Fireworks: Reproducible Machine Learning and Preprocessing with PyTorch

Saad M. Khan[1] **and Libusha Kelly**[1]

1 Systems & Computational Biology, Albert Einstein College of Medicine

DOI: 10.21105/joss.01478

Software

- Review
- Repository
- Archive

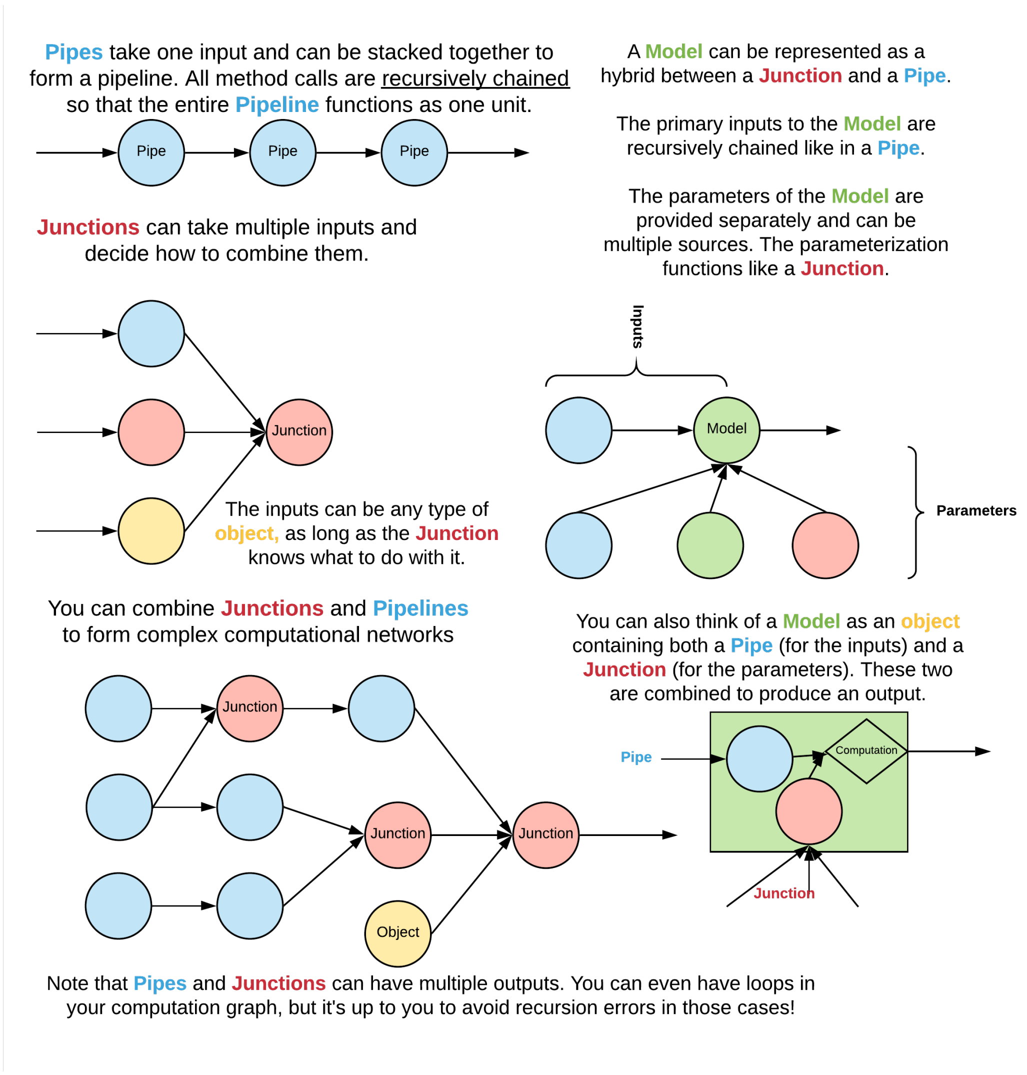
Submitted: 01 May 2019
Published: 21 July 2019

Summary

Here, we present a batch-processing library for constructing machine learning pipelines using PyTorch and dataframes. It is meant to provide an easy method to stream data from a dataset into a machine learning model while performing reprocessing steps such as randomization, train/test split, batch normalization, etc. along the way. Fireworks offers more flexibility and structure for constructing input pipelines than the built-in dataset modules in PyTorch (Paszke et al., 2017), but is also meant to be easier to use than frameworks such as Apache Spark (Zaharia et al., 2016).

Steps in a pipeline are represented using objects that can be composed in a reusable manner with a standard input and output. These input/outputs are performed via a DataFrame-like object called a Message which can have PyTorch Tensor-valued columns in addition to all of the functionality of a Pandas DataFrame (McKinney, 2010). Each of these pipeline objects can be serialized, enabling one to save the entire state of their workflow rather than just an individual model's when creating checkpoints. Additionally, we provide a suite of extensions and tools built on top of this library to facilitate common tasks such as relational database access, model training, hyperparameter optimization, saving/loading of pipelines, and more. A pain point in all deep learning frameworks is data entry; one has to take data in its original form and convert it into the form that the model expects (for example tensors, TFRecords (Abadi et al., 2015)). Data conversion can include acquiring the data from a database or an API call, formatting data, applying preprocessing transforms, and preparing mini batches of data for training. At each step, one must be cognizant of memory, storage, bandwidth, and compute limitations. For example, if a dataset is too large to fit in memory, then it must be streamed or broken into chunks. In practice, dealing with these issues is time consuming. Moreover, ad-hoc solutions often introduce biases. For example, if one chooses not to shuffle their dataset before sampling it due to its size, then the order of elements in that dataset becomes a source of bias. Ideally, proper statistical methodology should not have to be compromised for performance and memory considerations, but in practice, that often becomes the case.

