
The Lottery Ticket Hypothesis

Authors

This codebase was developed by Jonathan Frankle and David Bieber at Google during the summer of 2018.

Background

This library reimplements and extends the work of Frankle and Carbin in “The Lottery Ticket Hypothesis: Finding Small, Trainable Neural Networks” (<https://arxiv.org/abs/1803.03635>). Their paper aims to explore why we find large, overparameterized networks easier to train than the smaller networks we can find by pruning or distilling. Their answer is the lottery ticket hypothesis:

Any large network that trains successfully contains a subnetwork that is initialized such that - when trained in isolation - it can match the accuracy of the original network in at most the same number of training iterations.

They refer to this special subset as a *winning ticket*.

Frankle and Carbin further conjecture that pruning a neural network after training reveals a winning ticket in the original, untrained network. They posit that were pruned after training were never necessary at all, meaning they could have been removed from the original network with no harm to learning. Once pruned, the original network becomes a winning ticket.

To evaluate the lottery ticket hypothesis in the context of pruning, they run the following experiment:

1. Randomly initialize a neural network.
2. Train the network until it converges.
3. Prune a fraction of the network.
4. To extract the winning ticket, reset the weights of the remaining portion of the network to their values from (1) - the initializations they received before training began.
5. To evaluate whether the resulting network at step (4) is indeed a winning ticket, train the pruned, untrained network and examine its convergence behavior and accuracy.

Frankle and Carbin found that running this process iteratively produced the smallest winning tickets. That is, the network found at step (4) becomes a new network to train and further prune again at steps (2) and (3). By training, pruning, resetting, and repeating many times, Frankle and Carbin achieved their best results.

Purpose

This library reimplements Frankle and Carbin’s core experiment on fully-connected networks for MNIST (Section 2 of their paper). It also includes several additional capabilities for further examining the behavior of winning tickets.

Getting Started

1. Run `setup.py` to install library dependencies.
2. Modify `mnist_fc/locations.py` to determine where to store MNIST (`MNIST_LOCATION`) and the data generated by experiments (`EXPERIMENT_PATH`).
3. Run `download_data.py` to install MNIST in those locations.

Code Walkthrough

This codebase is divided into four top-level directories: `foundations`, `datasets`, `analysis`, and `mnist_fc`.

Foundations

The `foundations` directory contains all of the abstractions and machinery for running lottery ticket experiments.

Dataset and Model A learning task is represented by two components: a `Model` to train and a `Dataset` on which to train that network. Base classes for these abstractions are in `foundations/dataset_base.py` and `foundations/model_base.py`. Any networks on which you wish to run the lottery ticket experiment must subclass the `ModelBase` class in `foundations/model_base.py`; likewise, any datasets on which you wish to train data must subclass the `DatasetBase` class in `foundations/dataset_base.py`. `foundations.model_fc` implements a generic fully-connected model.

`Model` objects in this codebase have two special features that distinguish them from normal models:

- *masks*: arrays of 0/1 values that are multiplied by tensors of network parameters to permanently disable particular parameters.
- *presets*: specific values to which network parameters should be initialized.

Masks are the mechanism by which weights are pruned. To prune a weight, set the value of the corresponding position in the mask to 0. Presets are the mechanism by which a network can be initialized to specific values, making it possible to perform the “reset” step of the lottery ticket experiment.

`ModelBase` has a `dense` method that mirrors the `tf.layers.dense` method but automatically integrates masks and presets. You should use this method when you build your networks so that weights can properly be managed during the course of the lottery ticket experiment, and you may desire to write similar methods for `conv2d`, etc. if you want to work with other kinds of layers.

Trainer, Pruner, and Experiment To train a network, run the `train` function of `foundations.trainer`, providing it with a `Model` and `Dataset`.

A lottery ticket experiment comprises iteratively training a network, pruning it, resetting it, and training it again. The infrastructure for doing so is in the `experiment` function of `foundations.experiment`. This function expects four functions as arguments (in addition to other parameters).

