Homework 4

Neural Networks - Back-propagation pass on MNIST

Xiaoyu Xiang(xiang43@purdue.edu)

Platform and Packages

- Python 3.6.5
- PyTorch 0.4.0, torchvision
- Numpy, matplotlib

Dataset: MNIST

To load the dataset, we use the PyTorch MNIST dataloader.

For the MNIST dataset, we separate it into 2 parts: training set(60,000 images), and and test set(10,000 images). During the training process, we expect to see the training loss and validation loss decrease over epochs.

Then we decide the batch size. Batch size decides how many samples would be sent to train. Let the whole sample number of training set be N, and batch size c, so the whole training set is divided into floor(N/c) parts.

We only shuffle the training set's data. For validation set, since the shuffle wouldn't inlfuence the output, we skip the shuffle process.

After one epoch, we will calculate the training error, which should be the average over all training examples(or the average over all batches).

Code Implementation

neural_network.py

In this script, we define a class NeuralNetwork, to use this class:

```
from neural_network import NeuralNetwork
```

It takes 1 input, that defines the size of input for each layer.

`NeuralNetwork.getlayer(layer_index: int)` can assign the weights for each layer. If not assigned, each layer will take random numbers(mean = 0, std = 1/sqrt(layer size)) as weights.

`NeuralNetwork.forward(input: FloatTensor)` can do the forward pass. For each layer, the output `y` has the following relationship with the input `x`:

`y = Wx + b`

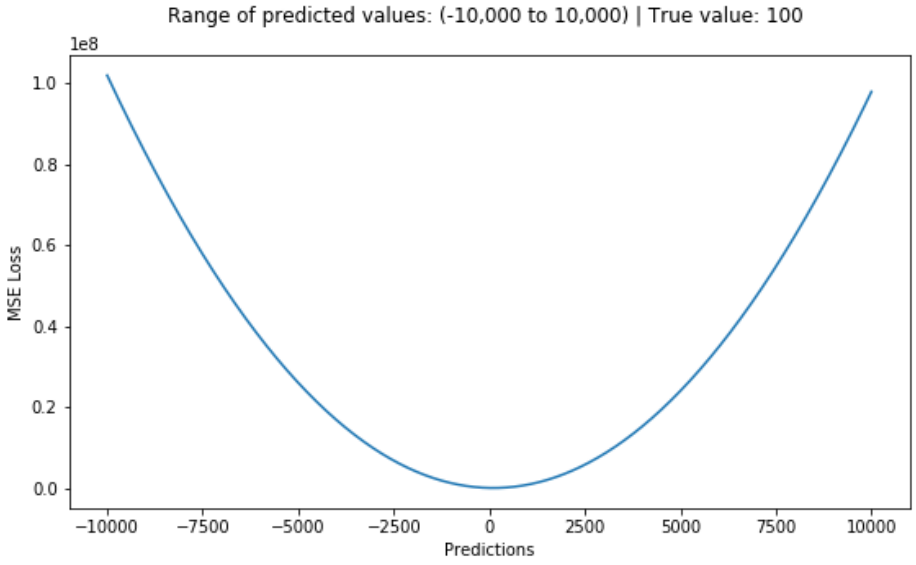
Where the b is the bias node.

`NeuralNetwork.backward(target: FloatTensor, loss: string)` can take 1D or 2D FloatTensor as input, which can compute the gradient of target matrix for back-propagation pass.

`NeuralNetwork.updateParams(eta: float)` is used for update parameters with the calculated backward values and given rate eta.

The loss function used in this update strategy is Mean Square Error Loss, or L2 Loss. Which can be expressed in the formula below.

$$MSE = \frac{\sum_{i=1}^{n} (y_i - y_i^p)^2}{n}$$



Also, the loss function can be changed and even defined by ourselves. That's what we can set through the parameter loss.

If we set it `'None'`, it will apply the default loss MSE.

By setting it as `'CE'`, cross-entropy loss is applied:

$$CE = -\sum_{x} p(x) \log q(x)$$

Where p is the ground truth, an q is the network output(or, prediction result).

