe.logging*), 221
get_trial_dir() (*in module flambe.logging.utils*), 213
get_user() (*flambe.experiment.Experiment method*), 184
get_user() (*flambe.experiment.experiment.Experiment method*), 176
global_step (*flambe.logging.datatypes.EmbeddingT attribute*), 210
global_step (*flambe.logging.datatypes.HistogramT attribute*), 208
global_step (*flambe.logging.datatypes.ImageT attribute*), 209
global_step (*flambe.logging.datatypes.PRCurveT attribute*), 209
global_step (*flambe.logging.datatypes.ScalarsT attribute*), 208
global_step (*flambe.logging.datatypes.ScalarT attribute*), 207
global_step (*flambe.logging.datatypes.TextT attribute*), 209
global_step (*flambe.logging.EmbeddingT attribute*), 218
global_step (*flambe.logging.HistogramT attribute*), 216
global_step (*flambe.logging.ImageT attribute*), 217
global_step (*flambe.logging.PRCurveT attribute*), 218
global_step (*flambe.logging.ScalarsT attribute*), 216
global_step (*flambe.logging.ScalarT attribute*), 215
global_step (*flambe.logging.TextT attribute*), 217
GPUFactoryInsT (*in module flambe.cluster.cluster*), 124
GPUFactoryInstance (*class in flambe.cluster.instance*), 117
GPUFactoryInstance (*class in flambe.cluster.instance.instance*), 109

gradient_norm (*flambe.nn.Module attribute*), 269
gradient_norm (*flambe.nn.module.Module attribute*), 257
GraphT (*class in flambe.logging*), 218
GraphT (*class in flambe.logging.datatypes*), 210
GridSearchOptions (*class in flambe.experiment*), 185
GridSearchOptions (*class in flambe.experiment.options*), 176

H

has_alarm() (*flambe.cluster.aws.AWSCluster method*), 123
has_alarm() (*flambe.cluster.AWSCluster method*), 140
has_schemas_or_options() (*in module flambe.experiment.utils*), 180
HIGHEST_SERIALIZATION_PROTOCOL_VERSION (*in module flambe.compile.const*), 152
HistogramT (*class in flambe.logging*), 216
HistogramT (*class in flambe.logging.datatypes*), 208
http_exists() (*in module flambe.compile.downloader*), 153
HyperbolicDistance (*class in flambe.nn.distance*), 253
HyperbolicDistance (*class in flambe.nn.distance.hyperbolic*), 251
HyperbolicMean (*class in flambe.nn.distance*), 253
HyperbolicMean (*class in flambe.nn.distance.hyperbolic*), 251

I

ImageClassifier (*class in flambe.vision.classification*), 309
ImageClassifier (*class in flambe.vision.classification.model*), 308
ImageT (*class in flambe.logging*), 217
ImageT (*class in flambe.logging.datatypes*), 208
img_tensor (*flambe.logging.datatypes.ImageT attribute*), 208
img_tensor (*flambe.logging.ImageT attribute*), 217
import_modules() (*in module flambe.compile.extensions*), 155
index() (*in module flambe.experiment.webapp.app*), 174
inject_content() (*flambe.runnable.Runnable method*), 287
inject_content() (*flambe.runnable.runnable.Runnable method*), 285
inject_extensions() (*flambe.runnable.Runnable method*), 288
inject_extensions() (*flambe.runnable.runnable.Runnable method*), 286
inject_secrets() (*flambe.runnable.Runnable method*), 287
inject_secrets() (*flambe.runnable.runnable.Runnable method*), 286
input_size (*flambe.model.logistic_regression.LogisticRegression attribute*), 233
input_size (*flambe.model.LogisticRegression attribute*), 233
input_to_model (*flambe.logging.datatypes.GraphT attribute*), 210, 211
input_to_model (*flambe.logging.GraphT attribute*), 218, 219
InsT (*in module flambe.cluster.instance.instance*), 104
install_cuda() (*flambe.cluster.instance.GPUFactoryInstance method*), 117
install_cuda() (*flambe.cluster.instance.instance.GPUFactoryInstance method*), 109
install_docker() (*flambe.cluster.instance.Instance method*), 114
install_docker() (*flambe.cluster.instance.instance.Instance method*), 107
install_extensions() (*flambe.cluster.instance.Instance method*), 114
install_extensions() (*flambe.cluster.instance.instance.Instance method*), 107
install_extensions() (*in module flambe.compile.extensions*), 154
install_extensions_in_factories() (*flambe.cluster.Cluster method*), 136
install_extensions_in_factories() (*flambe.cluster.cluster.Cluster method*), 129
install_extensions_in_orchestrator() (*flambe.cluster.Cluster method*), 136
install_extensions_in_orchestrator() (*flambe.cluster.cluster.Cluster method*), 129
install_flambe() (*flambe.cluster.instance.Instance method*), 115
install_flambe() (*flambe.cluster.instance.instance.Instance method*), 107
Instance (*class in flambe.cluster.instance*), 111
Instance (*class in flambe.cluster.instance.instance*), 104
is_cuda_installed() (*flambe.cluster.instance.GPUFactoryInstance method*), 117
is_cuda_installed() (*flambe.cluster.instance.instance.GPUFactoryInstance method*), 109
is_dev_mode() (*in module flambe.runnable.utils*), 286
is_docker_installed() (*flambe.cluster.instance.Instance method*),

115
is_docker_installed() (flambe.cluster.instance.instance.Instance method), 107
is_docker_running() (flambe.cluster.instance.Instance method), 115
is_docker_running() (flambe.cluster.instance.instance.Instance method), 108
is_empty() (flambe.dataset.tabular.DataView method), 96
is_flambe_installed() (flambe.cluster.instance.Instance method), 115
is_flambe_installed() (flambe.cluster.instance.instance.Instance method), 107
is_flambe_running() (flambe.cluster.instance.Instance method), 115
is_flambe_running() (flambe.cluster.instance.instance.Instance method), 108
is_installed_module() (in module flambe.compile.extensions), 155
is_node_running() (flambe.cluster.instance.Instance method), 115
is_node_running() (flambe.cluster.instance.instance.Instance method), 108
is_ray_cluster_up() (flambe.cluster.Cluster method), 134
is_ray_cluster_up() (flambe.cluster.cluster.Cluster method), 127
is_report_site_running() (flambe.cluster.instance.instance.OrchestratorInstance method), 110
is_report_site_running() (flambe.cluster.instance.OrchestratorInstance method), 117
is_tensorboard_running() (flambe.cluster.instance.instance.OrchestratorInstance method), 110
is_tensorboard_running() (flambe.cluster.instance.OrchestratorInstance method), 117
is_up() (flambe.cluster.instance.Instance method), 112
is_up() (flambe.cluster.instance.instance.Instance method), 105

