Project 1: Running Stochastic Gradient by PyTorch

Zhiping Xiao (Patricia), Student ID 604775684

***Abstract*—This is the first project of UCLA CS 269: Optimization Methods for Deep Learning, 2019 Spring quarter. The requirements are listed on course website. Done individually by Zhiping Xiao (Patricia). The project involves: (1) setting up PyTorch environment; (2) build a simple CNN model; (3) run it. It is experiment-oriented, and thus in the following parts I'm going to discuss the details of the experiments.**

***Index Terms*—PyTorch, Stochastic Gradient Descent, CNN**

I. Introduction

As the very first project of CS269, this project is done individually, served as a warm-up project getting us familiar with the use of PyTorch [1], a rising star among all open-sourced deep learning platforms.

In this report I'm going to discuss:

- installation process of PyTorch;
- discussion on PyTorch usage and difficulties;
- results in terms of the running time and accuracy.

My code is not released to the public. In case that the code is needed, please contact me at *patriciaxiao@g.ucla.edu*.

II. Installation

PyTorch installation is slightly more troublesome than TensorFlow or Keras (who are maturer at the current stage), but it is tolerable in general. Guidlines on installation are available at https://pytorch.org/get-started/locally/.

My operating system Mac OS 10.13.3. It is said that in order to install PyTorch with Python 3.6 or higher, we can't use the default installation. Anaconda or Homebrew should help.

Since Python2 which I am currently using will be deprecated within a few years, I installed Anaconda [2], who perfectly helped me handle Python version, and by default I am using Python 3.7.1 right now.

There's no specific difficulty in installation.

III. PyTorch

Generally speaking, PyTorch is pretty understandable. I found the following links useful:

- official documentation on *torch.nn* at https://pytorch.org/docs/stable/nn.html, with the detailed definitions of activation functions, convolution layers, and pooling layers, etc;
- an example of CNN provided by official documentation at https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html;
- the optimization options documentation is available at https://pytorch.org/docs/stable/optim.html;
- in particular, the detailed description of SGD is here at https://pytorch.org/docs/stable/_modules/torch/optim/sgd.html;
- introduction of padding from official documentation at https://pytorch.org/docs/stable/_modules/torch/nn/modules/padding.html.

There isn't any real difficulty in using the package. However, from my perspective, the documentation is less maturer than that of TensorFlow or Keras. We can easily figure out one way of implementation, but it is hard to figure out other equivalent ways without hints from other developers. For instance, it is easy to figure out that we can write convolutional layers, pooling layers, padding, etc. separately, but I didn't see obvious hint popping out *nn.Sequential*, which contributed a lot to my code's readability. [3]

Plus, just like any other deep learning package, the suggestions on online forums and blogs, such as stackoverflow, are not guaranteed to be reliable. Everything need to be tested. But the workload doesn't exceed half a day in total.

I think there could be one improvement: providing default-settings on more parameters if possible. Besides, when it comes to the cascaded layers, it is kind of confusing that we should manually compute the shapes of some layers and hard-code it in. It'll be much better if this part could be automatically handled.

IV. Results and Summary

First of all, I specify some hyper-parameters as indicated in Table I. I didn't tune them, and for all other hyper-parameters, I used the default settings, as required.

Parameter	learning rate	# epochs	batch size	momentum
Value	0.01	6	64	0 (default)

TABLE I
My Parameters

As for the datasets, I used MNIST and CIFAR10, according to the requirements.

Although it is clearly stated in the project assignment that we don't need to tryout different settings, I am kind of curious about what impact would normalization bring to the performance, so personally I tried with and without normalization on both datasets.

The accuracy of prediction on test dataset, after 6 epochs, is as shown in Table II. It shows that the naive CNN approach performs much better on MNIST than on CIFAR10, and that with normalization, the accuracy is significantly increased.

However, as is shown in Figure 1, normalization of the data points slows down the training process a little bit. But generally speaking, considering the benefit, this delay could be ignored.

[1]https://pytorch.org/

[2]https://www.anaconda.com/distribution/#macos

[3]Here I should specifically thank Haoran, for giving me this hint.

Normalized?	MNIST	CIFAR10
NO	0.9764	0.4701
YES	0.9827	0.555

TABLE II
FINAL ACCURACY ON DIFFERENT SETTINGS

Comparing the model's performance on the two datasets, Figure 3 shows how well it converges, and Figure 2 shows its performance measured by accuracy of prediction on test set. It shows that the dataset's features have great impact on the outcome of the model.

Other results of loss and accuracy are shown in the figures listed.

