EE5024 PA4: Feedforward Neural Network (MNIST)

Aakash (ME16B001)
Department of Mechanical Engineering
Indian Institute of Technology Tirupati
me16b001@iittp.ac.in

April 13, 2020

Aim

Train and test a Feedforward Neural Network for MNIST digit classification.

Procedure

1. Download MNIST file.zip which contains mnist data as a pickle file and read mnist.py for loading partial MNIST data.
2. Run read mnist.py file which will give 1000 train and 500 test images per each class.
3. x train, y train gives the image (784×1) and corresponding label for training data. Similarly, for test data.
4. Write:
   - Neural network model using library functions.
   - Your own neural network model and train with Back propagation.
     - On the training data and report accuracy.
     - Train with Five fold cross validation (4 fold training and 1 fold testing. Repeating this for 5 times changing the test fold each time) and report the average accuracy as train accuracy.
5. Test both models with the test data.
6. Find the confusion matrix and report the accuracy.

Working Theory

In this section we shall bring out the mathematical framework of the feed-forward neural network. This network shall use sigmoid activation for the hidden layers and softmax for the last layer with multi-class cross entropy loss (see later sections for details). Before proceeding any further we shall now define a few notations and terminologies.

- l indicates a layer in the network, here, $1 \leq l \leq N$ where N represents the numbers of layers in the network (Note: as per the convention input layer is not counted when we say N-layer neural network. Furthermore, for the network shown in Fig. 1 $N = 2$).
- Subscripts $k, j, i, \ldots$ usually denotes neuron indices in layers $l = N, N-1, N-2, \ldots$
- z_k^l represents the weighted sum of activations from the previous layer at layer l. That is,

$$z_k^l = \sum_j w_{kj} a_j^{l-1} + b_k \tag{1}$$

- a_k^l represents the neuron activations at layer l, $a_k^l = f(s_k^l)$, where $f(.)$ is the activation function. We shall be using softmax activation for the last layer ($l = N$)

$$a_k^N = \frac{e^{z_k^N}}{\sum_c e^{z_c^N}} \tag{2}$$

  Here, c is the number of classes. For all other layers ($l \neq N$) we shall use the sigmoid activation function.

$$a_k^l = \frac{1}{1 + e^{-z_k^l}} \tag{3}$$

- y_k is the ground truth (one-hot encoded) vector for the k^{th} sample.
- $\hat{y}_k$ is the predicted vector for the k^{th} sample.

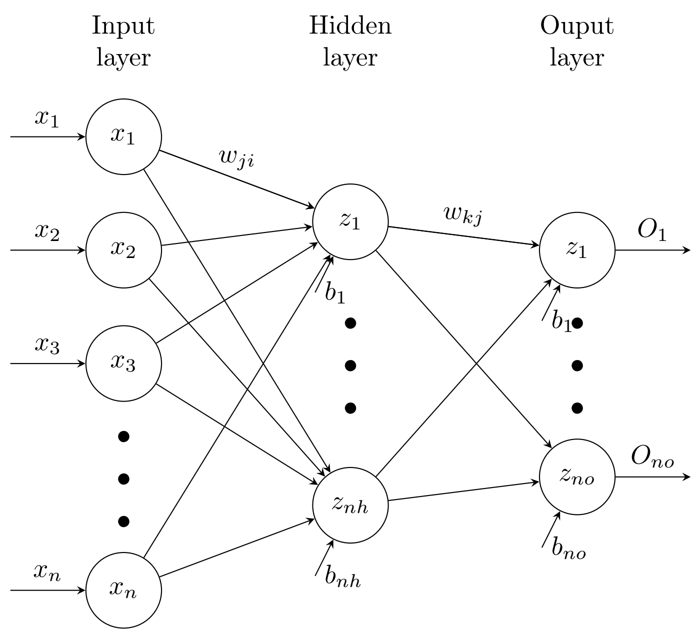

Figure 1: Feedforward Neural Network architechture

Forward pass

Now let us consider a Neural Network having two layers ($N = 2$) as shown in Fig. 1. Here, we have n neurons in the input layer (features), nh neurons in the hidden layer and no neurons in the output layer. As an aside, $1 \leq i \leq n$, $1 \leq j \leq nh$, and $1 \leq k \leq no$. We shall now lay out the equations for the forward pass through the network. For layer $l = N - 1 = 1$

$$z_j^l = \sum_i^n w_{ji} a_i^{l-1} + b_j \tag{4}$$

where, $a_i^{l-1} = a_i^0 = x_i^0$. Now we shall pass this weighted sum s_j^l through the activation function $f_1()$ i.e. sigmoid activation.

$$a_j^l = f_1(z_j^l) \tag{5}$$

This output of layer $l = 1$ is now fed to layer $l = 2$ i.e. the output layer. For layer $l = N = 2$

$$z_k^l = \sum_j^{nh} w_{kj} a_j^{l-1} + b_k \tag{6}$$

Similarly, we shall fed this weighted sum to another activation function $f_2()$ i.e. softmax activation.

