Neural Network:
Classifying Handwritten Number

Dyanne Macalinao
PHYS 300
Dr. Zhu

I. Introduction

Machine learning provides computers the capability to learn without being explicitly programmed. In this project, machine learning will be used to solve a classification problem. In machine learning, classification is a supervised learning approach where a computer learns from data inputs given to it, and then classifies new data based on its previous observations. For this report, a model will be trained to classify handwritten numbers (from 0 to 9) obtained from the Modified National Institute of Standards and Technology database.

II. Data Preparation

First thing to do before getting started with building a neural network, is to obtain and load the data. The dataset used in this project is a popular dataset from the Modified National Institute of Standards and Technology (MNIST) database. The dataset contains 70000 28 x 28 images of handwritten numbers. The data is already flattened and so it is 70000 x 784 when uploaded. A very important part of preparing the data is turning the y-values (in this case the labels 0 to 9), which are categorical, to binary numbers that the computer will better understand. This will be done through one-hot encoding. Once this is done, the data is split into training (first 60000 images) and test data (last 10000).

III. Building a Neural Network

There are three main parts in building a neural network model after preparing the data: initialization, forward and back propagation.

A. Initialization

The first part in building a neural network model is initializing parameters. This includes initializing weights and bias that would then be used for back propagation. The weights are generated randomly. These random weights are needed in stochastic optimization algorithm that neural network uses called stochastic gradient descent. Random weight breaks symmetry and gives better accuracy. In addition, we can normalize variance of each output to one by scaling the weight vector by multiplying it with the square root of the number of its output. Now that the parameters are initialize, forward propagation can ensue.

B. Helper Functions

Before moving on to what forward propagation is, let's first identify the helper functions that we will be using in forward propagation. There are two function we need called activation and cost functions. There are many types of activation functions. These include: ReLu, tanh, and logistic (Sigmoid) functions. For this report, the Sigmoid function will be utilized. The purpose of the activation function is to introduce non-linearity to the network. The function converts input signal to an output signal. Just like probability, Sigmoid exists between 0 to 1, which is why it is ideal to use. Furthermore, since the dataset deals with multiple classes (10 outputs), the Softmax function will be used in the output layer. This function normalizes an input value into a vector that follows a distribution that adds up to one.

$$y = \frac{1}{1 + \exp(-x)}$$

Eq. 1 Sigmoid Function

$$Softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Eq. 2 Softmax Function

Another helper function is the cost function. Its role is to determine "how good" a network did with respect to its training sample and the expected output. Similar to the activation function, there are also several types of cost functions: quadratic, cross entropy, and exponential, to name a few. In this particular model, the cross entropy function will be used.

$$L(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}).$$

Eq. 3 Cross Entropy

$$L(Y, \hat{Y}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right)$$

Eq. 4 After averaging over a training set of *m* examples.

Now that both activation and cost functions are chosen, a model can now be trained.

C. Forward Propagation

In forward propagation, the inputs provide the initial information that propagates to hidden units at each layer, and produce the output. Each hidden layer accepts the input data, processes it as per that activation functions and passes to the successive layer.

$$\hat{y} = \sigma(w^T x + b),$$

D. Backward Propagation

After computing the total cost, it's essential to improve the performance of the neural network on the training data. This is done in backward propagation. To improve, weights and bias should be updated, which would lower the cost. To know how much the specific weights and bias affect the total cost, partial derivatives of the total cost with respect to the weight and bias will be calculated. This will be done by using the Chain rule. For the purpose of this report, only one calculation will be shown:

$$\begin{aligned}
\frac{\partial L}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}}(-y\log(\hat{y}) - (1-y)\log(1-\hat{y})) \\
&= -y\frac{\partial}{\partial \hat{y}}\log(\hat{y}) - (1-y)\frac{\partial}{\partial \hat{y}}\log(1-\hat{y}) \\
&= \frac{-y}{\hat{y}} + \frac{(1-y)}{1-\hat{y}} \\
&= \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}.
\end{aligned}$$

Chain Rule:

$$\begin{aligned}
\frac{\partial}{\partial z}\sigma(z) &= \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right) \\
&= -\frac{1}{(1+e^{-z})^2}\frac{\partial}{\partial z}\left(1+e^{-z}\right) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}} \\
&= \sigma(z)\frac{e^{-z}}{1+e^{-z}} \\
&= \sigma(z)\left(1-\frac{1}{1+e^{-z}}\right) \\
&= \sigma(z)\left(1-\sigma(z)\right) \\
&= \hat{y}(1-\hat{y}).
\end{aligned}$$

