Assignment 5: Backpropagation

Xiyu Wang

xw5638 xiyu_wang@utexas.edu

Apr. 26, 2020

1 Introduction

Backpropagation is a widely used algorithm in training feedforward networks. It computes the gradient of the loss function with respect to the weights of the network. The main idea of it is to break big functions in small parts and use partial derivatives to get function derivative with using the Chain Rule. When applying backpropagation algorithm, a neural network is trained, and some data analyze works are done according to the type of data and the purpose of the experiments. For instance, neural networks with backpropagation algorithm can do classification for a series of data. It can also do forecasting according to the data and their target value set.

In this report, the famous MNIST digits data set is used again to verify the classification with a back propagation based neural network. First, the MNIST is observed and unrolled to meet the requirement of the network. Then, a simple network was built to do the classification, during which process the changes of training loss during each epoch are shown as the result of backpropagation. After that, the result of the training and testing is evaluated by raising some evaluation methods. Following is a important part of the experiment. Different networks are trained with the same data set and tested for comparison. The parameters of the networks are compared for the possibilities when a backpropagation network is designed and built for the experiment.

2 Methods

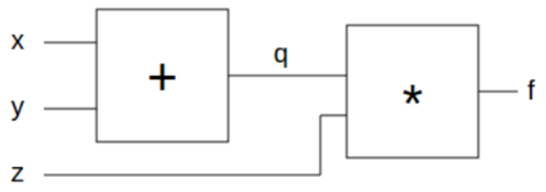
Backpropagation computes the gradient of the loss function with respect to the weights of the network. Generally, training a BP network model is solving the equation to minimize the loss function:

$$loss(model(X), Target) = 0$$

Since solving this equation can be pretty hard, the algorithm would do backpropagation and gradient descent by updating weights by a small amount based on the gradient to move in the way of loss minimization. The algorithm breaks big functions in small parts and use partial derivatives to get function derivative with using the Chain Rule:

$$\frac{df}{dx} = \frac{df}{dy} * \frac{dy}{dx}$$

Here is a classic example diagram of the idea. Assume there is a simple function with two nodes representing two operations when we do mapping from data to labels/targets:



$$f(x, y, z) = (x + y)z.$$

So for the partial derivatives, there are:

$$\begin{aligned} &Multiplication: \\ &\qquad (q * z)dq = 1 * z \\ &\qquad (q * z)dz = q * 1 \\ &Summation: \\ &\qquad (x + y)dx = 1 \\ &\qquad (x + y)dy = 1 \end{aligned}$$

And according to the Chain Rule:

$$\frac{df}{dx} = 1 * (z * loss)$$

$$\frac{df}{dy} = 1 * (z * loss)$$

$$\frac{df}{dz} = q * loss$$

Where the loss is from the last mapping with existing node weights which is the difference between the mapping result and the target value. In the following backpropagation process the network need to calculation partial derivatives for every operation node. In this way the weights are adjusted to the direction where the loss can be minimized, and the accuracy of the network will be increased as we want.

3 Results

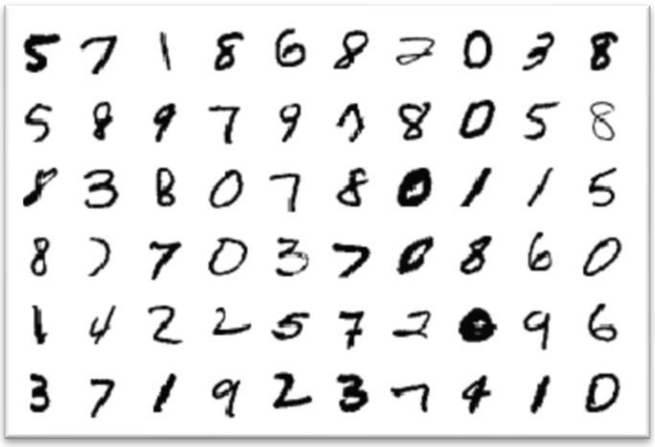
The experiment is based on the MNIST data set with hand writing digits. It contains the images in 28 * 28 size as well as corresponding labels for each digit from 0 to 9. A training set and a testing set are included for verification.

