Flambé: A Customizable Framework for Machine Learning Experiments

Jeremy Wohlwend
ASAPP Inc.
jeremy@asapp.com

Nicholas Matthews
ASAPP Inc.
nick@asapp.com

Ivan Itzcovich
ASAPP Inc.
ivan@asapp.com

Abstract

Flambé is a machine learning experimentation framework built to accelerate the entire research life cycle. Flambé's main objective is to provide a unified interface for prototyping models, running experiments containing complex pipelines, monitoring those experiments in real-time, reporting results, and deploying a final model for inference. Flambé achieves both flexibility and simplicity by allowing users to write custom code but instantly include that code as a component in a larger system which is represented by a concise configuration file format. We demonstrate the application of the framework through a cutting-edge multistage use case: fine-tuning and distillation of a state of the art pretrained language model used for text classification. [1]

1 Introduction

Scientists and engineers in the machine learning community dedicate many hours and resouces towards preprocessing data, iterating on model architectures, tuning hyperparameters, aggregating results and ultimately deploying their most performant model. While frameworks like PyTorch (Paszke et al., 2017) and Tensorflow (et al., 2016) abstract away the details of operations like backprogpagation and make building models possible in a few lines of code, they do not explicitly aim to solve these other parts of the research cycle.

The explosion of available resources in the machine learning community (Dean et al., 2018) has included many tools that address one or more of these other phases of research, but these isolated tools do not always work harmoniously with one another, trading off customizability to provide high-level interfaces. Understanding that machine learning research particularly in the field of Natural Language Processing might require innovation at any level of abstraction and across any stage in the research process, we've built Flambé to include standardized implementations of modeling components, hyperparameter optimization and distributed execution that can all be effortlessly replaced with custom user-developed code.

By facilitating customization and iteration on a particular data pipeline and model architecture, we aim for Flambé users to spend the majority of their time doing research, not re-implementing tools for training, tuning, reporting and deploying.

Flambé's contributions are:

1. Modular machine learning components to develop replicable, state of the art research results. This includes: neural network components (pretrained or not), benchmark datasets, and standardized training and evaluation modules.

2. A configuration format that natively enables searching over hyperparameters and running remote multistage experiments at scale.

3. Smooth reporting and exporting, to facilitate sharing models and results with collaborators and the larger community.

4. An open source framework for both the academic community and teams in industry.

We demonstrate the application of our framework through a cutting-edge use case, namely knowledge distillation of a state of the art language model, the BERT model (Devlin et al., 2019), on a downstream text classification task.

2 Related work

Many different tools are attempting to tackle the various challenges of building machine learning

[1]The code and documentation can be found at https://flambe.ai

systems from different angles. Frameworks like PyTorch and Tensorflow (et al., 2016) provide the building blocks of models as simple modules e.g. various linear and recurrent layers, losses, optimizers etc. Many model implementations have been built on top of these modules, with some proposing new standardizations of specific architectures like sequence-to-sequence modeling (et al, 2019).

Libraries such as Keras (Chollet et al., 2015) offer a high-level API for building and training models. Others including AllenNLP (Gardner et al., 2018), FastAI (Howard et al., 2018) and Texar (et al, 2018) focus on some specific domains or tasks like reading comprehension or text style transfer. These types of frameworks tend to focus on training a single model at a time, but many research experiments consist of complex multistage pipelines, with hyperparameter tuning and distributed computation required at each stage. With Flambé, users can write their custom code independent from these concerns, and then easily start using algorithms like Hyperband (Li et al., 2016) and Bayesian Optimization (Bergstra et al., 2013), link components across stages, and run everything on a cluster without any modifications.

MLFlow (Zaharia et al., 2018) focuses on experiment tracking, metric reporting, and contains powerful features aimed at production deployment. However, it does not have a natural way to run hyperparameter tuning, or advanced trial sampling and scheduling.

Ray (Moritz et al., 2017) implements infrastructure for distributing computational tasks on a cluster, and it also provides a higher level extension, Tune (Liaw et al., 2018), that handles hyperparameter optimization.

Flambé leverages and builds upon existing tools, connecting the dots between frameworks like PyTorch and Ray, and providing a smooth integration between them with a powerful layer of abstraction on top. By not trying to re-implement solved problems like back-propagation and distributed task execution, we can focus our attention on usability and efficiency.