After calculating the partial derivatives with respect to the cost, weights and biases can be updated by multiplying the learning rate and substracting the results from the weights and bias:

$$W_1 = W_1 - (\alpha \times \frac{\partial C}{\partial W_1})$$

E. Hyperparameter Tuning

The optimization technique in part D can be referred to as batch gradient descent. This method is usually slow and inaccurate, so another method, called mini-batch gradient descent, will also be used.

In mini-batch gradient descent, the cost function is averaged over a small number of samples. This is opposed to the batch gradient, where the cost function is averaged over all of the training samples. The mini-batch method is computationally more efficient.

Aside from changing optimization methods, hyperparameters can also be changed. Hyperparameters are variables that determine the network structure and how the network is trained. Hyperparameters are set before training. In this project, the number of units in the hidden layer and learning rate will be tuned. The hidden layer is the layer in between the input and output layer. Having a small number of hidden units can cause under fitting. Meanwhile, many hidden units can increase. Furthermore, learning rate defines how quickly a network updates its parameters. If it is low, training will perform very slowly, while too large, the cost function might not converge.

With all these steps explained, training can be started. The code for all of the phases can be viewed here.

IV. Results

As mentioned in part B, batch- gradient descent and mini-batch gradient descent are used. After training and testing our datasets, the following results are acquired for batch and mini-batch, respectively.

	precision	recall	f1-score	support
0	0.79	0.77	0.78	995
1	0.90	0.86	0.88	1183
2	0.63	0.63	0.63	1037
3	0.67	0.65	0.66	1039
4	0.61	0.64	0.62	933
5	0.48	0.53	0.50	806
6	0.74	0.70	0.72	1013
7	0.73	0.74	0.74	1017
8	0.52	0.57	0.54	896
9	0.61	0.57	0.59	1081
accuracy			0.67	10000
macro avg	0.67	0.67	0.67	10000
weighted avg	0.68	0.67	0.67	10000

TABLE 1: Classification report for batch gradient descent.

	precision	recall	f1-score	support
0	0.98	0.94	0.96	1024
1	0.98	0.96	0.97	1151
2	0.89	0.93	0.91	990
3	0.91	0.89	0.90	1032
4	0.93	0.91	0.92	999
5	0.86	0.91	0.88	843
6	0.94	0.93	0.93	971
7	0.92	0.93	0.92	1007
8	0.88	0.89	0.88	970
9	0.90	0.90	0.90	1013
accuracy			0.92	10000
macro avg	0.92	0.92	0.92	10000
weighted avg	0.92	0.92	0.92	10000

TABLE 2: Classification report for mini-batch gradient descent.

According to the classification reports in Table 1 and 2, using batch gradient descent has a much lower accuracy, 68%, than the mini-batch gradient descent, 92%. In addition, the batch gradient descent had more epochs (100 compared to 20 with mini-batch), but still performed worse. The mini-batch gradient descent method is used for the rest of this report to obtain the highest accuracy.

Two of the hyperparameters, number of hidden units and learning rate, are tuned to see which values are optimal. The hidden units are tested from one to eighty. In the figures below, it can be seen that having few hidden units are not good (training and test cost are about 1.74). However, the difference between 20-80 units do not have much differences. The training cost for 21,41,61, and 81 hidden units are 0.19, 0.15, 0.13, 0.14. It is important to note that after 80 units, the cost starts increasing a little. With these observations, 75 is chosen to be used in the next trials.