We developed Fireworks to to enable reproducible machine learning pipelines by overcoming some of these challenges in data handling and processing. Pipelines are made up of objects called Pipes which can represent some transformation. Each Pipe has an input and an output. As data flows from Pipe to Pipe, transforms are applied one at a time, and because each Pipe is independent, they can be stacked and reused. Operations are performed on a pipeline by performing method calls on the most downstream Pipe. These method calls are then recursively chained up the pipeline. So for example, if one calls iterator methods on the most downstream Pipe ('iter' and 'next'), the iteration will trigger a cascade by which the most upstream Pipe produces its next output and each successive Pipe applies a transformation before handing it off. The same phenomena will apply when calling any method, such as 'getitem' for data access. This recursive architecture enables each Pipe to abstract away the inner workings of its upstream Pipe and present a view of the dataset that a downstream



Pipes take one input and can be stacked together to form a pipeline. All method calls are recursively chained so that the entire Pipeline functions as one unit.
Pipe
Pipe
Pipe
Junctions can take multiple inputs and decide how to combine them.
Junction
The inputs can be any type of object, as long as the Junction knows what to do with it.
A Model can be represented as a hybrid between a Junction and a Pipe.
The primary inputs to the Model are recursively chained like in a Pipe.
The parameters of the Model are provided separately and can be multiple sources. The parameterization functions like a Junction.
Inputs
Model
Parameters
You can combine Junctions and Pipelines to form complex computational networks
Junction
Junction
Junction
Object
You can also think of a Model as an object containing both a Pipe (for the inputs) and a Junction (for the parameters). These two are combined to produce an output.
Pipe
Computation
Junction
Note that Pipes and Junctions can have multiple outputs. You can even have loops in your computation graph, but it's up to you to avoid recursion errors in those cases!

Figure 1: Illustration of the core primitives in Fireworks.

Pipe can build additional abstractions off of. For example, a Pipe could cache data from its upstream sources so that whenever a downstream data access call is made, the caching Pipe can retrieve the item from its cache. As another example, a Pipe could virtually shuffle its inputs so that every time an element at a given index is requested, a random element at some other index is returned instead. This framework also enables just-in-time evaluation of data transformations. A downstream Pipe could call a method only implemented by an upstream Pipe, enabling one to delay applying a transformation until a certain point in a pipeline. Lastly, there are additional primitives for creating branching pipelines and representing PyTorch models as Pipes that are described in the documentation.

Because of this modular architecture and the standardization of input/output via the Message object, one can adapt traditional machine learning pipelines built around the Pandas library, because Messages behave like DataFrames. As a result, Fireworks is useful for any data processing task, not just for deep learning pipelines, as described here. It can be used, for example, to interact with a database and construct statistical models with Pandas as well. Additionally, we provides modules for database querying, saving/loading checkpoints, hyperparameter optimization, and model training in addition premade functions for common preprocessing tasks such as batch normalization and one-hot vector mapping using Pipes and

Messages. All of these features are designed to improve the productivity of researchers and the reproducibility of their work by reducing the amount of time spent on these tasks and providing a standardized means to do so. The overall design of Fireworks is aimed at transparency and backwards-compatibility with respect to the underlying libraries. This library is designed to be easy to incorporate into a project without requiring one to change the other aspects of their workflow in order to help researchers implement cleaner machine learning pipelines in a reproducible manner to enhance the scientific rigor of their work.

We currently use this library in our lab to implement neural network models for studying the human microbiome and for finding phage genes within bacterial genomes. In the future, we will implement integration with other popular data science libraries in order to facilitate the inclusion of Fireworks in common data workflows. For example, distributed data processing systems such as Apache Spark could be used to acquire initial data from a larger data-set that can then be pre-processed and mini-batched using Fireworks to train a PyTorch Model. The popular python library SciKit-Learn (Pedregosa et al., 2011) also has numerous pre-processing tools and models that could be used within a pipeline. There is also a growing interest in conducting data science research on the cloud in a distributed environment. Libraries such as KubeFlow ("Kubeflow," 2019) enable one to spawn experiments onto a cloud environment so that multiple runs can be performed and tracked simultaneously. In the future, we will integrate these libraries and features with Fireworks.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Retrieved from https://www.tensorflow.org/

Kubeflow. (2019, January). https://www.kubeflow.org/.

McKinney, W. (2010). Data Structures for Statistical Computing in Python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th python in science conference* (pp. 51–56).

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., et al. (2017). Automatic differentiation in PyTorch. In *NIPS autodiff workshop*.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., et al. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., et al. (2016). Apache spark: A unified engine for big data processing. *Commun. ACM*, *59*(11), 56–65. doi:10.1145/2934664