3 The Flambé Framework

Flambé executes *experiments* which are composed of a pipeline of modeling and processing stages (Subsection A), extensions that import user-supplied code (Subsection B), links to existing components (Subsection C), and tunable hyperparameters (Subsections D, E, F). All of these features are demonstrated in the `Experiment` shown in Figure 1, which defines a simple text classification task consisting of training an LSTM (Hochreiter and Schmidhuber, 1997) on the TREC dataset (Li and Roth, 2002).

```
cl: github.com/.../classification  B
---

name: trec-text-classification

pipeline:  A

  stage0: !cl.TCProcessor
    dataset: !cl.TrecDataset

  stage1: !Trainer
    train_sampler: !BaseSampler
      data: !@ stage0.train  C
      batch_size: 32
    dev_sampler: !BaseSampler
      data: !@ stage0.dev
    model: !cl.TextClassifier
      embedding: !EmbeddingEncoder
        input_size: !@ stage0.vocab_size
        embedding_size: 300
      encoder: !RNNEncoder
        input_size: 300
        rnn_type: lstm
        n_layers: !g [2, 3, 4]  D
        hidden_size: 256
        pooling: last
      decoder: !SoftmaxDecoder
        input_size: 256
        output_size: !@ stage0.n_labels
    loss_fn: !NLLLoss
    metric_fn: !Accuracy
    optimizer: !torch.Adam
      params: !@ stage1.model.params
    max_steps: 100
    iter_per_step: 10

  stage2: !Evaluator
    model: !@ stage1.model
    metric_fn: !Accuracy
    eval_sampler: !BaseSampler
      data: !@ stage0.test
      batch_size: 512

schedulers:  E
  stage1: !tune.HyperBandScheduler

reduce:  F
  stage1: 1  # pick best
```

Figure 1: Example YAML config for text classification on the TREC dataset. The highlighted and labeled sections refer to the subsections in 3.1. There are a number of different objects that could be used in any place of this config e.g. the optimizer could be `!torch.SGD` and the scheduler `tune.HyperOpt` (Bayesian optimization). Note the pipeline stage names "stage0", etc. are arbitrary.

Each tag in the YAML (Oren Ben-Kiki, 2009) config (anything beginning with '!') corresponds to a python object that will be initialized with the keyword arguments following the tag. These tags are not hardcoded into the system, and users can use their own classes in the config just as easily as the ones we've already built. After we explain all the aforementioned features, we introduce how Flambé saves object state, enables simple metric logging, and deploys models for production.

3.1 Walkthrough

In this section we present an example driven explanation of the core features as they're used in Figure 1.

A. Pipeline

The most important section of the YAML file is the `pipeline` section. This section contains a series of stages which each implement a `step` method. The example shown in Figure 1 contains 3 stages: (1) dataset loading and processing, (2) training of each model variant, and (3) evaluating the best model from `stage1`.

A stage in the pipeline can be any Python object. Users need only add a parent class to their class definition if they intend to use it in the YAML config. All objects will receive the keyword arguments given inline in the configuration file. For example, in Figure 1 the `TextClassifier` object receives an embedding, encoder and decoder, matching its definition in code:

```
from flambe.model import Model

class TextClassifier(Model):
  def __init__(self, embedding,
               encoder, decoder):
    ...
```

All subclasses of Flambé classes like `Model` are automatically registered with YAML

B. Extending Flambé with Custom Code

Flambé is flexible because of its ability to use custom *Flambé objects* in the experiment configuration file. By default, only classes in the main Flambé library and PyTorch can be referenced, but by using the `extensions` feature users can include their own classes and functions, from either local or remote source code repositories.

To create an extension, users need only organize their code into one or more pip-installable packages. After declaring the extensions and including them at the top of the config file, they are useable anywhere in the YAML configuration file.

In the example, the `TRECDataset` object is defined in an external extension hosted in GitHub. By adding its URL at the top of the YAML configuration file, the `cl.TrecDataset` object and any other Flambé class can be used. If you cannot or do not want to inherit from one of our pipeline classes (Model, Trainer, etc.) you can inherit from `flambe.nn.Module` which will supply the minimum needed functionality to support use in the config file and automatic hierarchical serialization (See later sections).