K

K (in module flambe.compile.component), 147
KEEP_VARS_KEY (in module flambe.compile.const), 152
key (flambe.runnable.environment.RemoteEnvironment attribute), 284
key (flambe.runnable.RemoteEnvironment attribute), 290
keywords (flambe.compile.component.Schema attribute), 144
keywords (flambe.compile.Schema attribute), 163
kill_tmux_session() (flambe.cluster.instance.instance.OrchestratorInstance method), 110
kill_tmux_session() (flambe.cluster.instance.OrchestratorInstance method), 118
kwargs (flambe.logging.datatypes.GraphT attribute), 210, 211
kwargs (flambe.logging.GraphT attribute), 219

L

label_count (flambe.field.label.LabelField attribute), 192
label_count (flambe.field.LabelField attribute), 196
label_freq (flambe.field.label.LabelField attribute), 193
label_freq (flambe.field.LabelField attribute), 196
label_img (flambe.logging.datatypes.EmbeddingT attribute), 210
label_img (flambe.logging.EmbeddingT attribute), 218
label_inv_freq (flambe.field.label.LabelField attribute), 193
label_inv_freq (flambe.field.LabelField attribute), 196
LabelField (class in flambe.field), 196
LabelField (class in flambe.field.label), 192
labels (flambe.logging.datatypes.PRCurveT attribute), 209
labels (flambe.logging.PRCurveT attribute), 217
LabelTokenizer (class in flambe.tokenizer), 305
LabelTokenizer (class in flambe.tokenizer.label), 301
LanguageModel (class in flambe.nlp.language_modeling), 244
LanguageModel (class in flambe.nlp.language_modeling.model), 241
LastPooling (class in flambe.nn), 274
LastPooling (class in flambe.nn.pooling), 259
launch_flambe() (flambe.cluster.instance.instance.OrchestratorInstance method), 111
launch_flambe() (flambe.cluster.instance.OrchestratorInstance method), 118
launch_node() (flambe.cluster.instance.CPUFactoryInstance method), 116

launch_node() (*flambe.cluster.instance.instance.CPUFactoryInstance method*), 109
launch_node() (*flambe.cluster.instance.instance.OrchestratorInstance method*), 111
launch_node() (*flambe.cluster.instance.OrchestratorInstance method*), 118
launch_ray_cluster() (*flambe.cluster.Cluster method*), 133
launch_ray_cluster() (*flambe.cluster.cluster.Cluster method*), 127
launch_report_site() (*flambe.cluster.instance.instance.OrchestratorInstance method*), 110
launch_report_site() (*flambe.cluster.instance.OrchestratorInstance method*), 117
launch_tensorboard() (*flambe.cluster.instance.instance.OrchestratorInstance method*), 110
launch_tensorboard() (*flambe.cluster.instance.OrchestratorInstance method*), 118
launch_tensorboard() (*in module flambe.runner.report_site_run*), 293
layers (*flambe.nn.MixtureOfSoftmax attribute*), 270
layers (*flambe.nn.mos.MixtureOfSoftmax attribute*), 258
length() (*flambe.nlp.language_modeling.CorpusSampler method*), 244
length() (*flambe.nlp.language_modeling.sampler.CorpusSampler method*), 242
length() (*flambe.sampler.base.BaseSampler method*), 296
length() (*flambe.sampler.BaseSampler method*), 299
length() (*flambe.sampler.episodic.EpisodicSampler method*), 297
length() (*flambe.sampler.EpisodicSampler method*), 299
length() (*flambe.sampler.Sampler method*), 298
length() (*flambe.sampler.sampler.Sampler method*), 298
level (*flambe.logging.datatypes.DataLoggingFilter attribute*), 211
Link (*class in flambe.compile*), 168
Link (*class in flambe.compile.component*), 146
LinkError, 145, 284
LMField (*class in flambe.nlp.language_modeling*), 243
LMField (*class in flambe.nlp.language_modeling.fields*), 241
load() (*in module flambe.compile*), 170
load() (*in module flambe.compile.serialization*), 159
load_all_instances() (*flambe.cluster.aws.AWSCluster method*), 120
load_all_instances() (*flambe.cluster.AWSCluster method*), 138
load_all_instances() (*flambe.cluster.Cluster method*), 132
load_all_instances() (*flambe.cluster.cluster.Cluster method*), 126
load_all_instances() (*flambe.cluster.ssh.SSHCluster method*), 131
load_all_instances() (*flambe.cluster.SSHCluster method*), 142
load_from_path() (*flambe.compile.Component class method*), 167
load_from_path() (*flambe.compile.component.Component class method*), 151
load_state() (*flambe.compile.Component method*), 167
load_state() (*flambe.compile.component.Component method*), 150
load_state() (*in module flambe.experiment.webapp.app*), 173
load_state_from_file() (*in module flambe.compile*), 171
load_state_from_file() (*in module flambe.compile.serialization*), 159
LoadError, 143, 157
local_has_gpu() (*in module flambe.experiment.utils*), 181
local_user (*flambe.runnable.environment.RemoteEnvironment attribute*), 284
local_user (*flambe.runnable.RemoteEnvironment attribute*), 290
log() (*in module flambe.logging*), 219
log() (*in module flambe.logging.utils*), 213
log_histogram() (*in module flambe.logging*), 220
log_histogram() (*in module flambe.logging.utils*), 214
log_image() (*in module flambe.logging*), 220
log_image() (*in module flambe.logging.utils*), 214
log_map() (*in module flambe.nn.distance.hyperbolic*), 251
log_pr_curve() (*in module flambe.logging*), 220
log_pr_curve() (*in module flambe.logging.utils*), 214
log_scalar() (*in module flambe.logging*), 219
log_scalar() (*in module flambe.logging.utils*), 213
log_scalars() (*in module flambe.logging*), 219
log_scalars() (*in module flambe.logging.utils*), 213
log_text() (*in module flambe.logging*), 220
log_text() (*in module flambe.logging.utils*), 214
logger (*in module flambe.cluster.aws*), 119
logger (*in module flambe.cluster.cluster*), 124
logger (*in module flambe.cluster.instance.instance*), 104

