Info.6205 Project Report

A Neural Network Implementation For Handwritten Digits Recognize

Yuxuan Yang(001389098),

Haoqi Huang(001835259)

ABSTRACT

Neural networks are used as a method of deep learning, one of the many subfields of artificial intelligence. In this project, a small subsection of object recognition—digit recognition will be implemented, based on the MNIST database of handwritten digits. It includes building a neural network structure, training the model and accuracy evaluating.

STRUCTURE

1.Network

There are three layers in the neural network: Input, hidden and output layers, which is built by a one-dimensional array. Each element(an integer in this case) in the array represents one single node in the layer. Because the resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field[1], there are 28x28=784 input Neural and output layers contains 10 nodes that show the similarity between result and actual number(0-9).

2.Activation and cost functions

In this project, the activation interface is set up to normalize the process, which includes a guidance method for predicting and reverse guidance for weight fixing. The different activators could be switched by changing the parameter and sigmoid is the default.Also, the quadratic cost function to monitor how close we get to the expected result.

MODELING

1.Feedforward

In a word, we feed in the data and calculate the result. And that's where activation function comes in, we use the activation function to consider a neuron "fired" or not. The reason why we chose sigmoid function is that we want small changes in the weights and bias cause only a small change in the output. That is an important fact that will allow our sigmoid neurons to learn. Training and learning are crucial for an artificial neural network.

```
Start feed the hidden layer
feed value: 0.9998246002161049
origin value:8.648267294928193
feed value: 5.170704782225562E-5
origin value:-9.869864755842498
feed value: 0.07240761889380155
origin value:-2.5502808645976156
feed value: 0.9995783961215072
origin value:7.77102266854018
feed value: 0.9998546604219798
```

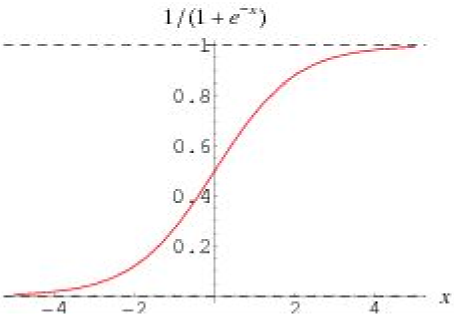
We use input and its weights plus bias to calculate the weighted sum, and take the weighted sum into our sigmoid function to compute the activation. Using the activation as the input to calculate the next layer's activation, we do this Layer by layer, at last, we will get our output.

2.Back-propagation

The backpropagation approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal[2]. The most important part of the training process, in the feed-forward process we get our output, but we want to know how far we are from the target. Error is calculated between the expected outputs and the outputs forward propagated from the network.

Our aim is to decrease the distance, in other words, the minimum the value of cost function.

In the previous feed part, We choose the sigmoid function, and its formula shows like[3]:



In order to calculate how much we need to fix between prediction and reality, we count the error into the derivatives of our activation method- in this case(sigmoid) it is

$$\frac{d}{dx}S(x) = S(x)(1 - S(x))$$

We can use the property of derivative, therefore, we can know which direction is the correct way to get to the minimum. So back-propagation is surrounded by one key factor that is partial derivative. The partial derivative of the cost function with respect to any weight and bias in the network. it will tell us how quickly the cost changes when we change the weights and bias.

Since we don't know the expected value until arriving at the very last layer, the direction would be the back layer to the front. In other words, we calculate the hidden layer error using the result we got in the output layer.

In this project, we only use sigmoid and tanh. The generic formula[4] is:

$$\delta^L = \nabla_a C \odot \sigma'(z^L).$$

BP1: Output error

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

BP2: Hidden layer error

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j.$$

BP3: Bias Change Equation

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j.$$

BP4: Weight Change Equation

BP1 is to get the error from the last layer (output), BP2 is used for calculating the error for the hidden layer, BP3 is to get the new bias and the last one BP4 is for weights.

3.Update weight and bias

To find out how we change the weights and bias, we have to know the error between the output and the expected result, each output neural more or less have some difference with the expected result. We can know whether cost function depends much on a particular output neuron, by calculating the partial derivative of activation of specific output neuron. If the error is small, then the cost function does not depend on that neuron. if the error is big, it means when we change the weights connected to this neuron and the bias on this neuron will effectively reduce the cost function.

Since we got the error value for every node in the hidden layer and output layer, we could update the weight and bias now with the learning degree.

Because of the equation: result=input*weight+bias, which is obvious, we could simply regard the total training progress as fix the weight and bias.

