# Project #[Neural Networks]

CMSC 25400, Prof. Kondor
Ken Chen

## Description of the Code

### 1. Structure of the Neural Network

As suggested in the homework, I set the Neural Network to be a fully connected one. It is characterized with the following features:

- **Input layer**: The input layer has 785 neurons, which consists of 784 pixels and 1 bias neuron.
- **Hidden layers**: The hidden layers ($l = 1, 2, ...L-1$) are fully connected, and each layer has the same $N$ number of neurons. Later I will entertain with $L$ and $N$ to determine the optimal parameters.
- **Output layer**: The output layer has ten neurons. To make predictions, the outputs will be transformed into probability interpretations by the softmax function $p(y, a^L) = e^{[a^L]y} / \sum_{i=1}^{10} e^{[a^L]i}$.

The Neural Network structure is generated by using the function **layer-dims()**.

### 2. Parameters to Estimate

The parameters we have to estimate are linear weights and biases. Let's denote the $W$ as,

$$W = (W^1, W^2, \ ... \ W^L)$$

where $W^{(l)} \in \mathbb{R}^{N_{l-1} \times N_l}$. And the biases,

$$B = (B^1, B^2, \ ... \ B^L)$$

where $B^{(l)} \in \mathbb{R}^{N_l}$. In my code, these parameters are initialized by using the function **initialize()**. I initialized these weights and biases using the standard normal distribution, then times 0.1 to avoid getting big activation values and small gradients at the sigmoid functions. The inputs and outputs at each layer are governed by these parameters,

$$a^{l+1} = \phi(a^l W^{l+1} + b)$$

where $a^l$ denotes the activated output at layer $l$, $l \in \{0, 1, 2, ..., L-1\}$. The linear transformation is completed by function **linear-forward()** and the nonlinear transformation is completed by the sigmoid function **sigmoid-forward()**.
* Special note: $a^0$ is simply the raw data from the samples, and the final prediction of probabilities is determined by the softmax function $p(y, a^L)$.

**3. Loss function**

I used the L2-norm loss,

$$\epsilon = \left\|a^L - y\right\|_2 = \sum_{i=0}^{9}(a_i^L - y_i)^2$$

The label $y$ is defined as $y_i = 0$ if $i \neq k$ and $y_i = 1$ if $i = k$, where $k$ is the true label. The loss can be computed using the **comp-loss()** function

**4. Weight Updating**

The general strategy to implement gradient descent is:

$$w_{s\to t} \leftarrow w_{s\to t} - \eta\frac{\partial\epsilon}{\partial w_{s\to t}}$$

Here the $\eta$ parameter is defined as our learning rate, which indicates how fast we update our weights. The derivative of $\epsilon$ with respect to $w_{s\to t}$ is given by using the chain rule,

$$\frac{\partial\epsilon}{\partial w_{s\to t}} = \frac{\partial\epsilon}{\partial a_t}\frac{\partial a_t}{\partial z_t}\frac{\partial z_t}{\partial w_{s\to t}}$$

where $z_t$ denotes the linear output, $a^l W^{l+1} + b$, at each layer. At the output layer $L$, we have the following results:

$$\frac{\partial\epsilon}{\partial a_t^L} = 2(a_t^L - y_t)$$
$$\frac{\partial a_t^L}{\partial z_t^L} = \phi'(z_t^L)$$
$$\frac{\partial z_t^L}{\partial w_{s\to t}^L} = a_s^{L-1}$$

So briefly, we can rewrite the backward propagation process as:

$$\frac{\partial\epsilon}{\partial a_t^L} = \delta_t^L a_s^{L-1}$$

By storing the $\delta_t$, we can propagate the process backward. Whereas for the hidden layers,

$$\delta_t^{l-1} = \phi'(z_t^{l-1})\sum_{u\in O^l}\delta_u w_{t\to u}$$

and then the propagation can be progressed backward. The derivative with respect to $b^l$ is essentially the same of derivatives with respect to $w_s^l$, and the associated $a_s^{l-1}$ is 1. To vectorize the computation, we have the following two 'passing' functions,

$$\delta^L = 2(a_t^L - y_t) \otimes \phi'(z_t^L)$$
$$\delta^{l-1} = \phi'(z_t^{l-1}) \otimes (W^l(\delta^l)^\top))^\top$$

where the $\otimes$ notation denotes the element wise product. Then, our weights updating strategy is explicitly,

$$W^l \leftarrow W^l - \eta(a^{l-1})^\top \delta^l$$

The backward propagation can be implemented using the function **backward-propagate()**, which takes advantage of two other ancillary functions **linear-backward()** and **sigmoid-backward()**.

## Training and Results

**1. Training with minibatches**
My training strategies follow the Stochastic Gradient Descent algorithm, where I feed the algorithm with minibatches of the same size each time, and then we update the weights and biases right away. The training process will be repeated multiple times equal to the value we define as epoch.

**2. Experimenting with different parameter settings**
As the increasing the epoches will increase the time expense linearly, I first started playing with the other four tunable parameters $N_l, L, \eta, b$, namely the number of neurons at the hidden layers, the number of layers, learning rate and the size of the minibatches. The default settings of the parameters are configured as $N_l = 256, L = 3, \eta = 0.1, b = 100$. What I did is simply varying the parameter I am testing with, and fixed all the other parameters at their default values, repeating 8000 times. The plot is shown in **Figure 1** below.
The essential takeaway from these experiments is that: the error rate decreases as the number of neurons, the learning rate and the size of the minibatches increases. For the number of layers, the minimum error rate occurs at 3.