- `make_dataset`: A function that generates a dataset. This function is called before each training run to re-generate the dataset.
- `make_model`: A function that generates the mode on which the lottery ticket experiment is to be run. This function is also called after each pruning step to generate the next model to be trained. This function can take masks and presets, which is useful for pruning the network and initializing its parameters to the same values as those of the original network.
- `train_model`: A function that trains the model generated by `make_model` on the dataset generated by `make_dataset`.
- `prune_masks`: A function that performs a pruning step, updating the previous masks based on network weights at the end of training. `foundations.pruner` implements the pruner from the original lottery ticket paper.

At a high level, here is how an experiment is structured:

- An *experiment* consists of running the complete lottery ticket process (starting with a network and iteratively training and pruning many times). Experiments often build off of one another, reusing and transforming networks from other experiments for new purposes.
- We often perform the same experiment more than once in order to demonstrate repeatability, so there are likely to be multiple *trials* for each experiment.
- When k pruning steps have taken place, a network is said to be pruned to *level* k .
- Training one individual network at one level of one experiment trial is called a *run*.
- In a run, a network is trained for a certain number of training steps, or *iterations*.

Paths, Saving, and Restoring The `foundations.paths` module contains helper functions for managing the locations where data generated by the experiments is stored. Each experiment generates five outputs:

- The `initial` weights, `final` weights, and `masks` of the network.
- Training, test, and validation loss and accuracy at frequent intervals throughout the training process as both a JSON file and a set of tensorflow summaries.

`foundations.paths` has functions that create the appropriate filenames for each of these records when provided with the directory in which they should be stored. It also has functions that structure where the results of a particular experiment, trial, and run are stored.

The `foundations.save_restore` module contains functions for saving networks, masks, and experimental results.

Networks and masks are stored as dictionaries whose keys are layer names and whose values are numpy arrays of the corresponding values for each layer. The `standardize` function of `foundations.save_restore` takes as input either a dictionary storing a network or the path to the location where such a network is stored; either way, it returns the dictionary. This function is used throughout the codebase to handle cases where a network could be represented by either a path or a dictionary.

Datasets

The `datasets` directory stores specific datasets - children of the `DatasetBase` class. Right now, only `dataset_mnist` is present.

MNIST on a Fully-Connected Network

The `mnist_fc` directory contains the experimental infrastructure for instantiating the foundations on a fully-connected network for MNIST. It has three main components:

- Top-level files.
- Runners.
- Argfiles.

Top-level Files The top-level files (`train.py`, `lottery_experiment.py`, etc.) contain the infrastructure for running MNIST experiments.

Support infrastructure:

-
- `locations.py`: locations where datasets and data should be stored.
 - `download_data.py`: downloads MNIST and converts them into the formats expected by `dataset_mnist.py`.
 - `constants.py`: constants specific to the MNIST experiments (hyperparameters) and functions that construct locations for storing experimental results.

Infrastructure for running experiments:

- `train.py`: Trains a single network, optionally with masks and presets.
- `lottery_experiment.py`: Performs the lottery ticket experiment, optionally with presets.
- `reinitialize.py`: Runs the random reinitialization (“control”) experiment on a particular network.

Runners For each of the scripts for running experiments, there is a corresponding `runner` which uses Python Fire to make these scripts callable from the command line. For example, you can run `runners/lottery_experiment.py` to execute it from the command line by using its function arguments as flags.

Argfiles The `argfiles` directory contains scripts that generate sets of flags for the runners to perform experiments. The `argfile_runner.py` script will run the experiments specified in an argfile on a particular runner. For example:

```
python argfile_runner.py runners/lottery_experiment.py argfiles/  
lottery_experiment_argfile.py
```

will run the lottery experiment for each of the sets of flags generated by `lottery_experiment_argfile.py`.

Disclaimer: This is not an official Google product.