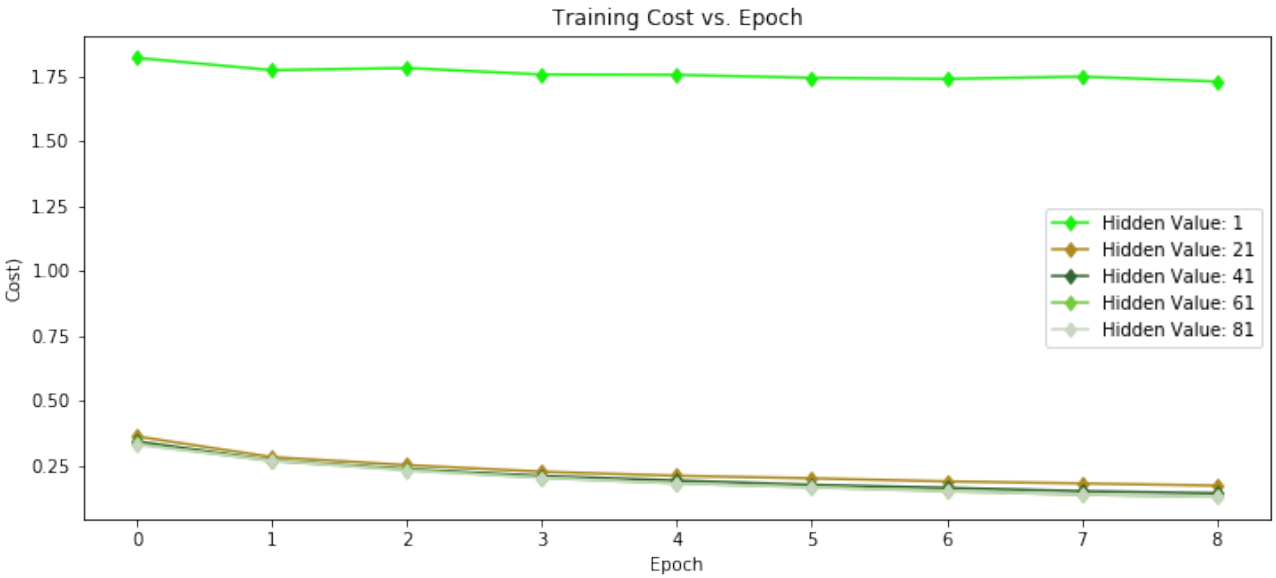

Fig. 1 Training cost vs. Epoch with varying hidden units.

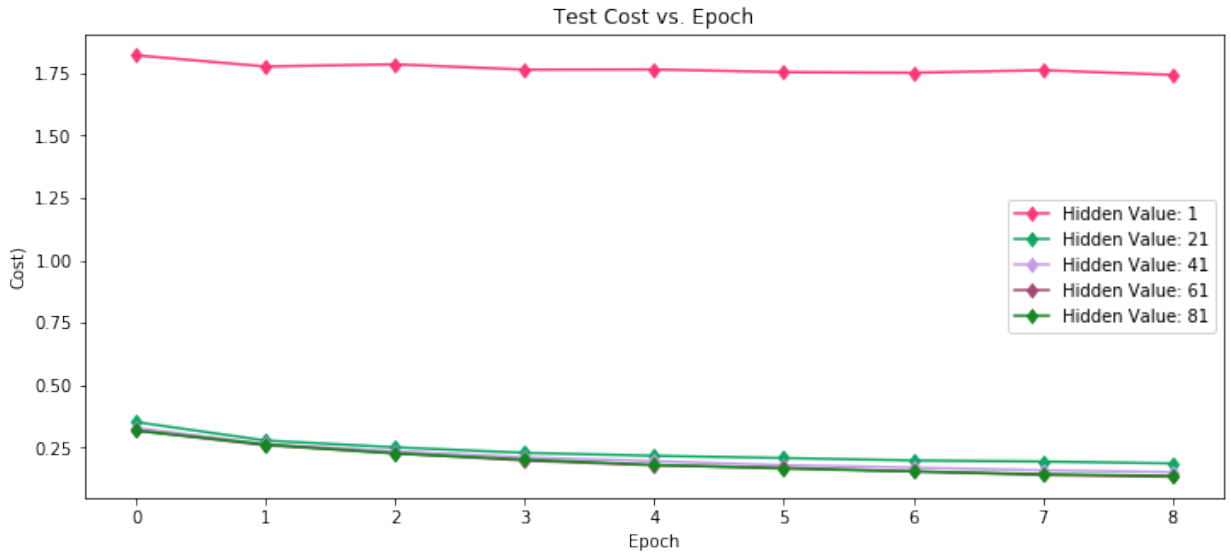

Fig. 2 Test cost vs. Epoch with varying hidden units.