logger (*in module flambe.cluster.ssh*), 130
logger (*in module flambe.compile.component*), 143
logger (*in module flambe.compile.downloader*), 152
logger (*in module flambe.compile.extensions*), 154
logger (*in module flambe.compile.registrable*), 155
logger (*in module flambe.compile.serialization*), 157
logger (*in module flambe.experiment.experiment*), 174
logger (*in module flambe.experiment.wording*), 182
logger (*in module flambe.export.builder*), 187
logger (*in module flambe.nn.rnn*), 260
logger (*in module flambe.runnable.context*), 282
logger (*in module flambe.runner.report_site_run*), 293
logger (*in module flambe.runner.utils*), 294
LogisticRegression (*class in flambe.model*), 233
LogisticRegression (*class in flambe.model.logistic_regression*), 233
loss (*flambe.nlp.classification.model.TextClassifier attribute*), 236
loss (*flambe.nlp.classification.TextClassifier attribute*), 237
loss (*flambe.nlp.fewshot.model.PrototypicalTextClassifier attribute*), 238
loss (*flambe.nlp.fewshot.PrototypicalTextClassifier attribute*), 239
loss() (*in module flambe.nn.distance.hyperbolic*), 251

M

main() (*in module flambe.runner.run*), 293
main_tag (*flambe.logging.datatypes.ScalarsT attribute*), 208
main_tag (*flambe.logging.ScalarsT attribute*), 216
make_component() (*in module flambe.compile*), 169
make_component() (*in module flambe.compile.utils*), 160
make_from_yaml_with_metadata() (*in module flambe.compile.registrable*), 155
make_to_yaml_with_metadata() (*in module flambe.compile.registrable*), 155
MalformedLinkError, 145
MappedRegistrable (*class in flambe.compile*), 162
MappedRegistrable (*class in flambe.compile.registrable*), 157
mat (*flambe.logging.datatypes.EmbeddingT attribute*), 210
mat (*flambe.logging.EmbeddingT attribute*), 218
MB (*in module flambe.logging.logging*), 212
MB (*in module flambe.runner.utils*), 294
mdot() (*in module flambe.nn.distance.hyperbolic*), 251
MeanModule (*class in flambe.nn.distance*), 252
MeanModule (*class in flambe.nn.distance.distance*), 250
merge_kwargs() (*in module flambe.compile.component*), 147
metadata (*flambe.logging.datatypes.EmbeddingT attribute*), 210
metadata (*flambe.logging.EmbeddingT attribute*), 218
metadata_header (*flambe.logging.datatypes.EmbeddingT attribute*), 210
metadata_header (*flambe.logging.EmbeddingT attribute*), 218
Metric (*class in flambe.metric*), 228
Metric (*class in flambe.metric.metric*), 227
metric (*flambe.nlp.classification.model.TextClassifier attribute*), 236
metric (*flambe.nlp.classification.TextClassifier attribute*), 238
metric (*flambe.nlp.fewshot.model.PrototypicalTextClassifier attribute*), 238
metric (*flambe.nlp.fewshot.PrototypicalTextClassifier attribute*), 239
metric() (*flambe.compile.Component method*), 164
metric() (*flambe.compile.component.Component method*), 148
metric() (*flambe.learn.eval.Evaluator method*), 199
metric() (*flambe.learn.Evaluator method*), 202
metric() (*flambe.learn.train.Trainer method*), 200
metric() (*flambe.learn.Trainer method*), 202
MissingAuthError, 103
MissingSecretsError, 285
MixtureOfSoftmax (*class in flambe.nn*), 270
MixtureOfSoftmax (*class in flambe.nn.mos*), 258
MLPEncoder (*class in flambe.nn*), 272
MLPEncoder (*class in flambe.nn.mlp*), 257
MNISTDataset (*class in flambe.vision.classification*), 309
MNISTDataset (*class in flambe.vision.classification.datasets*), 307
mode (*flambe.logging.handler.contextual_file.ContextualFileHandler attribute*), 206
model (*flambe.logging.datatypes.GraphT attribute*), 210, 211
model (*flambe.logging.GraphT attribute*), 218, 219
Module (*class in flambe.nn*), 269
Module (*class in flambe.nn.module*), 257
MultiLabelCrossEntropy (*class in flambe.metric*), 228
MultiLabelCrossEntropy (*class in flambe.metric.loss.cross_entropy*), 227
MultiLabelNLLLoss (*class in flambe.metric*), 229
MultiLabelNLLLoss (*class in flambe.metric.loss.nll_loss*), 227

N

name_hosts() (*flambe.cluster.aws.AWSCluster method*), 121
name_hosts() (*flambe.cluster.AWSCluster method*), 138

name_instance() (*flambe.cluster.aws.AWSCluster method*), 121
name_instance() (*flambe.cluster.AWSCluster method*), 138
named_trainable_params (*flambe.nn.Module attribute*), 269
named_trainable_params (*flambe.nn.module.Module attribute*), 257
NewsGroupDataset (*class in flambe.nlp.classification*), 237
NewsGroupDataset (*class in flambe.nlp.classification.datasets*), 236
NGramsTokenizer (*class in flambe.tokenizer*), 304
NGramsTokenizer (*class in flambe.tokenizer.word*), 303
NLTKWordTokenizer (*class in flambe.tokenizer*), 304
NLTKWordTokenizer (*class in flambe.tokenizer.word*), 303
NonExistentResourceError, 285
norm() (*in module flambe.nn.distance.hyperbolic*), 251
num_cpus() (*flambe.cluster.instance.CPUFactoryInstance method*), 116
num_cpus() (*flambe.cluster.instance.instance.CPUFactoryInstance method*), 109
num_gpus() (*flambe.cluster.instance.CPUFactoryInstance method*), 116
num_gpus() (*flambe.cluster.instance.instance.CPUFactoryInstance method*), 109
num_parameters() (*flambe.nn.Module method*), 269
num_parameters() (*flambe.nn.module.Module method*), 258
num_thresholds (*flambe.logging.datatypes.PRCurveT attribute*), 209
num_thresholds (*flambe.logging.PRCurveT attribute*), 218
Number (*in module flambe.experiment.options*), 176