C. Referencing Earlier Objects

A core feature of Flambé is the ability to connect (or "link") different components with the `!@` notation, a custom YAML tag we've implemented. Any value anywhere in the pipeline can be a reference to an earlier value that has already been defined. Each link consists of the identifier of a stage, e.g. "stage1" which in this case is the `Trainer` object, followed by the rest of the object attributes. In the highlighted example (C), the link `stage0.train` means that the data keyword argument for `BaseSampler` should point to the `train` attribute of the `TCProcessor`.

D. Hyperparameter Search

In addition to referencing other values via links, the value for any parameter in the config can be replaced with either a list of possible options to try (for grid search) or a distribution for sampling possible options. Grid search options are defined with the `!g` tag followed by the list of candidate values; Flambé will automatically duplicate the stage, choosing a single value for each variant of the stage. In the example we use this mechanism to search over different numbers of layers.

If distributions are used instead of lists of candidate values, Flambé performs a simple random search. Users can also specify a search field that maps stage names to the hyperparameter search algorithm, e.g. Bayesian optimization, which changes the distributions used to sample the tunable hyperparameters.

When `Links` reference stages with multiple variants, the stage containing the link is duplicated as many times as there are variants.

E. Trial Scheduling

Regardless of the strategy used to choose hyperparameters, some variants will start to clearly outperform others and scheduling algorithms like Hyperband (Li et al., 2016) use that information to intelligently allocate resources to the variants that are performing the best. Flambé surfaces an interface to these schedulers in the same way as the search algorithms: "schedulers" maps pipeline stage names to the desired scheduling algorithms, as shown in the example configuration.

F. Selecting the Best Variants

After trying many different combinations of hyperparameters, only the best will propagate to the next stages if the *reduce* operation is used. For example, with *reduce* mapping `stage1` to `1` in the example, only the single best configuration, with the optimal number of layers, will be evaluated in the final stage. In order to use this feature, the stages need to supply a *metric_fn* that can be used to rank the variants.

3.2 Hierarchical Serialization

While PyTorch already provides a clear and robust saving mechanism, we augment this functionality with a generic serialization protocol for all objects that includes opt-in versioning and a directory based file format that anyone can inspect. Rather than dumping all of the model weights and other state into a single file, the directory based structure mirrors the object hierarchy and enables the possibility of referencing a specific component. Rather than having to load the save file to inspect the contents, it can be navigated like any other directory. By default, only what PyTorch normally saves is included in the save file; users can add additional state by overriding `custom_state` and `load_custom_state`

```
trec-text-classification/
  ...
  stage1/
    train_sampler/
      state.pt
    dev_sampler/
      state.pt
    model/
      embedding/
        state.pt
      encoder/
        state.pt
      decoder/
        state.pt
    optimizer/
      state.pt
    ...
```

Figure 2: Save file directory structure for the `Experiment` in Figure 1

.

3.3 Using a cluster

To run experiments on a cluster, an additional piece of YAML is needed to define the remote manager. As shown below in Figure 3 one can indicate the instance types and a timeout flag for both the orchestrator and the factories. We use this feature to keep our experiment tracking website running on the orchestrator once an experiment is over, but also to keep factories alive when rapidly experimenting or debugging. The orchestrator will communicate with workers in the cluster via Ray and Tune to execute and checkpoint progress at each step. If an experiment fails or is interrupted, it can be quickly resumed with an additional flag `resume: True`. Crucially, this remote functionality allows to distribute the execution of the variants across a cluster of machines by only adding a few lines to the configuration.

```
manager: !AWSManager
  factories_num: 1
  factories_type: "p3.2xlarge"
  orchestrator_type: "t3.large"
  factory_timeout: 1  # in hours
  orchestrator_timeout: 1
```

Figure 3: Example remote config for AWS cluster.

3.4 Deploying

Typically after experimentation, machine learning projects require packaging a model together with some preprocessing and post-processing functions into a single inference-ready interface, e.g. a text classifier that actually takes raw string(s) as input. Flambé facilitates this use-case with the `Exporter` object, wherein users can define a new version of the model from the best variants tested, and with the right interface for later use.

3.5 Library usage

In addition to using the Flambé framework via YAML configuration files, users can also use the individual objects (e.g. the Trainer, or RNNEncoder classes) in any python script. This usage may be important for users that already have a production codebase (including training scripts) written purely in Python. In a future version of the software we plan to support creating full experiments and deploying models via code (instead of YAML) to enable dynamic experiment creation and model exporting.

