Programming Assignment 3: Training Neural Networks

CS4787/5777 — Principles of Large-Scale ML — Fall 2023

Project Due: Monday, November 6, 2023 at 11:59pm

Late Policy: Up to two slip days can be used for the final submission.

Please submit all required documents to **Gradescope**. There are two separate assignments there, one for the code and one for the report. You need to submit **both**.

This is a group project. You can either work alone, or work **ONLY** with the other members of your group, which may contain AT MOST three people in total. Failure to adhere to this rule (by e.g. copying code) may result in an Academic Integrity Violation.

Overview: In this project, you will be learning how to train a deep neural network using a machine learning framework. While you've seen in the homework that differentiating even simple deep neural networks by hand can be tedious, machine learning frameworks that run backpropagation automatically make this easy. This assignment instructs you to use PyTorch, one the most popular machine learning frameworks at the moment, but the experience should transfer to whatever machine learning frameworks you decide to use in your own projects. These frameworks and learning methods drive machine learning systems at the largest scales, and the goal of this assignment is to give you some experience working with them so that you can build your intuition for how they work.

This assignment is designed and tested with the most recent version of PyTorch, running on a CPU. If you run into issues with the code, please check you have the right version of PyTorch installed.

In this assignment, you are going to explore training a neural network on the MNIST dataset, the same dataset you have been working with so far in this class. MNIST is actually a relatively small dataset to use with Deep Learning, but it's a good first dataset to use to start playing around with these frameworks and learning how they work. While image datasets like MNIST usually use convolutional neural networks (CNNs), here for simplicity we'll mostly look at how a fully connected neural network performs on MNIST.

Please do not wait until the last minute to do this assignment! While I have constructed it so that the programming part will not take so long, (especially since it's over the prelim) actually training the networks can take some time, depending on the machine you run on. It takes my implementation about five minutes to train all the networks (without any hyperparameter optimization).

Instructions: This project is split into three parts: the training and evaluation of a fully connected neural network for MNIST, the exploration of hyperparameter optimization for this network, and the training and evaluation of a convolutional neural network (CNN) which is more well-suited to image classification tasks.

Part 1: Fully Connected Neural Network on MNIST.

1. Start by implementing the following functions:
    - `construct_dataloaders`, which constructs dataloaders of type `torch.utils.data.DataLoader` for the MNIST dataset. These dataloader objects are responsible for constructing minibatches in PyTorch.
    - `evaluate_model`, which evaluates the average loss and accuracy of a model on a dataset represented as a `DataLoader`.
    - `train`, which trains a model by running an optimization algorithm for a specified number of epochs. To do this, `train` should iterate through the training dataloader, and at each step should (1) evaluate the model at the example from the dataloader, (2) compute the loss, (3) zero the gradient accumulators by calling `zero_grad` on the optimizer, (4) call `backward` on the loss, and (5) call `optimizer.step`. The function `train` should output six arrays of per-epoch statistics:

- Two arrays which report the average of the loss and accuracy of the minibatches used during training. Note that while this is often called the "training loss" and "training accuracy," it isn't really the loss/accuracy on the training set, since it's not the accuracy/loss of any one model: rather, it's an average of the loss/accuracy values at whatever the model parameters were at each step of the optimization algorithm.
- Two arrays which report the training loss of the model at the end of the epoch. This can be measured by calling `evaluate_model`. The computation of these arrays can be turned off with a flag, which you may find useful for saving time in hyperparameter optimization later.
- Two arrays which report the test loss of the model at the end of the epoch. This can be measured by calling `evaluate_model`. The computation of these arrays can be turned off with a flag, which you may find useful for saving time in hyperparameter optimization later.

Be sure to call `model.train()` before the training part of each epoch, and `model.eval()` before the calls to `evaluate_model` at the end of each epoch. This will ensure that the batch norm layers are running correctly.

Also implement a function, `make_fully_connected_model_part1_1`, that uses PyTorch to construct a `torch.nn.Sequential` neural network model with the following architecture.

- A flatten layer
- A dense (fully connected) layer with output size $d_1 = 1024$
- A ReLU nonlinearity.
- A second dense (fully connected) layer with output size $d_2 = 256$.
- A ReLU nonlinearity.
- As the last layer, a dense (fully connected) layer with output size $c = 10$.

Run your function to this network with a cross entropy loss (hint: you will find the `torch.nn.CrossEntropyLoss` loss from PyTorch to be useful here) using stochastic gradient descent (SGD). Use the following hyperparameter settings and instructions:

- Learning rate $\alpha = 0.1$.
- Minibatch size $B = 100$.
- Run for 10 epochs.

Your code should save the statistics that are output by `train` in addition to the total wall-clock time used for training. You should expect to get about 98% test accuracy here.

2. Now modify the hyperparameters for code you used above in Part 1.1 to support learning with momentum. Train your network with momentum SGD using the following hyperparameter settings and instructions
   - Learning rate $\alpha = 0.1$.
   - Momentum $\beta = 0.9$.
   - Minibatch size $B = 100$.
   - Run for 10 epochs.

   You should save the same statistics as listed above.
3. Now modify your code to train the same neural network using Adam. Train your network using the following hyperparameter settings and instructions
   - Learning rate $\alpha = 0.001$.
   - First moment decay $\rho_1 = 0.99$.
   - Second moment decay $\rho_2 = 0.999$.
   - Minibatch size $B = 100$.

- Run for 10 epochs.

You should save the same statistics as listed above.