O

object_stash (*flambe.compile.serialization.SaveTreeNode attribute*), 157
OptionalSearchAlgorithms (*in module flambe.experiment.experiment*), 174
OptionalTrialSchedulers (*in module flambe.experiment.experiment*), 174
Options (*class in flambe.experiment.options*), 176
orchestrator_ip (*flambe.runnable.environment.RemoteEnvironment attribute*), 284
orchestrator_ip (*flambe.runnable.RemoteEnvironment attribute*), 290
OrchestratorInstance (*class in flambe.cluster.instance*), 117
OrchestratorInstance (*class in flambe.cluster.instance.instance*), 110
output_layer (*flambe.nlp.classification.model.TextClassifier attribute*), 236
output_layer (*flambe.nlp.classification.TextClassifier attribute*), 237
output_layer (*flambe.vision.classification.ImageClassifier attribute*), 310
output_layer (*flambe.vision.classification.model.ImageClassifier attribute*), 308

P

padding_idx (*flambe.nlp.transformers.field.PretrainedTransformerField attribute*), 245
padding_idx (*flambe.nlp.transformers.PretrainedTransformerField attribute*), 247
parameter_norm (*flambe.nn.Module attribute*), 269
parameter_norm (*flambe.nn.module.Module attribute*), 257
parse() (*flambe.cluster.aws.AWSCluster method*), 122
parse() (*flambe.cluster.AWSCluster method*), 140
parse() (*flambe.cluster.Cluster method*), 134
parse() (*flambe.cluster.cluster.Cluster method*), 128
parse() (*flambe.experiment.Experiment method*), 184
parse() (*flambe.experiment.experiment.Experiment method*), 176
parse() (*flambe.runnable.Runnable method*), 288
parse() (*flambe.runnable.runnable.Runnable method*), 286
parse_link_str() (*in module flambe.compile.component*), 145
parser (*in module flambe.runner.report_site_run*), 293
parser (*in module flambe.runner.run*), 293
ParsingRunnableError, 285
Perplexity (*class in flambe.metric*), 229
Perplexity (*class in flambe.metric.dev.perplexity*), 226
pi (*flambe.nn.MixtureOfSoftmax attribute*), 270
pi (*flambe.nn.mos.MixtureOfSoftmax attribute*), 258
PickledDataLink (*class in flambe.compile.component*), 145
PooledRNNEncoder (*class in flambe.nn*), 273
PooledRNNEncoder (*class in flambe.nn.rnn*), 260
pooling (*flambe.nn.Embedder attribute*), 272
pooling (*flambe.nn.embedding.Embedder attribute*), 256
PRCurveT (*class in flambe.logging*), 217
PRCurveT (*class in flambe.logging.datatypes*), 209
precompile() (*flambe.compile.Component class method*), 167
precompile() (*flambe.compile.component.Component class method*), 151
precompile() (*flambe.learn.train.Trainer class method*), 201
precompile() (*flambe.learn.Trainer class method*), 202

predictions (*flambe.logging.datatypes.PRCurveT attribute*), 209
predictions (*flambe.logging.PRCurveT attribute*), 218
prepare() (*flambe.cluster.instance.CPUFactoryInstance method*), 116
prepare() (*flambe.cluster.instance.GPUFactoryInstance method*), 117
prepare() (*flambe.cluster.instance.Instance method*), 112
prepare() (*flambe.cluster.instance.instance.CPUFactoryInstance method*), 109
prepare() (*flambe.cluster.instance.instance.GPUFactoryInstance method*), 109
prepare() (*flambe.cluster.instance.instance.Instance method*), 105
prepare() (*flambe.cluster.instance.instance.OrchestratorInstance method*), 110
prepare() (*flambe.cluster.instance.OrchestratorInstance method*), 117
prepare_all_instances() (*flambe.cluster.Cluster method*), 133
prepare_all_instances() (*flambe.cluster.cluster.Cluster method*), 126
preprocess() (*flambe.runnable.context.SafeExecutionContext method*), 282
preprocess() (*flambe.runnable.SafeExecutionContext method*), 289
PretrainedTransformerEmbedder (*class in flambe.nlp.transformers*), 247
PretrainedTransformerEmbedder (*class in flambe.nlp.transformers.model*), 246
PretrainedTransformerField (*class in flambe.nlp.transformers*), 247
PretrainedTransformerField (*class in flambe.nlp.transformers.field*), 245
print_extensions_cache_size_warning() (*in module flambe.experiment.wording*), 182
print_useful_local_info() (*in module flambe.experiment.wording*), 182
print_useful_metrics_only_info() (*in module flambe.experiment.wording*), 182
print_useful_remote_info() (*in module flambe.experiment.wording*), 182
PRIVATE_KEY (*in module flambe.cluster.const*), 130
process() (*flambe.field.bow.BoWField method*), 192
process() (*flambe.field.BoWField method*), 195
process() (*flambe.field.Field method*), 194
process() (*flambe.field.field.Field method*), 192
process() (*flambe.field.label.LabelField method*), 193
process() (*flambe.field.LabelField method*), 196
process() (*flambe.field.text.TextField method*), 194
process() (*flambe.field.TextField method*), 195
process() (*flambe.nlp.language_modeling.fields.LMField method*), 241
process() (*flambe.nlp.language_modeling.LMField method*), 243
process() (*flambe.nlp.transformers.field.PretrainedTransformerField method*), 245
process() (*flambe.nlp.transformers.PretrainedTransformerField method*), 247
process_resources() (*flambe.experiment.Experiment method*), 183
process_resources() (*flambe.experiment.experiment.Experiment method*), 175
ProgressState (*class in flambe.experiment*), 184
ProgressState (*class in flambe.experiment.progress*), 177
project() (*in module flambe.nn.distance.hyperbolic*), 251
PROTOCOL_VERSION_FILE_NAME (*in module flambe.compile.const*), 152
ProtocolError, 284
PrototypicalTextClassifier (*class in flambe.nlp.fewshot*), 239
PrototypicalTextClassifier (*class in flambe.nlp.fewshot.model*), 238
PTB_URL (*flambe.nlp.language_modeling.datasets.PTBDataset attribute*), 240
PTB_URL (*flambe.nlp.language_modeling.PTBDataset attribute*), 243
PTBDataset (*class in flambe.nlp.language_modeling*), 243
PTBDataset (*class in flambe.nlp.language_modeling.datasets*), 240
public_factories_ips (*flambe.runnable.environment.RemoteEnvironment attribute*), 284
public_factories_ips (*flambe.runnable.RemoteEnvironment attribute*), 290
PUBLIC_KEY (*in module flambe.cluster.const*), 130
public_orchestrator_ip (*flambe.runnable.environment.RemoteEnvironment attribute*), 284
public_orchestrator_ip (*flambe.runnable.RemoteEnvironment attribute*), 290