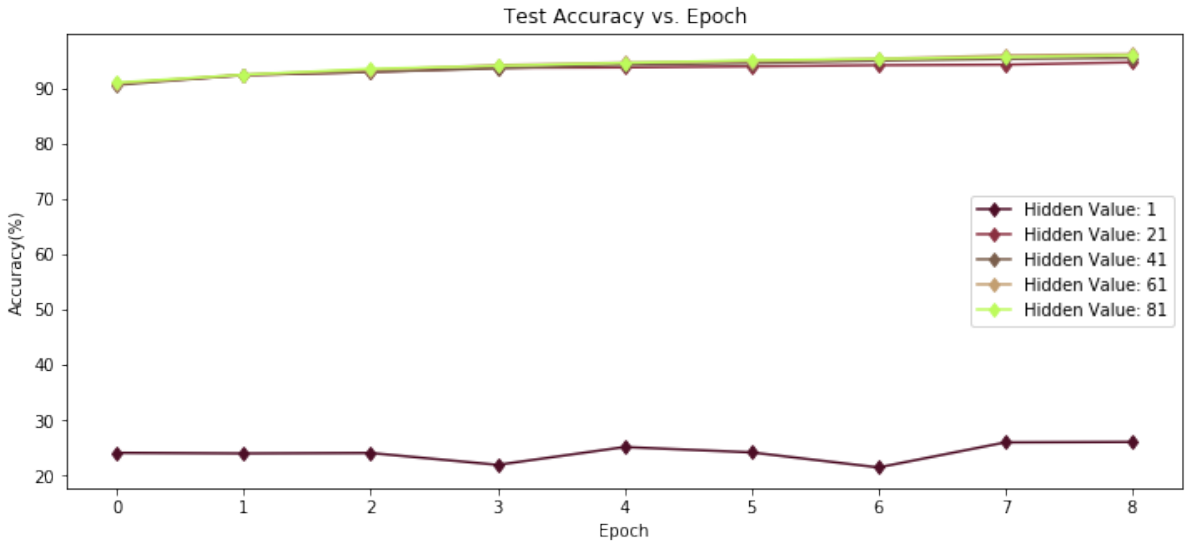

Fig. 3 Test accuracy vs. Epochs with varying hidden units.

```
Number of Hidden Units: 1
Total Time: 4.541866302490234
Final train cost: 1.7431476612050538         Final test cost: 1.7305717831378384
Done.

Number of Hidden Units: 21
Total Time: 7.920375108718872
Final train cost: 0.18697290433191802         Final test cost: 0.17177162894171247
Done.

Number of Hidden Units: 41
Total Time: 10.837084293365479
Final train cost: 0.1519054107386974         Final test cost: 0.14214030620211915
Done.

Number of Hidden Units: 61
Total Time: 12.305041313171387
Final train cost: 0.13335308529844953         Final test cost: 0.1284024869836345
Done.

Number of Hidden Units: 81
Total Time: 14.411478042602539
Final train cost: 0.13657449696290394         Final test cost: 0.12726884943981104
Done.
```

Fig. 4 Training and Test costs with varying hidden units

The next hyperparameter that was varied is the learning rate. The following values were tested: .001, .01, .1, and 1.0. It is clear that the highest learning rate, 1.0, has the least cost(.25) as seen in Figure 2. The relationship between the cost seems to inversely proportional since the higher the learning rate, the lower the cost. However, to avoid over fitting, .9 is chosen to be used for the final training.