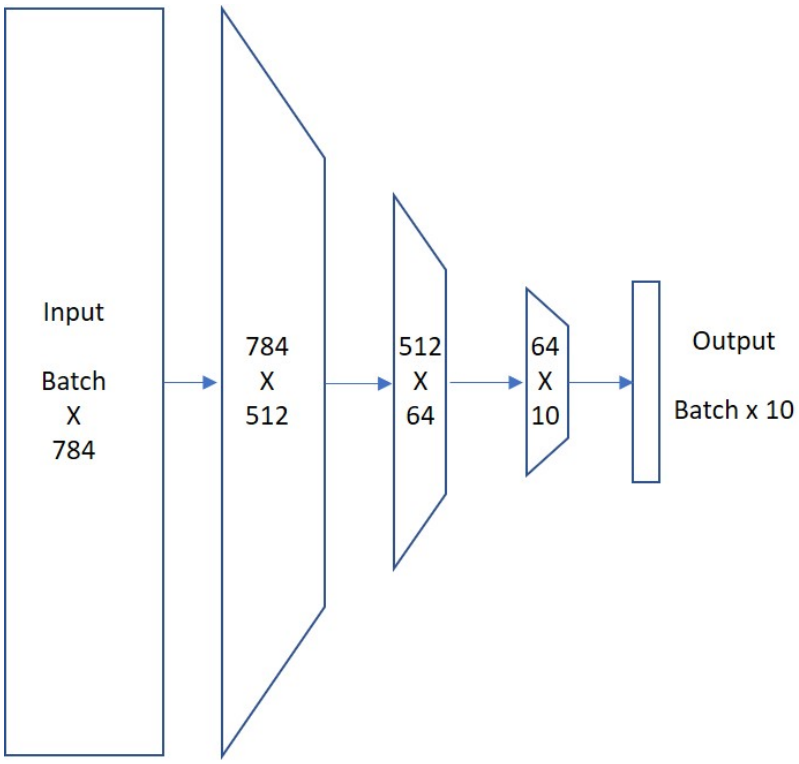
myimg2num.py

```
from myimg2num import MyImg2Num
```

This class is used to turn images in mnist (handwritten numbers) into numbers. The network is init by `net = MyImg2Num()`.

To achieve this, we build a 5-layer neural network. The input of this network is 28x28x1 images(which is turned into a 1-D vector: 1 x 784), and the output is the probability of 10 classes.

The structure of this network can be expressed below, including 3 hidden layers:



Use `net.train()` to start the training process.

To speed up training, we use the NeuralNetwork API in batch mode. Let c be the batch size, so the network's input would be c x 782, and output would be c x 10. The batch size only influence the backward propagation: instead of calculate the gradient one by one, we calculate the average gradient over all samples in this batch, and return the average loss.

After training, it will also give a plot of training loss, and test accuracy, and the time spent on training. All those data would also be written as log file.

This class also has a function: `net.forward(image)` , which takes one single image of handwritten number(28 x 28 size), and give the output based on highest score among all classes.

nnimg2num.py

```
from nnimg2num import NnImg2Num
```

This class achieves the same goal as `myimg2num.py` , but instead of using our homemade neural network, we apply PyTorch's nn package to build the neural network, and realize the optimization step.

To init the network: `net = NnImg2Num()`

Traing this network: `net.train()`

Get number output from input image: `net.forward(image)`

Test Result

We set the same parameters for training process in my neural network and pytorch's neural network:

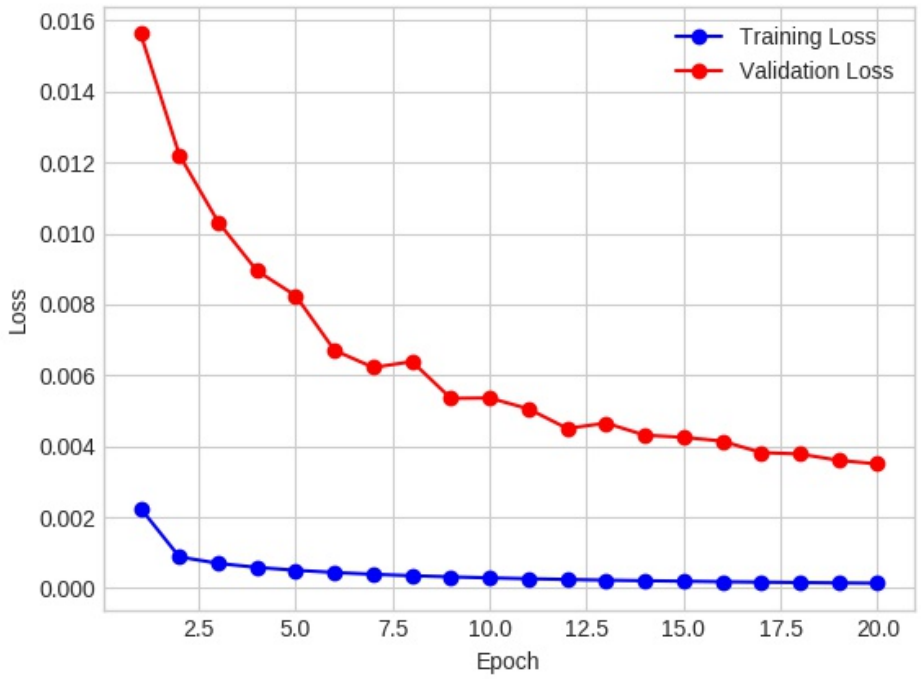
- Epoch Number: 20
- Learning Rate: 0.1
- Batch Size: 16
- Image Normalization: True

- Trained on: single CPU

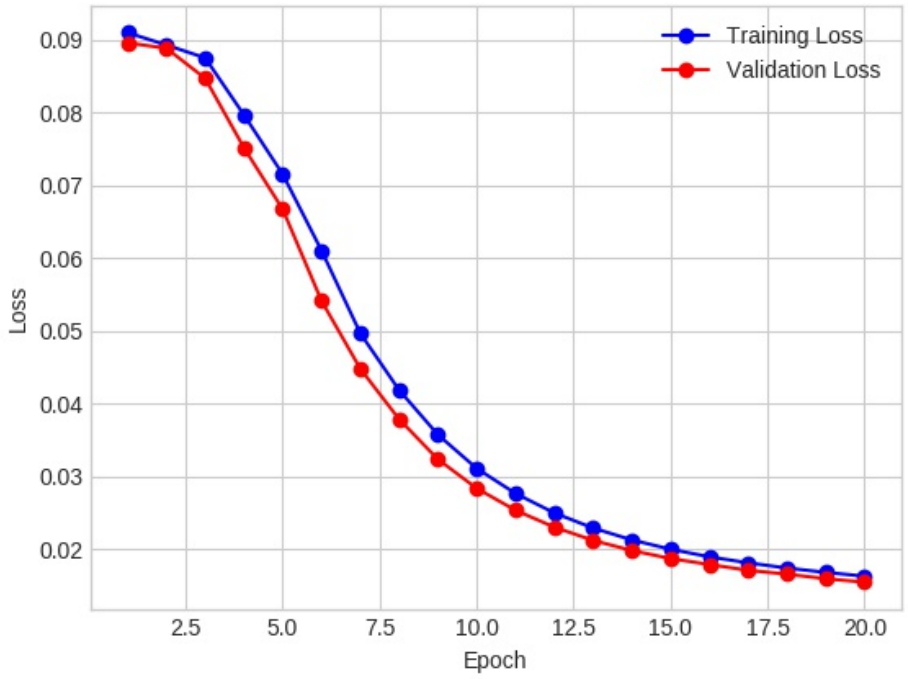
Run `python test.py`, and we get the following results:

Training and Validation Loss

The training and validation loss over epochs on our homemade neural network are shown below:



The training and validation loss over epochs on our pytorch implementation are shown below:

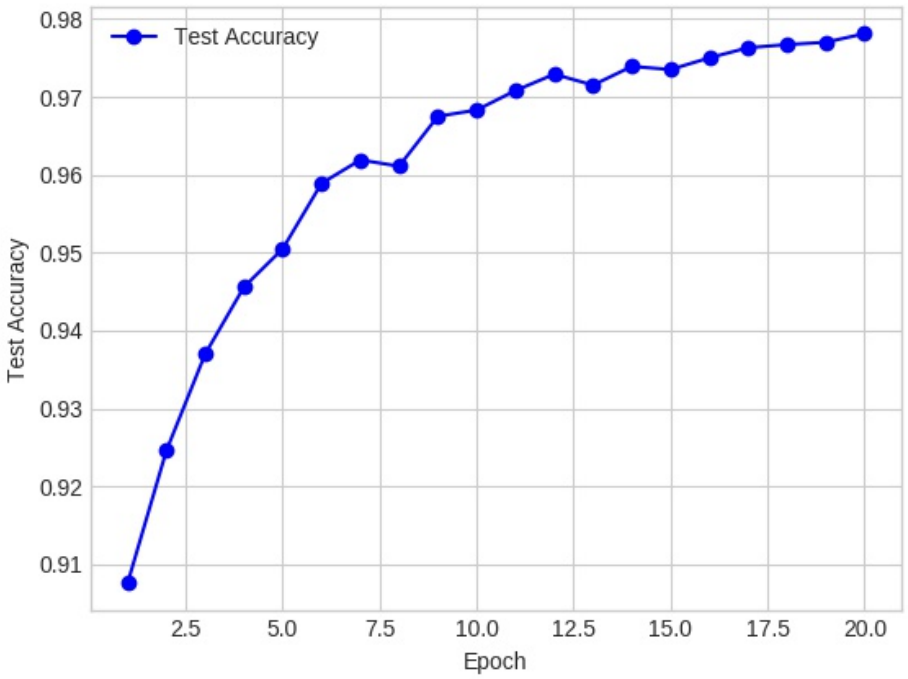


Compared the 2 results, we can find that our neural network can reach a lower loss quickly.

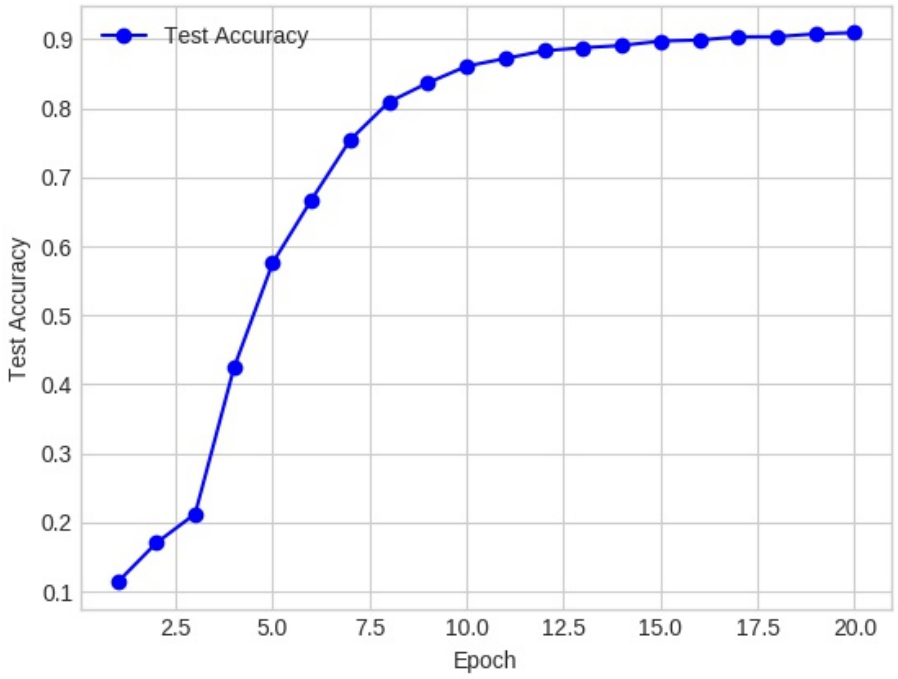
Test Accuracy

Test accuracy can also reflect the performance of network, which is calculated by: `accuracy = correct_num / total_num`

The test accuracy over epochs on our homemade neural network are shown below:



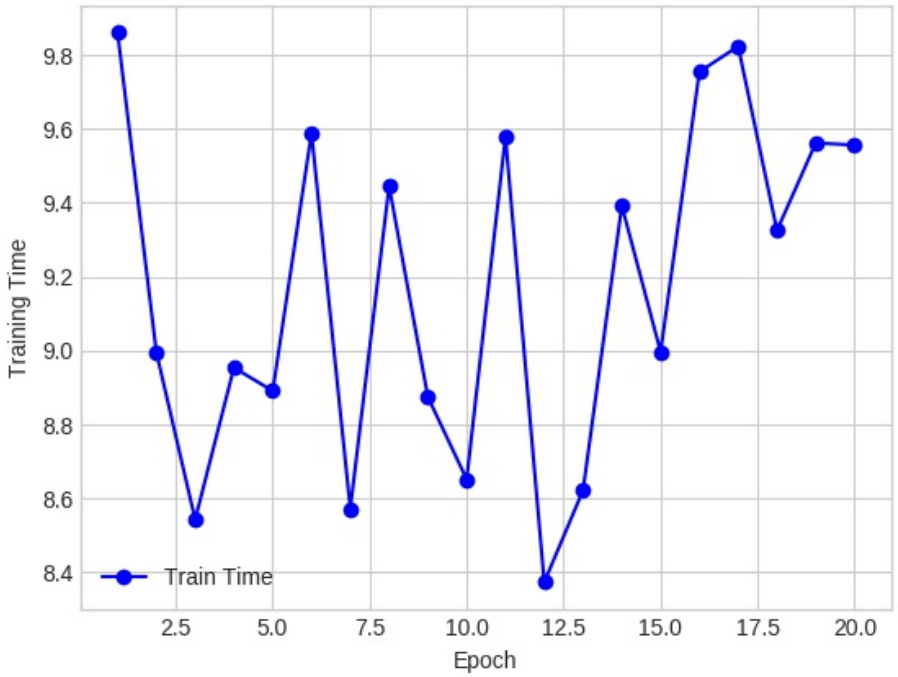
The test accuracy over epochs on PyTorch implemented network are shown below:



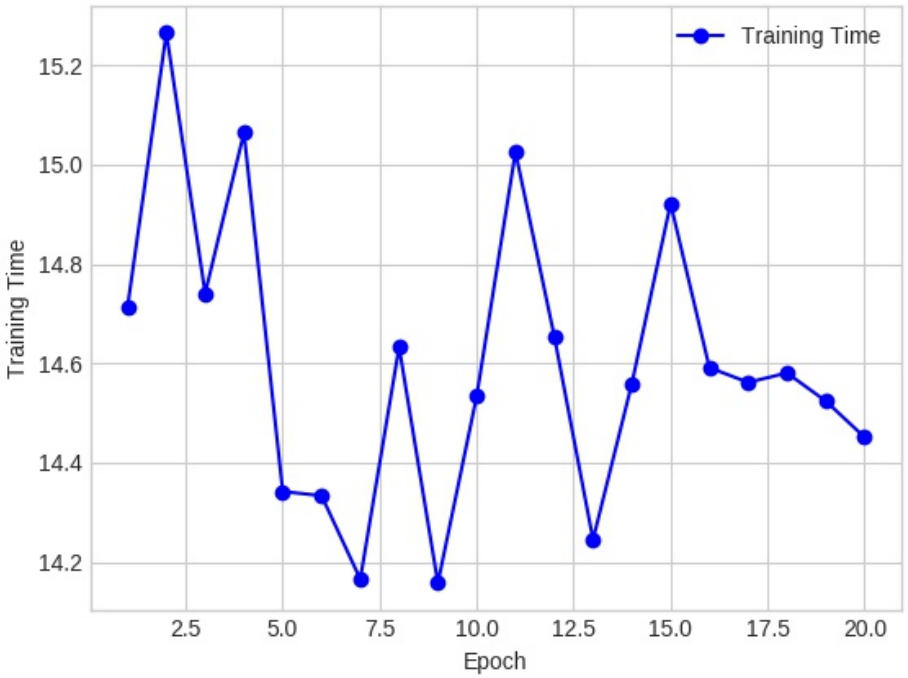
Both network's highest accuracy is achieved at the last epoch. Our network can achieve higher accuracy in less epochs.

Training Time

I plot the training time for each epoch below. The first plot is my network:



The next plot is PyTorch's network:



Our network's training time is less than PyTorch. Or to say, our network's running speed is faster.

Overall Comparision

To better compare the performance of our network and PyTorch's network, I made the following table that inludes our interested metrics:

Metric	My Network	PyTorch Network
Lowest Training Loss	0.0001416	0.0163000
Lowest Validation Loss	0.0035037	0.0154754
Highest Test Accuracy	0.9781	0.9094
Avarage Training Time / s	9.1683	14.6034

So we can make a safe conclusion based on the table above: our network outperforms the network implemented by PyTorch on this task.