During the experiment, python3 is used and the PyTorch framework is adopted for building the backpropagation network. Different networks are attempted for analysing the effects of the activation functions, the number of laysers and the introduction of other types of layers such as convolutional layer.

3 (1) Data Observation

The given data contains the hand writing digit images in shape of [n, 28, 28] and the labels in shape[n] when there are n samples in an array. In each batch there are 64 images according to the suggestion form some publications. So in each batch, there are images in shape of [64, 28, 28] and labels in shape of [64].

Plot the image for observation:

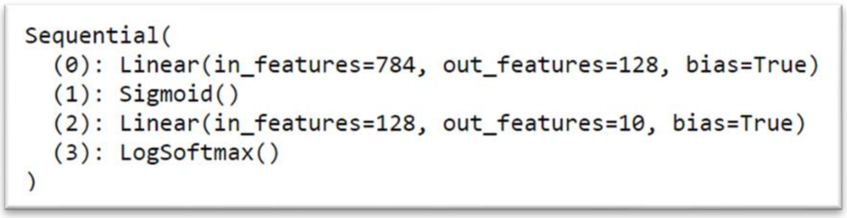


For training purpose, the image will be flattened into tensors in shape of [n, 784] to fit the network.

3 (2) Simple BP Neural Network

Using PyTorch framework, a simple backpropagation network is built for classification.

The structure of the network can be plot as shown:



```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): Sigmoid()
  (2): Linear(in_features=128, out_features=10, bias=True)
  (3): LogSoftmax()
)
```

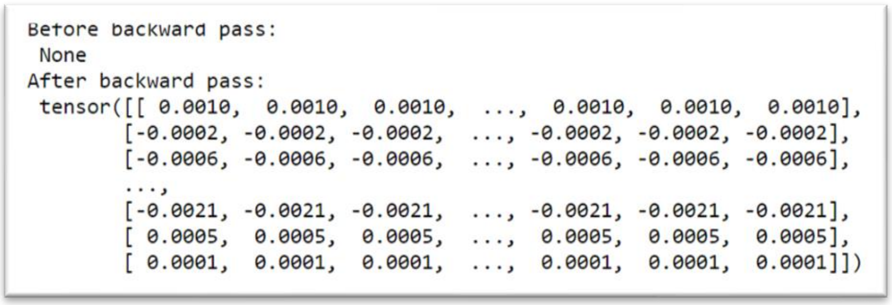
The network contains three layers: Input layer, hidden layer and output layer.

In this experiment, the input layer has the size of 784, which is the size of the flattened image. The hidden layer used Sigmoid function and the size is 128. For output, it's a LogSoftmax activation because it is a classification problem. The dim size of the LogSoftmax layer is 1 to meet the label dimension.

$$\mathrm{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

The negative log-likelihood loss of the network is defined by NLLLoss() function, which calculate the cross-entropy loss for the purpose of backpropagation.

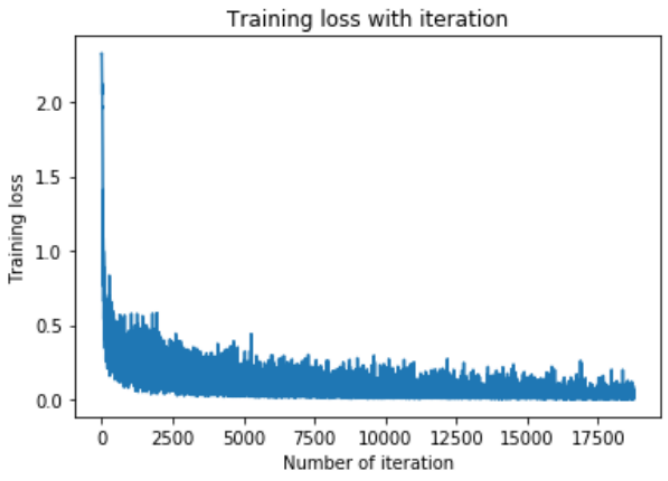
By iterate one time, the effect of backpropagation can be observed when we check the values of the weight:



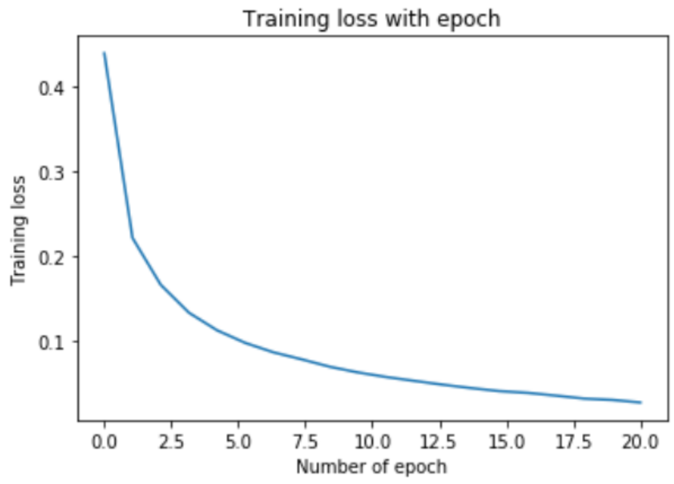
```
Before backward pass:
 None
After backward pass:
 tensor([[ 0.0010,  0.0010,  0.0010,  ...,  0.0010,  0.0010,  0.0010],
        [-0.0002, -0.0002, -0.0002,  ..., -0.0002, -0.0002, -0.0002],
        [-0.0006, -0.0006, -0.0006,  ..., -0.0006, -0.0006, -0.0006],
        ...,
        [-0.0021, -0.0021, -0.0021,  ..., -0.0021, -0.0021, -0.0021],
        [ 0.0005,  0.0005,  0.0005,  ...,  0.0005,  0.0005,  0.0005],
        [ 0.0001,  0.0001,  0.0001,  ...,  0.0001,  0.0001,  0.0001]])
```

Train the data by setting 20 epochs, recording the losses with different iterations and epochs. Also, record the training time for reference.

While feeding each image with its label into the network, the loss varies case by case but generally the tread is decreasing gradually.



By summing the loss up for each epoch and calculate the mean value, the loss for each epoch can be shown:



It is shown that the loss keeps decreasing when the epoch increases and the loss are lower than 0.03.

The total training time for this network is recorded:

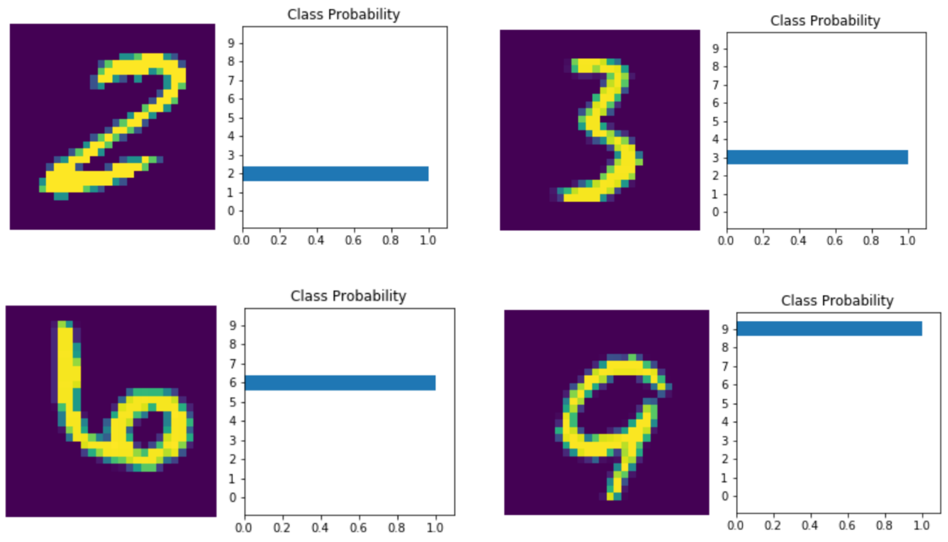
Training time/min	5.8777

3 (3) Testing and Evaluation

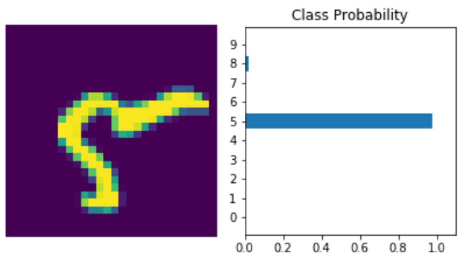
First, we can test the model by do some classification for the images from the test set.

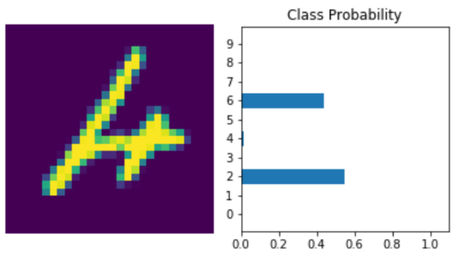
For each of the image, the model will return a log-prediction since we used LogSoftmax layer at the end. So first do exponential calculation for the result. Then, the result actually provided us an array with size 10 as the probability of the prediction. For the prediction, just find the one with maximum probability.

Also, we can plot the class probability for each result. There are images that there's almost no other probabilities:



There also can be some images that are difficult for the model and some other probabilities are provided. Some of them might be confusing and I think those are the main source of the error.



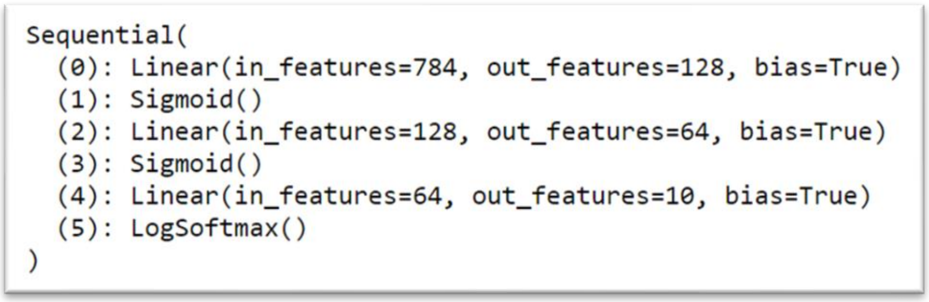


Above all, the testing accuracy can be simply calculated as the percentage of correct answers. The accuracy is **0.9700**, which is satisfying.

3 (4) Different Number of Layers

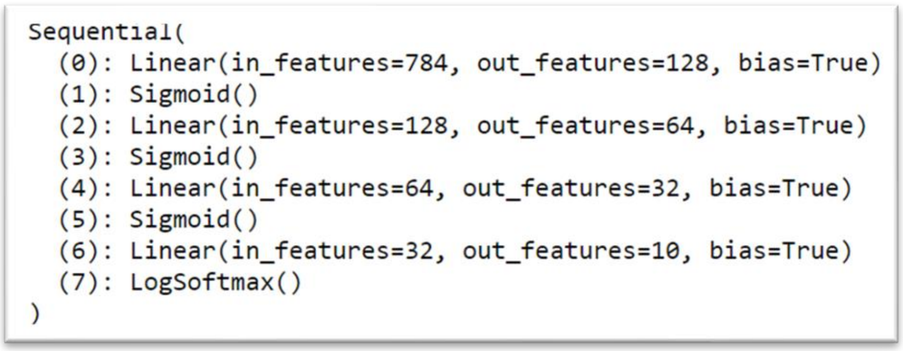
Apart from the simple 3 layer network, it's flexible when we introduce to more layers and compare the training time as well as the training loss changes.