4. Finally, let's explore whether batch normalization can help improve our accuracy or convergence speed here. Implement a function, `make_fully_connected_model_part1_4`, that builds the same neural network, but now with batch normalization: specifically, add a batch norm layer after each linear layer in the original network from 1.1. Train this network using the following hyperparameter settings and instructions:
   - Learning rate $\alpha = 0.001$.
   - Momentum $\beta = 0.9$.
   - Minibatch size $B = 100$.
   - Run for 10 epochs.

   You should save the same statistics as listed above.
5. For each of the four training algorithms you ran above, plot the following two figures:
   - One figure that displays the approximated training loss (from minibatch average), the end-of-epoch training loss, and the test loss versus epoch number.
   - One figure that displays the approximated training accuracy (from minibatch average), the end-of-epoch training accuracy, and the test accuracy versus epoch number.

   This is a total of eight figures.
6. Report the wall-clock times used the the algorithm for training. How does the performance of the different algorithms compare? Please explain briefly.
7. How does the "approximate" training loss gotten "for free" from the minibatches during traning compare to the "real" end-of-epoch training loss? Is this easier-to-compute proxy a good metric for evaluating convergence? Explain briefly.
8. For the rest of the assignment, you can/should turn off the evaluation of the training error/loss at each epoch, because it is costly.

Part 2: Hyperparameter Optimization.

1. For the SGD with momentum algorithm, use grid search to select the step size parameter from the options $\alpha \in \{1.0, 0.3, 0.1, 0.03, 0.01, 0.003, 0.001\}$ that maximizes the validation accuracy. Report the values of the validation accuracy and validation loss you observed for each setting of α, and report the α that the grid search selected. Did the step size found by grid search improve over the step size given in the instructions in Part 1?
2. Now choose any one of the four algorithms from Part 1, and choose any three hyperparameters you want to explore (e.g. the momentum, the layer width, the number of layers, et cetera). For each hyperparameter, choose a grid of possible values you think will be good to explore. Report your grid, and justify your selection. Then run grid search using the grid you selected. Report the best parameters you found, and the corresponding validation accuracy, validation loss, test accuracy, and test loss.
3. Now use random search to explore the same space as you did above in Part 2.2. For each hyperparameter you explored, choose a distribution over possible values that covers the same range as the grid you chose for that hyperparameter in Part 2.2. Report your distribution, and justify your selection. Then run random search using the distribution you selected, running at least 10 random trials. Report the best parameters you found, and the corresponding validation accuracy, validation loss, test accuracy, and test loss.
4. How did the performance of grid search compare to random search? How did the total amount of time spent running grid search compare to the total amount of time spent running random search?

Part 3: Convolutional Neural Networks.

1. Implement a function, `make_cnn_model_part3_1`, that builds a `torch.nn.Sequential` convolutional neural network with the following architecture.

- A 2D convolution layer with 16 output channels, a kernel size of (3,3), no padding, a stride of 1.
- A 2D batch norm layer.
- A ReLU activation.
- A second 2D convolution layer with 16 output channels, a kernel size of (3,3), no padding, a stride of 1.
- A 2D batch norm layer.
- A ReLU activation.
- A 2D max pooling layer with pool size (2,2). (This downscales the image by a factor of 2.)
- A 2D convolution layer with 32 output channels, a kernel size of (3,3), no padding, a stride of 1.
- A 2D batch norm layer.
- A ReLU activation.
- A 2D convolution layer with 32 output channels, a kernel size of (3,3), no padding, a stride of 1.
- A 2D batch norm layer.
- A ReLU activation.
- A 2D max pooling layer with pool size (2,2). (This downscales the image by a factor of 2.)
- A flatten layer.
- A dense (fully connected) layer with output size 128.
- A ReLU activation.
- As the last layer, a dense (fully connected) layer with output size $c = 10$.

Run your function to this network with a cross entropy loss (as before), using the Adam optimizer and the following hyperparameter settings and instructions:

- Learning rate $\alpha = 0.001$.
- First moment decay $\rho_1 = 0.99$.
- Second moment decay $\rho_2 = 0.999$.
- Minibatch size $B = 100$.
- Run for 10 epochs.

You should save the same statistics as listed above.

2. Plot the following two figures:
   - One figure that displays computed-from-minibatches-during-training training loss and test loss versus epoch number.
   - A second figure that displays computed-from-minibatches-during-training training accuract and test accuract versus epoch number.

   This is a total of two figures.
3. Report the wall-clock times used the the algorithm for training. How does the performance compare to the performance of the fully connected network you studied in Part 1?
4. How does the number of parameters used in this network compare to the number of parameters used in the MLPs you studied in Part 1?
5. Can you find a way to further improve this CNN architecture, either by improving its hyperparameters, decreasing its parameter count by adjusting its architecture, or some other method? Explain your approach and reasoning.

What to submit:

1. An implementation of the functions in main.py.
2. A lab report containing:
    - A one-paragraph summary of what you observed during this programming assignment.
    - Plots of the training/test loss/accuracy of the various training algorithms, as described in Parts 1 and 3.
    - Wall clock times for each algorithm used for training.
    - Answers to the questions in 1.6 and 1.7.
    - Answers to the question in 2.1.
    - A description of the grid you used in Part 2.2, and a justification for why you used that grid.
    - The best hyperparameter values you found in your grid search in Part 2.2, and the corresponding validation/test loss/accuracy.
    - A description of the distribution you used in Part 2.3, and a justification for why you used that distribution.
    - The best hyperparameter values you found in your random search in Part 2.3, and the corresponding validation/test loss/accuracy.
    - A short paragraph comparison of the performance of random search and grid search, as described in Part 2.4.
    - Answers to the questions in Parts 3.4 and 3.5.

Setup:

1. Run `pip3 install -r requirements.txt` to install the required python packages