R

R (*in module flambe.compile.registrable*), 155
raw (*flambe.dataset.tabular.DataView attribute*), 96
raw (*flambe.dataset.tabular.TabularDataset attribute*), 97
raw (*flambe.dataset.TabularDataset attribute*), 100

RAY_REDIS_PORT (*in module flambe.cluster.const*), 130
refresh() (*flambe.experiment.progress.ProgressState method*), 177
refresh() (*flambe.experiment.ProgressState method*), 184
register() (*in module flambe.compile*), 161
register() (*in module flambe.compile.registrable*), 156
register_attrs() (*flambe.compile.Component method*), 164
register_attrs() (*flambe.compile.component.Component method*), 148
register_tag() (*flambe.compile.Registrable static method*), 161
register_tag() (*flambe.compile.registrable.Registrable static method*), 156
Registrable (*class in flambe.compile*), 161
Registrable (*class in flambe.compile.registrable*), 156
registrable_factory (*class in flambe.compile*), 162
registrable_factory (*class in flambe.compile.registrable*), 156
registration_context (*class in flambe.compile*), 162
registration_context (*class in flambe.compile.registrable*), 155
RegistrationError, 155, 161
rel_to_abs_paths() (*in module flambe.experiment.utils*), 181
RemoteCommand (*in module flambe.cluster.utils*), 131
RemoteCommandError, 103
RemoteEnvironment (*class in flambe.runnable*), 290
RemoteEnvironment (*class in flambe.runnable.environment*), 283
RemoteFileTransferError, 103
remove_dir() (*flambe.cluster.Cluster method*), 135
remove_dir() (*flambe.cluster.cluster.Cluster method*), 128
remove_dir() (*flambe.cluster.instance.Instance method*), 116
remove_dir() (*flambe.cluster.instance.instance.Instance method*), 108
remove_existing_events() (*flambe.cluster.aws.AWSCluster method*), 123
remove_existing_events() (*flambe.cluster.AWSCluster method*), 140
remove_report_site() (*flambe.cluster.instance.instance.OrchestratorInstance method*), 110
remove_report_site() (*flambe.cluster.instance.OrchestratorInstance method*), 118
remove_tensorboard() (*flambe.cluster.instance.instance.OrchestratorInstance method*), 110
remove_tensorboard() (*flambe.cluster.instance.OrchestratorInstance method*), 117
REPORT_SITE_PORT (*in module flambe.cluster.const*), 130
ResourceError, 285
RETRIES (*in module flambe.cluster.const*), 130
RETRY_DELAY (*in module flambe.cluster.const*), 130
rnn (*flambe.nn.rnn.RNNEncoder attribute*), 260
rnn (*flambe.nn.RNNEncoder attribute*), 272
RNNEncoder (*class in flambe.nn*), 272
RNNEncoder (*class in flambe.nn.rnn*), 260
rollback_env() (*flambe.cluster.aws.AWSCluster method*), 122
rollback_env() (*flambe.cluster.AWSCluster method*), 140
rollback_env() (*flambe.cluster.Cluster method*), 134
rollback_env() (*flambe.cluster.cluster.Cluster method*), 127
rollback_env() (*flambe.cluster.ssh.SSHCluster method*), 131
rollback_env() (*flambe.cluster.SSHCluster method*), 142
root_schema (*flambe.compile.component.Link attribute*), 147
root_schema (*flambe.compile.Link attribute*), 168
rsync_folder() (*flambe.cluster.instance.instance.OrchestratorInstance method*), 111
rsync_folder() (*flambe.cluster.instance.OrchestratorInstance method*), 118
rsync_hosts() (*flambe.cluster.ssh.SSHCluster method*), 131
rsync_hosts() (*flambe.cluster.SSHCluster method*), 142
rsync_hosts() (*in module flambe.runnable.utils*), 287
rsync_orch() (*flambe.cluster.Cluster method*), 134
rsync_orch() (*flambe.cluster.cluster.Cluster method*), 128
RT (*in module flambe.compile.registrable*), 155
run() (*flambe.cluster.Cluster method*), 133
run() (*flambe.cluster.cluster.Cluster method*), 126
run() (*flambe.compile.Component method*), 164
run() (*flambe.compile.component.Component method*), 148
run() (*flambe.experiment.Experiment method*), 183
run() (*flambe.experiment.experiment.Experiment method*), 175
run() (*flambe.export.Builder method*), 188

run() (*flambe.export.builder.Builder method*), 187
run() (*flambe.export.Exporter method*), 189
run() (*flambe.export.exporter.Exporter method*), 188
run() (*flambe.learn.eval.Evaluator method*), 199
run() (*flambe.learn.Evaluator method*), 202
run() (*flambe.learn.Script method*), 203
run() (*flambe.learn.script.Script method*), 199
run() (*flambe.learn.train.Trainer method*), 200
run() (*flambe.learn.Trainer method*), 202
run() (*flambe.runnable.Runnable method*), 288
run() (*flambe.runnable.runnable.Runnable method*), 286
run_cmds() (*flambe.cluster.Cluster method*), 133
run_cmds() (*flambe.cluster.cluster.Cluster method*), 126
run_cmds() (*flambe.cluster.instance.Instance method*), 114
run_cmds() (*flambe.cluster.instance.instance.Instance method*), 106
Runnable (*class in flambe.runnable*), 287
Runnable (*class in flambe.runnable.runnable*), 285
RunnableFileError, 285