For instance, a 4-layer network with two hidden layers can be shown as below:



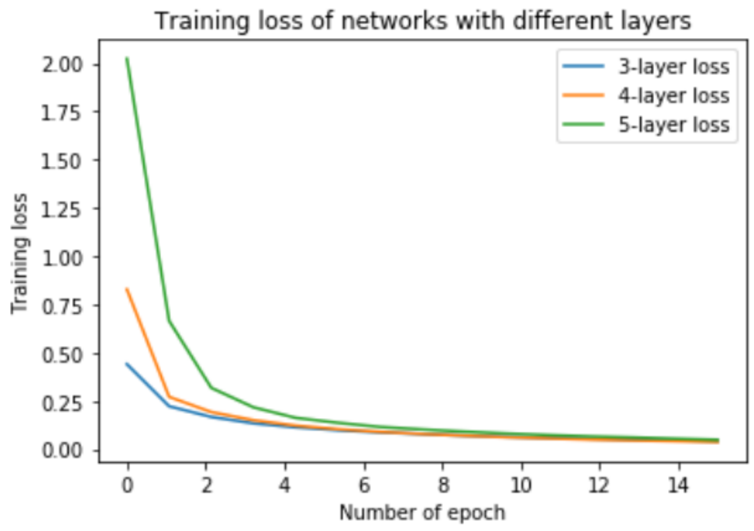
```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): Sigmoid()
  (2): Linear(in_features=128, out_features=64, bias=True)
  (3): Sigmoid()
  (4): Linear(in_features=64, out_features=10, bias=True)
  (5): LogSoftmax()
)
```

Similarly, the 5-layer with three hidden layers is like:



```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): Sigmoid()
  (2): Linear(in_features=128, out_features=64, bias=True)
  (3): Sigmoid()
  (4): Linear(in_features=64, out_features=32, bias=True)
  (5): Sigmoid()
  (6): Linear(in_features=32, out_features=10, bias=True)
  (7): LogSoftmax()
)
```

Collect the training loss of all the networks with the same dataset training, can do comparison.



It's shown that with more hidden layers, the initial loss is getting bigger since there are more parameters to train. With more layers, the loss decreases faster but reaching the similar value after all. In out case the loss, i.e. the accuracy is not affected significantly by the number of layers. The result might be that the case is not so difficult, and the extra layers are not only unnecessary but also increased the probability of more loss when doing backpropagation.

The training time is shown below:

	3-layer	4-layer	5-layer
Training time/min	6.8984	6.9591	7.0423

The loss for the last epoch:

	3-layer	4-layer	5-layer
Loss	0.0428	0.0387	0.0484

The accuracy of testing data:

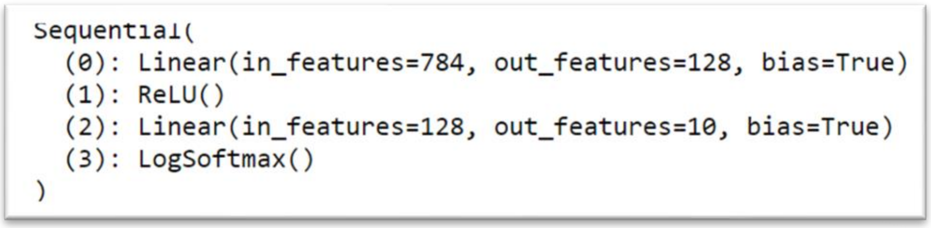
	3-layer	4-layer	5-layer
Accuracy	0.9700	0.9773	0.9699

According to the tables, the training time significantly increases but the loss is not getting much better. The 4-layer network might have better performance than the 3-layer with similar time for running it. The 5-layer does not provide us better performance. According to the testing accuracy, the 4-layer can improve the accuracy but 5-layer can not.