3.6 Logging

Flambé provides full integration with Python's `logging` module and Tensorboard ((et al., 2016)). Users are able to visualize their results by simply including `log` statements in their code (See Figure 4).

```
from flambe import log
... # inside training step
log("loss", loss, step_num)
```

Figure 4: Example log statement. Logging can be done anywhere inside a custom object.

All variants will appear under the same plot for easy analysis (see Figure 5).

4 Case study: BERT Distillation

In this section we showcase Flambé's ability to transform a pre-existing codebase with no pre-existing support for hyperparameter optimization into a complex multi-stage pipeline with a YAML config less than 80 lines long. Furthermore, We were able to find the optimal set of parameters in roughly half the time otherwise needed by adding Hyperband scheduling (Li et al., 2016), and running the experiment over a large cluster.

BERT (Devlin et al., 2019) is a popular model which performs competitively across several NLP tasks by leveraging language model pre-training over a very large corpus. Two crucial issues with the BERT model are the size of the model, and its inference speed, which generally inhibits its use in production environments. To address this issue, recent efforts have shown that most of BERT's performance on a downstream task can be conserved, while dramatically reducing its memory footprint (Chia et al., 2018).

In this experiment, we fine-tune the BERT model on two standard text classification benchmarks: TREC (Li and Roth, 2002) and Sentiment Treebank (Socher et al., 2013). We then apply knowledge distillation to reduce the BERT model to a simple 4 layer, 256 units, SRU network (Lei et al., 2018). This is a typical multistage experiment with preprossessing, fine tuning, and distillation stages. All of this can be expressed in a single, concise configuration. Results are provided in Table 1. The full configuration, containing all three stages and their respective hyperparameters, is provided as supplementary material.

Model	**TREC**	**SST2**	**# Parameters**
SRU	94.8	86.2	$\approx 5M$
BERT	96.8	91.0	$\approx 110M$
DISTILLED	95.5	87.8	$\approx 5M$

Table 1: Accuracy on benchmark text classification datasets: TREC and SST2 (Binary Sentiment Treebank). Distilling BERT improves the accuracy of the base SRU model, while reducing the number of parameters by more than 95%. All models were trained or fine-tuned using Flambé. The SRU and DISTILLED model have the same architecture, the SRU model being trained from scratch and the DISTILLED model benefiting from the BERT model's improved performance.

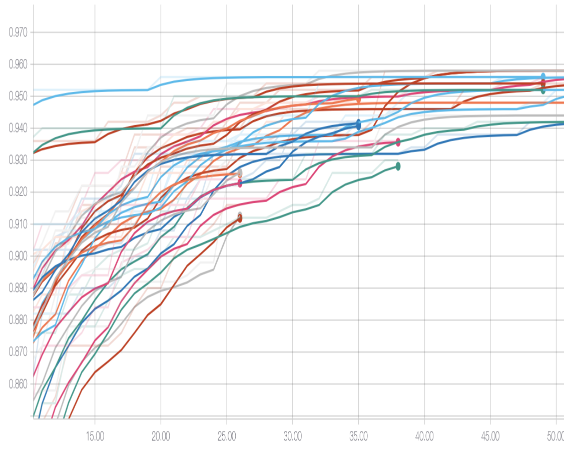


Figure 5: Some runs are pruned early by the Hyberband scheduling algorithm. The x-axis is training steps, and the y-axis is accuracy.

Not only can Flambé express the above experiment in a concise configuration, but using a state of the art trial scheduling algorithm such as Hyperband (Li et al., 2016) can be accomplished with a single additional line in the configuration. Figure 5 shows Hyperband allocating more training steps to the best-performing models. In this example, defining grid searches, running over a cluster, and using a scheduling algorithm on an existing codebase required little to no effort.

5 Future work

Flambé aims to integrate with research and engineering workflows through its focus on usability, modularity and reproducibility. We continue to pursue this goal by developing a large collection of machine learning components including state of the art models, benchmark datasets, and novel training strategies. Real, working, and reproducible experiment configurations will showcase these components alongside their performance in task-based leaderboards. In parallel, we will continue to develop user-friendly abstractions like the ability to auto-scale clusters based on the size of each stage in the pipeline, and to monitor or even alter experiment execution in real-time from a website.

References

Martin Abadi et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283.