S

s3_exists() (*in module flambe.compile.downloader*), 152
s3_remote_file() (*in module flambe.compile.downloader*), 152
SafeExecutionContext (*class in flambe.runnable*), 289
SafeExecutionContext (*class in flambe.runnable.context*), 282
sample() (*flambe.nlp.language_modeling.CorpusSampler method*), 244
sample() (*flambe.nlp.language_modeling.sampler.CorpusSampler method*), 242
sample() (*flambe.sampler.base.BaseSampler method*), 296
sample() (*flambe.sampler.BaseSampler method*), 299
sample() (*flambe.sampler.episodic.EpisodicSampler method*), 297
sample() (*flambe.sampler.EpisodicSampler method*), 299
sample() (*flambe.sampler.Sampler method*), 298
sample() (*flambe.sampler.sampler.Sampler method*), 298
SampledUniformSearchOptions (*class in flambe.experiment*), 185
SampledUniformSearchOptions (*class in flambe.experiment.options*), 177
Sampler (*class in flambe.sampler*), 298
Sampler (*class in flambe.sampler.sampler*), 297
save() (*flambe.compile.Component method*), 167
save() (*flambe.compile.component.Component method*), 151
save() (*flambe.experiment.tune_adapter.TuneAdapter method*), 178
save() (*flambe.experiment.TuneAdapter method*), 184
save() (*in module flambe.compile*), 169
save() (*in module flambe.compile.serialization*), 159
save_local() (*flambe.export.Builder method*), 188
save_local() (*flambe.export.builder.Builder method*), 187
save_s3() (*flambe.export.Builder method*), 188
save_s3() (*flambe.export.builder.Builder method*), 187
save_state_to_file() (*in module flambe.compile*), 170
save_state_to_file() (*in module flambe.compile.serialization*), 158
SaveTreeNode (*class in flambe.compile.serialization*), 157
scalar_value (*flambe.logging.datatypes.ScalarT attribute*), 207
scalar_value (*flambe.logging.ScalarT attribute*), 215
ScalarsT (*class in flambe.logging*), 216
ScalarsT (*class in flambe.logging.datatypes*), 207
ScalarT (*class in flambe.logging*), 215
ScalarT (*class in flambe.logging.datatypes*), 207
Schema (*class in flambe.compile*), 162
Schema (*class in flambe.compile.component*), 143
Script (*class in flambe.learn*), 202
Script (*class in flambe.learn.script*), 199
SearchComponentError, 284
select_device() (*in module flambe.learn.utils*), 201
send_local_content() (*flambe.cluster.Cluster method*), 134
send_local_content() (*flambe.cluster.cluster.Cluster method*), 128
send_rsync() (*flambe.cluster.instance.Instance method*), 114
send_rsync() (*flambe.cluster.instance.instance.Instance method*), 106
send_secrets() (*flambe.cluster.Cluster method*), 135
send_secrets() (*flambe.cluster.cluster.Cluster method*), 128
seq (*flambe.nn.mlp.MLPEncoder attribute*), 257
seq (*flambe.nn.MLPEncoder attribute*), 272
Sequential (*class in flambe.nn*), 274
Sequential (*class in flambe.nn.sequential*), 261
serialize() (*flambe.compile.component.Schema static method*), 145
serialize() (*flambe.compile.Schema static method*), 163
set_serializable_attr() (*flambe.runnable.cluster_runnable.ClusterRunnable*

method), 282
set_serializable_attr() (*flambe.runnable.ClusterRunnable method*), 289
setup() (*flambe.experiment.Experiment method*), 183
setup() (*flambe.experiment.experiment.Experiment method*), 175
setup() (*flambe.field.bow.BoWField method*), 192
setup() (*flambe.field.BoWField method*), 195
setup() (*flambe.field.Field method*), 194
setup() (*flambe.field.field.Field method*), 192
setup() (*flambe.field.label.LabelField method*), 193
setup() (*flambe.field.LabelField method*), 196
setup() (*flambe.field.text.TextField method*), 194
setup() (*flambe.field.TextField method*), 195
setup() (*flambe.runnable.cluster_runnable.ClusterRunnable method*), 282
setup() (*flambe.runnable.ClusterRunnable method*), 288
setup_default_modules() (*in module flambe.compile.extensions*), 155
setup_global_logging() (*in module flambe.logging*), 215
setup_global_logging() (*in module flambe.logging.logging*), 212
setup_inject_env() (*flambe.runnable.cluster_runnable.ClusterRunnable method*), 282
setup_inject_env() (*flambe.runnable.ClusterRunnable method*), 288
shutdown_flambe() (*flambe.cluster.instance.Instance method*), 116
shutdown_flambe() (*flambe.cluster.instance.instance.Instance method*), 108
shutdown_flambe_execution() (*flambe.cluster.Cluster method*), 134
shutdown_flambe_execution() (*flambe.cluster.cluster.Cluster method*), 127
shutdown_node() (*flambe.cluster.instance.Instance method*), 116
shutdown_node() (*flambe.cluster.instance.instance.Instance method*), 108
shutdown_ray_cluster() (*flambe.cluster.Cluster method*), 134
shutdown_ray_cluster() (*flambe.cluster.cluster.Cluster method*), 127
shutdown_ray_node() (*in module flambe.experiment.utils*), 181
shutdown_remote_ray_node() (*in module flambe.experiment.utils*), 182
SOCKET_TIMEOUT (*in module flambe.cluster.const*), 130
SoftmaxLayer (*class in flambe.nn*), 269
SoftmaxLayer (*class in flambe.nn.softmax*), 261
source_code (*flambe.compile.serialization.SaveTreeNode attribute*), 157
SOURCE_FILE_NAME (*in module flambe.compile.const*), 152
SSHCluster (*class in flambe.cluster*), 142
SSHCluster (*class in flambe.cluster.ssh*), 130
SSHConnectingError, 103
SSTDataset (*class in flambe.nlp.classification*), 237
SSTDataset (*class in flambe.nlp.classification.datasets*), 235
start_docker() (*flambe.cluster.instance.Instance method*), 115
start_docker() (*flambe.cluster.instance.instance.Instance method*), 108
STASH_FILE_NAME (*in module flambe.compile.const*), 152
State (*class in flambe.compile*), 171
State (*class in flambe.compile.serialization*), 157
state (*flambe.compile.serialization.SaveTreeNode attribute*), 157
state() (*in module flambe.experiment.webapp.app*), 174
STATE_DICT_DELIMETER (*in module flambe.compile.const*), 152
STATE_FILE_NAME (*in module flambe.compile.const*), 152
stream (*flambe.logging.handler.contextual_file.ContextualFileHandler attribute*), 206
stream() (*in module flambe.experiment.webapp.app*), 174
SumPooling (*class in flambe.nn*), 274
SumPooling (*class in flambe.nn.pooling*), 259