The implementations are like below. Note that the learning degree (n) for bias is 1/10 since it turns out that bias is actually more sensitive than weights.

```
//update weights from hidden to output
for(int i = 0; i < hiddenNeural; i++) {
    for(int j = 0; j < outputNeural; j++) {
        hiddenWeights[i][j]
        += (outputError[j] * hiddenOutput[i] * n);
        //ystem.out.println(hiddenWeights[i][j]);
    }
}
//update bias form output layer
for(int m = 0; m < outputNeural; m++) {
    outputBias[m] += (outputError[m] * (n/10));
}
```

The update result shows like the below screenshots:

```
origin hidden-output weights:0.7784438809371702
update to: 0.7784438809036628
origin hidden-output weights:-0.5093771237377779
update to: -0.5093771902612484
origin hidden-output weights:-0.41891407480616005
update to: -0.41891407529365876
origin hidden-output weights:-0.5714596660296514
update to: -0.5714596661103792
origin hidden-output weights:0.5623370689776958
update to: 0.562337068442108
origin hidden-output weights:-1.0612564055078968
update to: -1.0612565225655775
origin hidden-output weights:0.6447359352609557
update to: 0.5781725967398543
```

```
update to: -1.00631723130543
origin output bias: -0.5682440965578976
update to: -0.5682440965584016
origin output bias: -0.10129101511195532
update to: -0.10129108876125958
origin output bias: 0.9040803099761352
update to: 0.904080309976077
origin output bias: 0.5700159659972702
update to: 0.5700159658668689
origin output bias: -0.037945497431791424
update to: -0.03794550084706604
```

DELIVERABLES

Part1. the training process

Here you can see the change of cost function(Mean Square Error), as we mentioned before, to understand the variation of the cost function is a very important way to know how well our program has learned. We also can find our program bugs through MSE. So here is some screenshot.

```
0.048499929138995534
0.04724616092315616
0.04693241368074293
0.04195686525845711
0.043687934378906765
>>>>>>>>>>>>>>>>>   0   <<<<<<<<<<<<<<<<<<
0.040379772619202844
0.03976071413445911
0.04276599071725148
0.04118207625112017
0.04006043649286235
>>>>>>>>>>>>>>>>>   1   <<<<<<<<<<<<<<<<<<
0.03705951545286545
0.030588566856386615
0.03541322628595412
0.03557165212611026
0.03202906216231374
>>>>>>>>>>>>>>>>>   2   <<<<<<<<<<<<<<<<<<
```

Here is part of the output result divided by epoch(0, 1, 2), in each epoch, we have 5 MSE value, each MSE is the average MSE calculated from 100 random handwriting images, since each image through the program will generate an output, and through the expected result we can compute the MSE. We can clearly see that the MSE is reducing, that’s mean our program is learning well and making a good self adjust of weights and bias.

```
0.034903513955986405
0.030099759413834446
0.02548486144782896
0.022737831359049995
0.018532315537480134
>>>>>>>>>>>>>>>>>   0   <<<<<<<<<<<<<<<<<<
0.01576710990696851
0.015869447488996706
0.01603508013269236
0.012997841330167625
0.01228269119008553
>>>>>>>>>>>>>>>>>   1   <<<<<<<<<<<<<<<<<<
0.010726960442809406
0.008404266168124643
0.009173512482791501
0.008546437528492025
0.008788205420695923
>>>>>>>>>>>>>>>>>   2   <<<<<<<<<<<<<<<<<<
```

Up there, I change the number of training images for each MSE, instead of 100, I decided to add to 1000, and then get the above result. As you can see the MSE change more quickly than the previous one. That’s because we feed in much more training data than the first one. So the program will become more and more accurate when you train enough data. Also, we may notice that the MSE always fluctuates, sometimes it gets a bigger number than previous MSE value, It is normal because we give in random input. Let say we give it a number of hand-writing 5 digits, that is maybe the first time our program sees this number, so it makes sense we will get a larger error for the new stuff.

```
0.0024842927277099517
0.0019801873359757746
0.0020440586968376822
0.0023032812775366177
0.0022309629560886596
0.0018160608233893608
0.0021072096194123453
0.002643476271360989
0.0018348453293546142
0.001764088417118895
>>>>>>>>>>>>>>>>>   9   <<<<<<<<<<<<<<<<<<
```

Finally, we will get the MSE really small (but that is not good enough to get the best predict result), there are still some parameters that need to adjust, and we will discuss it later.

```
2.4203893329469902E-5
1.2727419612324622E-5
8.710373123753665E-6
6.639874446064906E-6
5.371325412605395E-6
>>>>>>>>>>>>>>>>>   0   <<<<<<<<<<<<<<<<<<
```

```
System gets 92 correct prediction from 1000 hand-writing images
```

Take a look at this training result of average MSE. in the first epoch of first 1000 "random" images, the MSE is too small to make sense, so there must be some problem with the random data we select, since we already pass the unit test of all the process in the feedforward and back-propagation part. After debugging, there is a problem in the random selection algorithm. We always chose the first data in the MNIST database, so that explains why we have such a small MSN and never fluctuate because we always training the same data! Therefore, the MSE can also help us detect the problem our program has.