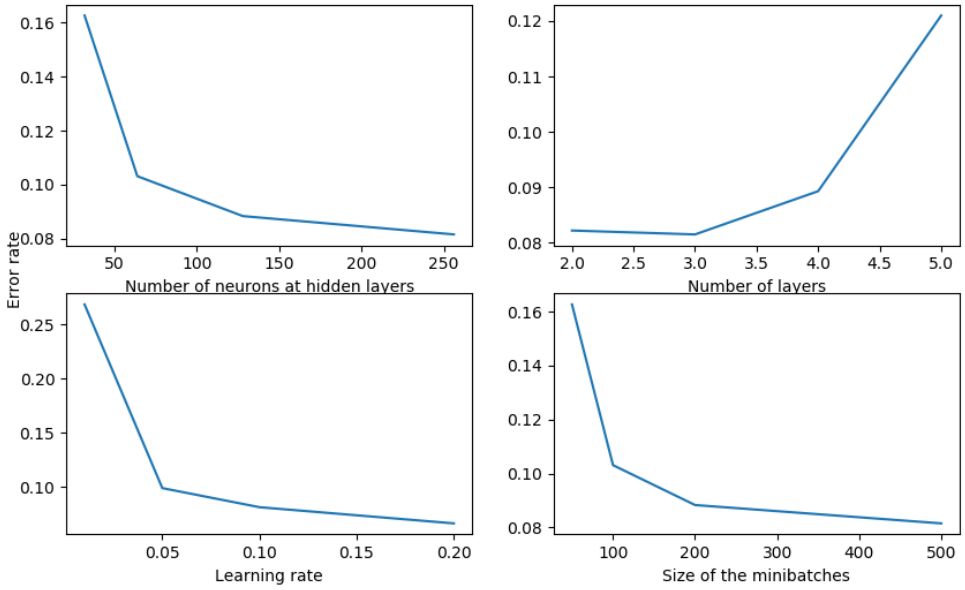


**Figure 1:** Error rate against different parameter settings

Setting $N_l, L, \eta, b$ at their optimal values, I then tried out different values of number of epoches, and here is the plot I obtained.

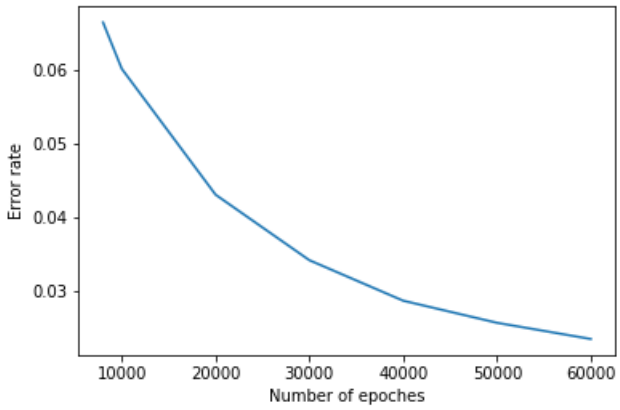


**Figure 2:** Error rate against different epoches

We can see that the error rate is decreasing substantially. And finally, setting the epoch number to be 60000, I can hit an error rate of 2.35%.

### 3. Getting predictions

My predicted labels are stored in the **TestDigitX.predict** and **TestDigitX2.predict**. And here are the plots of the average values for different predicted labels. We can see that the prediction probably worked well.

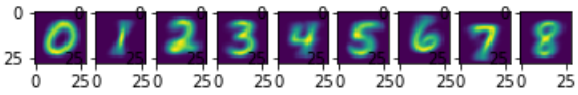

**Figure 3:** Average values for different predicted labels on TestDigitX

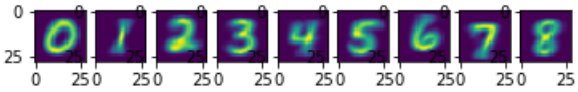

**Figure 4:** Average values for different predicted labels on TestDigitX2

## Some comments

1. Some experience I learned from tuning the parameters is that: (1) It is essential to feed the neural network model as many examples as possible. Increasing the number of epoches, and increasing the size of the minibatches do help to cut down the error rates. (2) For this simple classification task, we want to keep the neural network fat but short. We want the number of neurons at each hidden layers to be sufficiently large so that the information on all the 784 pixels are well captured, and the model should not contain too many layers, we might even harm its predicting power.

2. I first tried the cross entropy loss function with the softmax output, where the loss is determined by,

$$\epsilon = -\frac{1}{m}(y_0, y_1, ..., y_9) \cdot (log(p_0), log(p_1), ..., log(p_9))^\top$$

where for a sample with true label as $k$, $y_i = 0$ if $i \neq k$ and $y_i = 1$ if $i = k$. But I found the backward propagation generates many zero derivatives and performance is not improving clearly. This might be due to that the information on the incorrectly predicted probabilities (where $y_i = 0$ since $i \neq k$) are not fully taken advantage of.

3. The homework suggests we stop training when the error rate on the held out set increase substantially. I tried to kind of smoothed the error rate curve and stopped the training when the curve starts to pick up. However, the process constantly terminates too early, since the performance on the test set produces very high prediction error rates. Probably it takes effort to find the appropriate strategies to determine when the error rate has reached its global minimum.