T

T (*in module flambe.cluster.aws*), 119
TabularDataset (*class in flambe.dataset*), 99
TabularDataset (*class in flambe.dataset.tabular*), 96
tag (*flambe.logging.datatypes.EmbeddingT attribute*), 210
tag (*flambe.logging.datatypes.HistogramT attribute*), 208
tag (*flambe.logging.datatypes.ImageT attribute*), 208
tag (*flambe.logging.datatypes.PRCurveT attribute*), 209
tag (*flambe.logging.datatypes.ScalarT attribute*), 207
tag (*flambe.logging.datatypes.TextT attribute*), 209
tag (*flambe.logging.EmbeddingT attribute*), 218
tag (*flambe.logging.HistogramT attribute*), 216
tag (*flambe.logging.ImageT attribute*), 217
tag (*flambe.logging.PRCurveT attribute*), 217
tag (*flambe.logging.ScalarT attribute*), 215
tag (*flambe.logging.TextT attribute*), 217

tag_scalar_dict (*flambe.logging.datatypes.ScalarsT attribute*), 208
tag_scalar_dict (*flambe.logging.ScalarsT attribute*), 216
TagError, 285
teardown() (*flambe.experiment.Experiment method*), 183
teardown() (*flambe.experiment.experiment.Experiment method*), 175
TENSORBOARD_IMAGE (*in module flambe.cluster.const*), 130
TENSORBOARD_PORT (*in module flambe.cluster.const*), 130
TensorboardXHandler (*class in flambe.logging.handler.tensorboard*), 206
terminate_instances() (*flambe.cluster.aws.AWSCluster method*), 122
terminate_instances() (*flambe.cluster.AWSCluster method*), 139
test (*flambe.dataset.Dataset attribute*), 99
test (*flambe.dataset.dataset.Dataset attribute*), 95
test (*flambe.dataset.tabular.TabularDataset attribute*), 96, 97
test (*flambe.dataset.TabularDataset attribute*), 99
test (*flambe.vision.classification.datasets.MNISTDataset attribute*), 307
test (*flambe.vision.classification.MNISTDataset attribute*), 309
text_string (*flambe.logging.datatypes.TextT attribute*), 209
text_string (*flambe.logging.TextT attribute*), 217
TextClassifier (*class in flambe.nlp.classification*), 237
TextClassifier (*class in flambe.nlp.classification.model*), 236
TextField (*class in flambe.field*), 194
TextField (*class in flambe.field.text*), 193
TextT (*class in flambe.logging*), 216
TextT (*class in flambe.logging.datatypes*), 209
to_yaml() (*flambe.compile.Component class method*), 167
to_yaml() (*flambe.compile.component.Component class method*), 151
to_yaml() (*flambe.compile.component.Link class method*), 147
to_yaml() (*flambe.compile.component.PickledDataLink class method*), 145
to_yaml() (*flambe.compile.component.Schema class method*), 145
to_yaml() (*flambe.compile.Link class method*), 168
to_yaml() (*flambe.compile.MappedRegistrable class method*), 162
to_yaml() (*flambe.compile.Registrable class method*), 161
to_yaml() (*flambe.compile.registrable.MappedRegistrable class method*), 157
to_yaml() (*flambe.compile.registrable.Registrable class method*), 156
to_yaml() (*flambe.compile.Schema class method*), 163
to_yaml() (*flambe.experiment.options.ClusterResource class method*), 177
to_yaml() (*flambe.experiment.options.Options class method*), 176
to_yaml() (*flambe.experiment.options.SampledUniformSearchOptions class method*), 177
to_yaml() (*flambe.experiment.SampledUniformSearchOptions class method*), 185
to_yaml() (*flambe.runnable.environment.RemoteEnvironment class method*), 284
to_yaml() (*flambe.runnable.RemoteEnvironment class method*), 290
toJSON() (*flambe.experiment.progress.ProgressState method*), 177
toJSON() (*flambe.experiment.ProgressState method*), 184
tokenize() (*flambe.tokenizer.BPETokenizer method*), 305
tokenize() (*flambe.tokenizer.char.CharTokenizer method*), 301
tokenize() (*flambe.tokenizer.CharTokenizer method*), 304
tokenize() (*flambe.tokenizer.label.LabelTokenizer method*), 301
tokenize() (*flambe.tokenizer.LabelTokenizer method*), 305
tokenize() (*flambe.tokenizer.NGramsTokenizer method*), 305
tokenize() (*flambe.tokenizer.NLTKWordTokenizer method*), 304
tokenize() (*flambe.tokenizer.subword.BPETokenizer method*), 302
tokenize() (*flambe.tokenizer.Tokenizer method*), 304
tokenize() (*flambe.tokenizer.tokenizer.Tokenizer method*), 302
tokenize() (*flambe.tokenizer.word.NGramsTokenizer method*), 303
tokenize() (*flambe.tokenizer.word.NLTKWordTokenizer method*), 303
tokenize() (*flambe.tokenizer.word.WordTokenizer method*), 302
tokenize() (*flambe.tokenizer.WordTokenizer method*), 304
Tokenizer (*class in flambe.tokenizer*), 304
Tokenizer (*class in flambe.tokenizer.tokenizer*), 302
TORCH_TAG_PREFIX (*in module flambe.runner.run*), 293
TqdmFileWrapper (*class in flambe.logging.logging*),