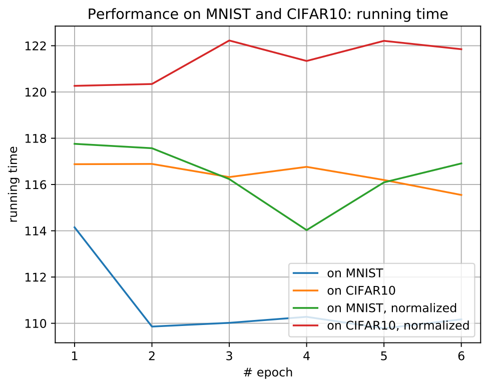

Fig. 1. Running Time for the Four Cases

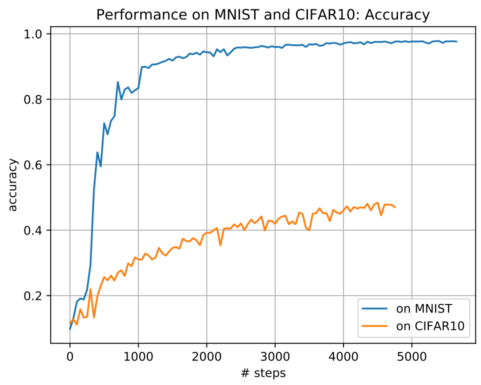

Fig. 2. Accuracy on the Datasets

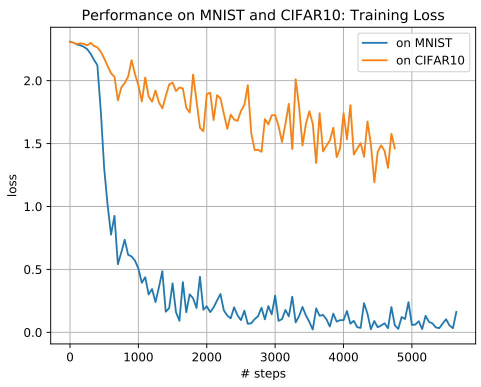

Fig. 3. Training-Loss on the Datasets

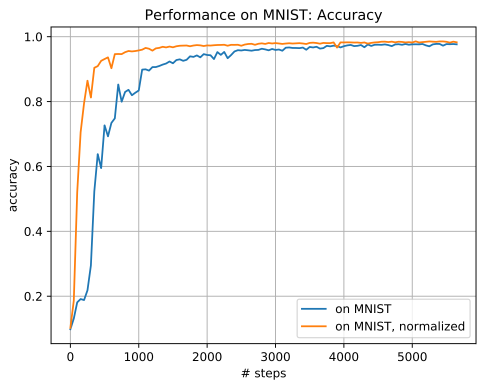

Fig. 4. Test Accuracy on MNIST

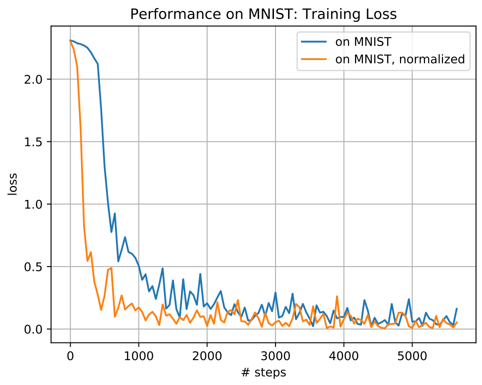

Fig. 5. Training Loss on MNIST

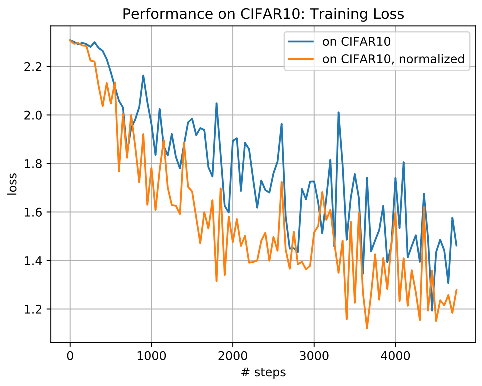

Fig. 6. Test Accuracy on CIFAR10

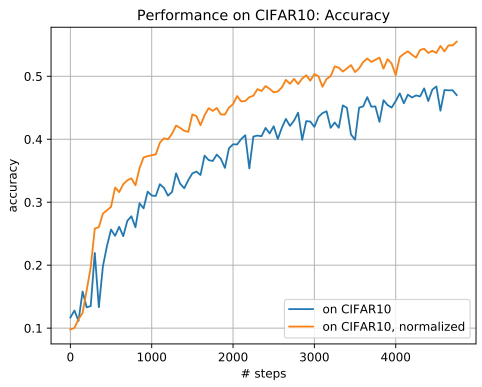

Fig. 7. Training Loss on CIFAR10

Acknowledgment

Lots of thanks to Haoran Wang, who also enrolled in this course, for giving me hints on using PyTorch and convinced me that PyTorch is elegant.