3 (5) Different Activation Function

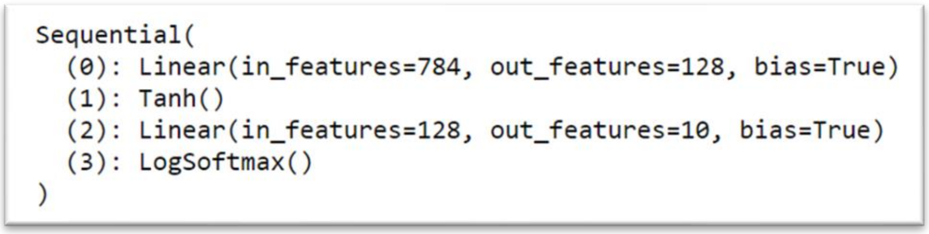
If we keep working with 3-layer networks, there are many options for the hidden layer. Varies of activations with different algorithms can provide the model different performance.

For instance, ReLU() can be in the place instead of Sigmoid():



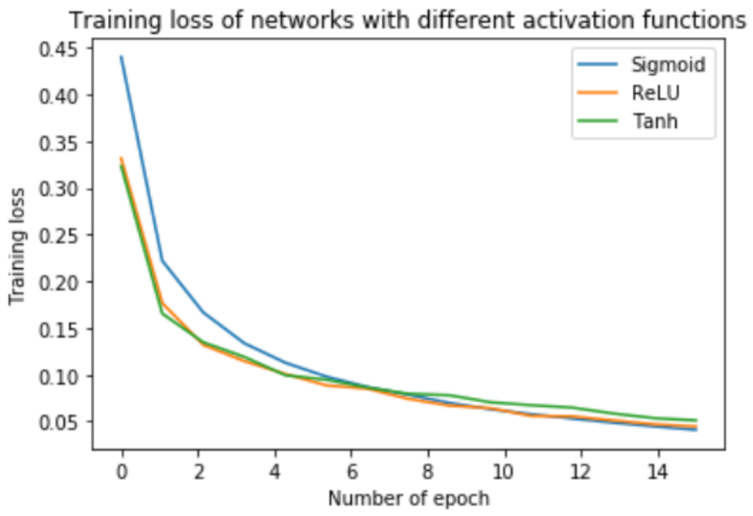
```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): ReLU()
  (2): Linear(in_features=128, out_features=10, bias=True)
  (3): LogSoftmax()
)
```

Also, Tanh() can be adopted:



```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): Tanh()
  (2): Linear(in_features=128, out_features=10, bias=True)
  (3): LogSoftmax()
)
```

For comparison, plot the training loss for different networks:



According to the diagram, for the start, ReLU and Tanh have better accuracy than Sigmoid. At the early epochs, ReLU and Tanh can fit the model faster than Sigmoid, too. But Sigmoid catches up at the 7th epoch and after that, ReLU and Sigmoid have similar fitting loss and Tanh has high fitting loss.

Similarly, the running time are shown below:

	Sigmoid	ReLU	Tanh
Training time/min	6.8984	12.6494	6.7245

The loss for the last epoch:

	Sigmoid	ReLU	Tanh
Loss	0.0428	0.0447	0.0511

It's shown that Sigmoid is slower and Tanh. ReLU is around 3 times of the time for Sigmoid.

The testing accuracy is:

	Sigmoid	ReLU	Tanh
Testing accuracy	0.9700	0.9672	0.9723

Interesting, During training, ReLU and Sigmoid have better loss performance than the Tanh to work as the hidden layer activation function. But for the testing part, ReLU is not as good as the rest two.

4 Summary

In this experiment, backpropagation algorithm is used to build networks for MNIST hand-writing digits classification. After flatting the image into tensors, a series of neural networks are attempted for the classification. Based on a simple 3-layer sigmoid-activated network, the networks are in different number of layers and different activation function for comparison.

First, the basic neural network is used for the classification. It gives us a great decrease of training loss and the testing probability is quite satisfying. Then, the result of the experiment shows that adding one layer as a 4-layer networks can slightly improve the accuracy but 5-layer ones are unnecessary for our case since the MNIST classification is not so complex. For the activation functions, the accuracy of Sigmoid is similar with the one of ReLU. Tanh has lower accuracy. But the drawback of ReLU is that it is much slower than the other two. Tanh has worse training accuracy but ok testing accuracy.