212
train (*flambe.dataset.Dataset attribute*), 99
train (*flambe.dataset.dataset.Dataset attribute*), 95
train (*flambe.dataset.tabular.TabularDataset attribute*), 96, 97
train (*flambe.dataset.TabularDataset attribute*), 99
train (*flambe.vision.classification.datasets.MNISTDataset attribute*), 307
train (*flambe.vision.classification.MNISTDataset attribute*), 309
trainable_params (*flambe.nn.Module attribute*), 269
trainable_params (*flambe.nn.module.Module attribute*), 257
Trainer (*class in flambe.learn*), 201
Trainer (*class in flambe.learn.train*), 200
Transformer (*class in flambe.nn*), 275
Transformer (*class in flambe.nn.transformer*), 262
TransformerDecoder (*class in flambe.nn*), 276
TransformerDecoder (*class in flambe.nn.transformer*), 263
TransformerDecoderLayer (*class in flambe.nn.transformer*), 264
TransformerEncoder (*class in flambe.nn*), 276
TransformerEncoder (*class in flambe.nn.transformer*), 263
TransformerEncoderLayer (*class in flambe.nn.transformer*), 264
TransformerSRU (*class in flambe.nn*), 277
TransformerSRU (*class in flambe.nn.transformer_sru*), 265
TransformerSRUDecoder (*class in flambe.nn*), 278
TransformerSRUDecoder (*class in flambe.nn.transformer_sru*), 267
TransformerSRUDecoderLayer (*class in flambe.nn.transformer_sru*), 268
TransformerSRUEncoder (*class in flambe.nn*), 278
TransformerSRUEncoder (*class in flambe.nn.transformer_sru*), 266
TransformerSRUEncoderLayer (*class in flambe.nn.transformer_sru*), 267
traverse() (*in module flambe.compile.serialization*), 158
traverse() (*in module flambe.experiment.utils*), 179
traverse_all() (*in module flambe.experiment.utils*), 179
traverse_spec() (*in module flambe.experiment.utils*), 179
TRECDataset (*class in flambe.nlp.classification*), 237
TRECDataset (*class in flambe.nlp.classification.datasets*), 235
TrialLogging (*class in flambe.logging*), 215
TrialLogging (*class in flambe.logging.logging*), 212
TUNE_TAG_PREFIX (*in module flambe.runner.run*), 293
TuneAdapter (*class in flambe.experiment*), 184
TuneAdapter (*class in flambe.experiment.tune_adapter*), 178

U

UnpreparedLinkError, 145
UnsuccessfulRunnableError, 284
update_link_refs() (*in module flambe.experiment.utils*), 181
update_nested() (*in module flambe.experiment.utils*), 179
update_schema_with_params() (*in module flambe.experiment.utils*), 180
update_tags() (*flambe.cluster.aws.AWSCluster method*), 121
update_tags() (*flambe.cluster.AWSCluster method*), 138
UPLOAD_WARN_LIMIT_MB (*in module flambe.cluster.cluster*), 125
URL (*flambe.nlp.classification.datasets.SSTDataset attribute*), 235
URL (*flambe.nlp.classification.datasets.TRECDataset attribute*), 235
URL (*flambe.nlp.classification.SSTDataset attribute*), 237
URL (*flambe.nlp.classification.TRECDataset attribute*), 237
URL (*flambe.vision.classification.datasets.MNISTDataset attribute*), 307
URL (*flambe.vision.classification.MNISTDataset attribute*), 309
user (*flambe.runnable.environment.RemoteEnvironment attribute*), 284
user (*flambe.runnable.RemoteEnvironment attribute*), 290
user_provider (*flambe.runnable.cluster_runnable.ClusterRunnable attribute*), 281
user_provider (*flambe.runnable.ClusterRunnable attribute*), 288
user_provider (*flambe.runnable.Runnable attribute*), 287
user_provider (*flambe.runnable.runnable.Runnable attribute*), 285

V

val (*flambe.dataset.Dataset attribute*), 99
val (*flambe.dataset.dataset.Dataset attribute*), 95
val (*flambe.dataset.tabular.TabularDataset attribute*), 96, 97
val (*flambe.dataset.TabularDataset attribute*), 99
val (*flambe.vision.classification.datasets.MNISTDataset attribute*), 307
val (*flambe.vision.classification.MNISTDataset attribute*), 309

values (*flambe.logging.datatypes.HistogramT attribute*), 208
values (*flambe.logging.HistogramT attribute*), 216
ValueT (*in module flambe.logging.utils*), 213
verbose (*flambe.logging.datatypes.GraphT attribute*), 210, 211
verbose (*flambe.logging.GraphT attribute*), 219
version (*flambe.compile.serialization.SaveTreeNode attribute*), 157
VERSION_FILE_NAME (*in module flambe.compile.const*), 152
VERSION_KEY (*in module flambe.compile.const*), 152
vocab_size (*flambe.field.bow.BoWField attribute*), 191
vocab_size (*flambe.field.BoWField attribute*), 195
vocab_size (*flambe.field.label.LabelField attribute*), 192
vocab_size (*flambe.field.LabelField attribute*), 196
vocab_size (*flambe.field.text.TextField attribute*), 193
vocab_size (*flambe.field.TextField attribute*), 195
vocab_size (*flambe.nlp.transformers.field.PretrainedTransformerField attribute*), 245
vocab_size (*flambe.nlp.transformers.PretrainedTransformerField attribute*), 247

W

wait_until_accessible() (*flambe.cluster.instance.Instance method*), 112
wait_until_accessible() (*flambe.cluster.instance.instance.Instance method*), 105
walltime (*flambe.logging.datatypes.HistogramT attribute*), 208
walltime (*flambe.logging.datatypes.ImageT attribute*), 209
walltime (*flambe.logging.datatypes.PRCurveT attribute*), 210
walltime (*flambe.logging.datatypes.ScalarsT attribute*), 208
walltime (*flambe.logging.datatypes.ScalarT attribute*), 207
walltime (*flambe.logging.datatypes.TextT attribute*), 209
walltime (*flambe.logging.HistogramT attribute*), 216
walltime (*flambe.logging.ImageT attribute*), 217
walltime (*flambe.logging.PRCurveT attribute*), 218
walltime (*flambe.logging.ScalarsT attribute*), 216
walltime (*flambe.logging.ScalarT attribute*), 215
walltime (*flambe.logging.TextT attribute*), 217
WARN_LIMIT_MB (*in module flambe.runner.utils*), 294
weights (*flambe.logging.datatypes.PRCurveT attribute*), 209
weights (*flambe.logging.PRCurveT attribute*), 218
Wiki103 (*class in flambe.nlp.language_modeling*), 243
Wiki103 (*class in flambe.nlp.language_modeling.datasets*), 240
WIKI_URL (*flambe.nlp.language_modeling.datasets.Wiki103 attribute*), 240
WIKI_URL (*flambe.nlp.language_modeling.Wiki103 attribute*), 243
WordTokenizer (*class in flambe.tokenizer*), 304
WordTokenizer (*class in flambe.tokenizer.word*), 302
worker_nodes() (*flambe.cluster.instance.instance.OrchestratorInstance method*), 111
worker_nodes() (*flambe.cluster.instance.OrchestratorInstance method*), 118
write() (*flambe.logging.logging.TqdmFileWrapper method*), 212
writer (*flambe.logging.handler.tensorboard.TensorboardXHandler attribute*), 206

Y

yaml (*in module flambe.compile*), 161
yaml (*in module flambe.compile.registrable*), 155
YAML_TYPES (*in module flambe.compile.component*), 143