Part2. Impact of hidden neurons

We keep all the variable unchanged, except the amount of the hidden neurons. We feed in 10 * 5 * 2000 random data, 10 epoch, each epoch loop 5 times, each loop train 2000 random images. And we got:

```
time consume: 17208.0 millisecs
System gets 921 correct prediction from 1000 hand-writing images
```

For this result, we got 50 hidden neurons. If we increase hidden neurons to 100:

```
time consume: 37041.0 millisecs
System gets 954 correct prediction from 1000 hand-writing images
```

If we increase hidden neurons to 150:

```
time consume: 56115.0 millisecs
System gets 955 correct prediction from 1000 hand-writing images
```

If we increase hidden neurons to 200:

```
time consume: 75401.0 millisecs
System gets 655 correct prediction from 1000 hand-writing images
```

It is easy to find out that the total time for the same amount of training data increase when you got more and more hidden neural. As for the correct rate predicted by the program, it increases when we have more hidden neurons but it seems to have a peak, after that the rate will decrease drastically. So we guess, that's maybe related to our learning rate. As a result, it is important to find out the right amount of hidden neurons, which will get the shortest time and highest correct rate.

Part3. explore the mystery in the learning rate

The reason why the learning rate is important is that it affects the training process directly if we give the value to the learning rate too high, we will not get the result we want, even worse, the result is confusing. If it is too low, we will not see the remarkable changing in our correct rate. But, it is always recommended to set the learning rate lower.

```
System gets 103 correct prediction from 1000 hand-writing images
```

This is the result we got, for the learning rate is 0.3 at the first time we ran our program.

```
0.03895622151821757
0.03799081501136365
0.04348888283294399
0.05351325020489515
0.03851025197136675
>>>>>>>>>>>>>>>>>   0   <<<<<<<<<<<<<<<<<
0.053309072151936615
0.04113869769300794
0.04701946733096267
0.04077490138294239
0.041858740003878146
>>>>>>>>>>>>>>>>>   1   <<<<<<<<<<<<<<<<<
0.04099605419734348
0.04020915189240252
0.05211316162562642
0.04646080750595501
0.03771325452993475
>>>>>>>>>>>>>>>>>   2   <<<<<<<<<<<<<<<<<
```

```
System gets 963 correct prediction from 1000 hand-writing images
```

The screenshot above is the first 3 epoch training result before we get the test result. We can get pretty much information through this, we can see the MSE fluctuate in an abnormal pattern. The right thing is, although some average MSE gets higher, it will decrease as a whole. This result tells that our program doesn't have a steady learning state, which will blame the high learning rate.

This is the result I accidentally change the bias into test bias, which means all the bias will be one and we don't update bias. And really surprise, we get a really good result. So in this case, I think the sensitivity of the learning rate for the weights and bias are different, in other word, bias needs a relatively small learning rate than the weights do.

So we try to give the bias different learning rates, to find a result, at least, as good as the upper one. Finally, we got this, with the 0.3 learning rate of weights and 0.3/35 learning rate of bias:

```
System gets 966 correct prediction from 1000 hand-writing images
```

We can't guarantee that is the best learning rate, but it's the best we been so far.

Part4. Extend: can we do better?

As we explore the result so far, we have known what is the important factor that affects the final result. We need to adjust the total number of our hidden neurons, we can enlarge our training date, we need to find a better learning rate. Consider all the factor, we ask ourself, can we do better? what is the best result can our program reach, as we try many times, we twist the parameters again and again, to reach the highest rate we can make.

```
5.274069639672489E-4
0.001009060519367504
6.812483937983991E-4
9.036958935802942E-4
9.60144852554637E-4
>>>>>>>>>>>>>>>>>   19   <<<<<<<<<<<<<<<<<
time consume: 210648.0 millisecs
System gets 983 correct prediction from 1000 hand-writing images
```

CONCLUSION

After we finish this project, we are more familiar with the AI. It seems like AI is not reachable as we thought at first. It turns out, especially for this ANNs project, mathematical knowledge is more important, we understand how the network works, the implementation part will be quite clear. We know more about the array data structure, because from start to end we run through the program with arrays and metrics, we got a whole new understanding about the partial derivative and gradient descent.

Reference

[1]http://yann.lecun.com/exdb/mnist/
[2]https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/
[3]https://en.wikipedia.org/wiki/Sigmoid_function
[4]http://neuralnetworksanddeeplearning.com/chap2.html#the_four_fundamental_equations_behind_backpropagation