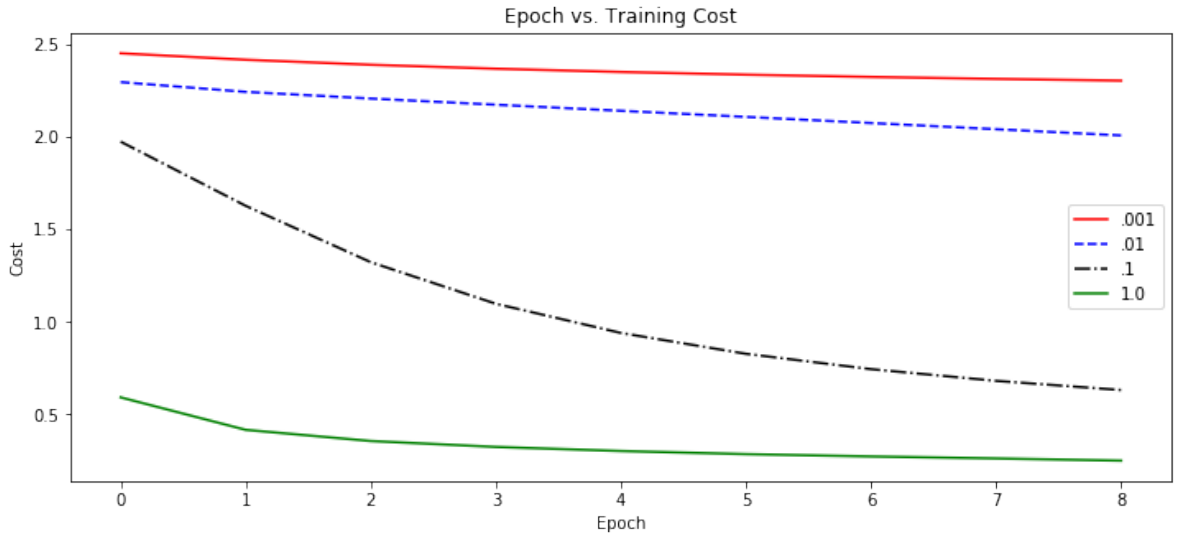

Fig. 5 Epoch vs. Training cost with varying learning rates.

```
LR: .001
Total Time: 10.744680881500244
Final train cost: 2.3060085600956715         Final test cost: 2.305168577219493
Done.

LR: .01
Total Time: 10.767755031585693
Final train cost: 2.0046016282157626         Final test cost: 2.008832008742616
Done.

LR: .1
Total Time: 10.264938116073608
Final train cost: 0.6146314668024857         Final test cost: 0.6323922838991939
Done.

LR: 1.0
Total Time: 10.411981105804443
Final train cost: 0.24606871506661288         Final test cost: 0.25065870522166483
Done.
```

Fig. 6 Training and Test costs with varying learning rates.

Setting the hidden units and learning to 75 and .9 respectively, the following results (table 3) were obtained. After tuning the hyperparameters, a 2-3% increase in accuracy was acquired. Comparing this to the accuracy of the built-in Scikit Learn MLPClassifier function (table 4), it is 2-3% lower even with the almost the same parameters.

	precision	recall	f1-score	support
0	0.98	0.95	0.97	1011
1	0.98	0.97	0.98	1152
2	0.94	0.94	0.94	1027
3	0.94	0.94	0.94	1017
4	0.95	0.93	0.94	1000
5	0.91	0.94	0.93	868
6	0.96	0.94	0.95	974
7	0.93	0.95	0.94	1007
8	0.93	0.94	0.93	959
9	0.91	0.94	0.92	985
accuracy			0.94	10000
macro avg	0.94	0.94	0.94	10000
weighted avg	0.95	0.94	0.94	10000

Table 3: Results after tuning hyperparameters.

	precision	recall	f1-score	support
0.0	0.99	0.97	0.98	998
1.0	0.99	0.98	0.99	1139
2.0	0.97	0.96	0.97	1044
3.0	0.97	0.96	0.97	1018
4.0	0.97	0.96	0.97	992
5.0	0.96	0.96	0.96	897
6.0	0.97	0.97	0.97	961
7.0	0.97	0.98	0.97	1014
8.0	0.95	0.98	0.96	947
9.0	0.95	0.97	0.96	990
accuracy			0.97	10000
macro avg	0.97	0.97	0.97	10000
weighted avg	0.97	0.97	0.97	10000

Table 4: Results using MLPClassifier.

V. Conclusion and Discussions

The goal of this project was to build a neural network system based on NumPy. A neural network consists of the following steps: parameter initialization, forward propagation, and backward propagation. Using these steps, a model was created. To make the most optimal model, the optimization method and hyperparameters were tweaked. Compared to the built-in Scikit function, MLPClassifier, the network built are almost similar in accuracy, 97% for MLPClassifier and 95% for the network built based on Numpy. To further increase the accuracy of the Numpy-based network, other hyperparameters that weren't tuned in this project can be changed. These hyperparameters include batch size and number of epochs. Another optimization technique, called ADAM, can also be tested.

References

Britz, D. (2016, January 10). Implementing a Neural Network from Scratch in Python – An Introduction.Retrieved fromhttp://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/.

Hardesty, L., & MIT News Office. (2017, April 14).

(n.d.).Retrievedfrom http://cs231n.github.io/optimization-1/.