Myle Ott et al. 2019. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota. Association for Computational Linguistics.

Zhiting Hu et al. 2018. Texar: A modularized, versatile, and extensible toolbox for text generation. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, pages 13–22, Melbourne, Australia. Association for Computational Linguistics.

James Bergstra, Dan Yamins, and David D Cox. 2013. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in science conference*, pages 13–20. Citeseer.

Yew Ken Chia, Sam Witteveen, and Martin Andrews. 2018. Transformer to cnn: Label-scarce distillation for efficient text classification.

François Chollet et al. 2015. Keras. `https://keras.io`.

Jeff Dean, David Patterson, and Cliff Young. 2018. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. 2018. AllenNLP: A deep semantic natural language processing platform. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, pages 1–6, Melbourne, Australia. Association for Computational Linguistics.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Jeremy Howard et al. 2018. fastai. `https://github.com/fastai/fastai`.

Tao Lei, Yu Zhang, Sida I Wang, Hui Dai, and Yoav Artzi. 2018. Simple recurrent units for highly parallelizable recurrence. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4470–4481.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*.

Xin Li and Dan Roth. 2002. Learning question classifiers. In *COLING 2002: The 19th International Conference on Computational Linguistics*.

Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889.

Ingy döt Net Oren Ben-Kiki, Clark Evans. 2009. Yaml. `https://yaml.org/spec/1.2/spec.html`.

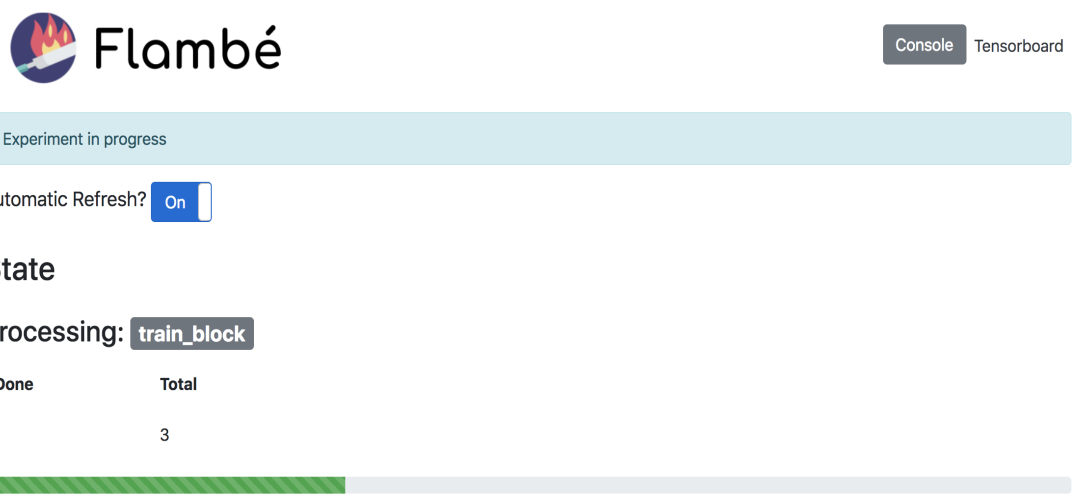
Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. In *NIPS-W*.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642.

Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the machine learning lifecycle with mlflow. *Data Engineering*, page 39.

A Screenshots

Below is a screenshot of the reporting site that includes a progress bar, links to see the console output and Tensorboard, and a download link for the model weights:



Experiment in progress

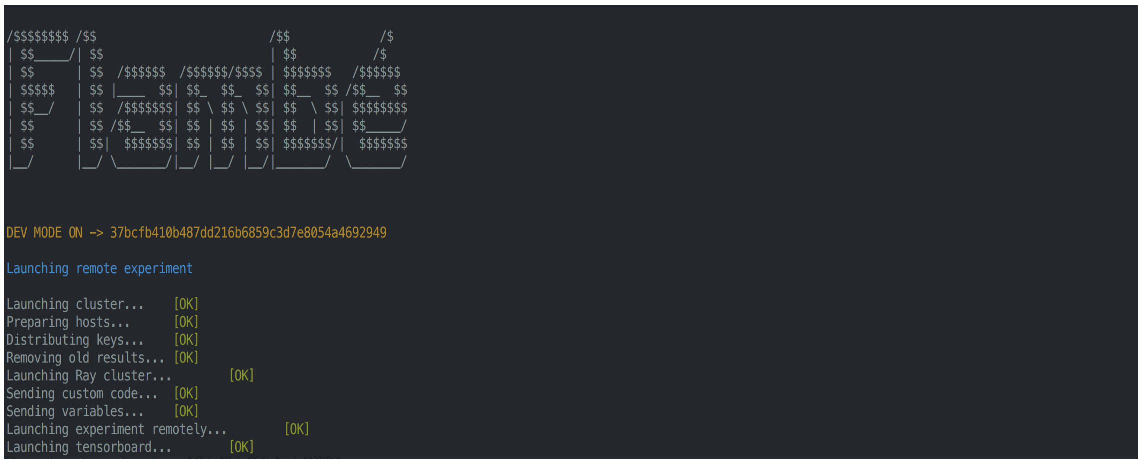
Automatic Refresh? On

State

Processing: train_block

Done	Total
1	3

When you launch an experiment from the console, you will see a series of status updates as shown below:



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



DEV MODE ON -> 37bcfb410b487dd216b6859c3d7e8054a4692949

Launching remote experiment

Launching cluster...      [OK]
Preparing hosts...        [OK]
Distributing keys...      [OK]
Removing old results... [OK]
Launching Ray cluster...          [OK]
Sending custom code...    [OK]
Sending variables...      [OK]
Launching experiment remotely...          [OK]
Launching tensorboard...          [OK]
```

B BERT Configuration File

```yaml
dist: https://github.com/.../flambe-extensions/tree/master/extensions/distillation
lm: https://github.com/.../flambe-extensions/tree/master/extensions/language_modeling
cl: https://github.com/..../flambe-extensions/tree/master/extensions/classification
---

!Experiment

name: flambe_distillation

pipeline:

  data: !cl.TRECDataset

  0_preprocess: !dist.DistillationProcessor
    student_processor: !cl.TCProcessor
      dataset: !@ data
      embeddings: !@ glove
      text_field: !TextField
    teacher_processor: !cl.TCProcessor
      dataset: !@ data
      text_field: !lm.BERTTextField.from_alias
        alias: 'bert-base-uncased'

  1_train_bert: !Trainer
    train_sampler: !BaseSampler
      data: !@ 0_preprocess.teacher_processor.train
      batch_size: 32
    dev_sampler: !BaseSampler
      data: !@ 0_preprocess.teacher_processor.dev
      batch_size: 32
    model: !cl.TextClassifier
      embedding: !lm.BERTEmbeddings.from_alias
        alias: 'bert-base-uncased'
      encoder: !lm.BERTEncoder.from_alias
        alias: 'bert-base-uncased'
      decoder: !SoftmaxDecoder
        input_size: !@ 1_train_bert.model.encoder.config.hidden_size
        output_size: !@ 0_preprocess.teacher_processor.num_labels
    loss_fn: !NLLLoss
    metric_fn: !Accuracy
    optimizer: !lm.BERTOptimizer
      params: !call 1_train_bert.model.named_parameters
      lr: 0.00005
      warmup: 0.1
    max_steps: 10
    iter_per_step: 100

  2_distillation: !dist.DistillationTrainer
    train_sampler: !BaseSampler
      data: !@ 0_preprocess.train
      batch_size: 64
    dev_sampler: !BaseSampler
      data: !@ 0_preprocess.dev
      batch_size: 64
    teacher_model: !@ 1_train_bert.model
    student_model: !cl.TextClassifier
      embedding: !EmbeddingEncoder
        input_size: !@ 0_preprocess.student_processor.vocab_size
        embedding_size: 300
        embedding_matrix: !@ 0_preprocess.student_processor.embeddings
      encoder: !RNNEncoder
        input_size: 300
        rnn_type: sru
        n_layers: 4
        hidden_size: 256
      decoder: !SoftmaxDecoder
        input_size: 256
        output_size: !@ 0_preprocess.student_processor.num_labels
    alpha_kl: !g [0.75, 0.9, 0.99]
    temperature: !g [10, 20, 30]
    loss_fn: !NLLLoss
    metric_fn: !Accuracy
    optimizer: !torch.Adam
      params: !@ 2_distillation.student_model.trainable_params
      lr: 0.001
    max_steps: 100
    iter_per_step: 100

schedulers:
  2_distillation: !tune.HyperBandScheduler
```