$$a_k^l = f_2(z_k^l) \tag{7}$$

The final output vector a_k^l contains the probability for the k^{th} class. This class prediction might not make much sense as they take in weights and biases which are randomly initialized. Our prime goal here is to find the weights such that the predicted class probabilities are consistent with the ground truth labels in the training data. In order to achieve this we need to come up with a metric to measure the goodness (or badness) of the network, this can be done by constructing a loss function. The loss function can then be used to perform optimization by updating weights. We shall be using the multi-class cross-entropy loss in our approach. The loss for a given sample can be calculated using it's one hot encoded vector (y) and the prediction $\hat{y}$ (which is essentially a_k^l or a_k^2 for $l = 2$).

$$L(\hat{a}, a) = -\sum_{k=1}^{c} y_k \log \hat{y_k} \tag{8}$$

This can then be used to calculate loss across all samples (total number of samples: m) as

$$J(w^1, b^1, \dots) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^i, y^i) \tag{9}$$

One important result that we take directly without any derivation is the gradient of J with respect to $z_k^{l=2}$ (simplified as z^2).

$$\frac{\partial J}{\partial z^2} = \hat{y} - y \tag{10}$$

Backward pass

In this sub section we shall bring out the mechanism to update the weights and biases by backpropagting the loss into the network. We shall update the weights using a iterative approach, more precisely using the Stochastic Gradient Descent (SGD). The weights can be updated using

$$w_{kj}(t+1) = w_{kj}(t) - \eta \frac{\partial J}{\partial w_{kj}} \tag{11a}$$

$$w_{ji}(t+1) = w_{ji}(t) - \eta \frac{\partial J}{\partial w_{ji}} \tag{11b}$$

Here, η is a hyperparameter called learning rate. Similarly, the biases can also be updated.

$$b_k(t+1) = b_k(t) - \eta \frac{\partial J}{\partial b_k} \tag{12a}$$

$$b_j(t+1) = b_j(t) - \eta \frac{\partial J}{\partial b_j} \tag{12b}$$

Now, our goal is to find the gradients. This gradients can be obtained by backpropagating via the network usinf the chain-rule.

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial z^2}\frac{\partial z^2}{\partial w_{kj}} = (\hat{y} - y)z_k \tag{13a}$$

$$\frac{\partial J}{\partial b_k} = \frac{\partial J}{\partial b_k}\frac{\partial z^2}{\partial b_k} = (\hat{y} - y) \tag{13b}$$

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial z^2}\frac{\partial z^2}{\partial a_j}\frac{\partial a_j}{\partial z^1}\frac{\partial z^1}{\partial w_{ji}} = (\hat{y} - y)w_{kj}f_1(z^1)(1 - f_1(z^1))a_i \tag{14a}$$

$$\frac{\partial J}{\partial b_j} = \frac{\partial J}{\partial z^2}\frac{\partial z^2}{\partial a_j}\frac{\partial a_j}{\partial z^1}\frac{\partial z^1}{\partial b_j} = (\hat{y} - y)w_{kj}f_1(z^1)(1 - f_1(z^1)) \tag{14b}$$

Hence, by using the above set of equations we can run SGD and update the trainable parameters for this two layered network. The same idea can be extended to build multilayered networks.

Results

In this final section we present the results obtained for the network. First we implemented the neural network model using library functions, for which the results are presented in Table 1.

Items	**Value**
Hidden activation	Sigmoid
Output actiation	Softmax
# layers	2
# neurons	256
# batch size	128
learning rate	0.1
Test accuracy	82.26

Table 1: Results of implementation using library functions

The neural network was also implemented from scratch using python3 (see attached notebooks). The model was trained using different combinations of activation functions and # of neurons (all models have same learning rate of 0.1). The models were validated using five fold cross validation and the results are reported in Table 2. The validation accuracy is highest for the sigmoid activation using 265 neurons. For the sigmoid activation the validation accuracy increases with increase in # neurons, while for the ReLU and tanh activation it first increases and then goes stagnant. This can be attributed to the fact that we haven't applied early stopping in our training process as it is a well known fact that Early stopping is some form of L2 Regularization. We wanted to perform our analysis on same set of parameters. In practice, it is a good idea to use early stopping by choosing some threshold where the validation loss starts increasing by that threshold than the previous validation loss.

Activation **# Neurons**	**Sigmoid**	**ReLU**	**Tanh**
32	88.99	39.10	70.10
64	89.67	63.01	71.99
128	90.29	61.57	68.35
256	91.16	32.14	54.41

Table 2: Validation accuracy using different number of hidden neurons and activations

In Table 3, we observe almost the same trend as in validation. The highest test accuracy is obtained for Sigmoid activation function with 256 neurons. One important observation here is that performance of ReLU is not as expected. This is attributed to the fact that we required a higher learning rate of 0.1 in SGD for Sigmoid activation function to converge. For lower learning rates the SGD wasn't converging whereas performance of ReLU was improving significantly improved as expected. Therefore, to test over same parameters we finally chose learning rate to be 0.1.

Activation # Neurons	**Sigmoid**	**ReLU**	**Tanh**
32	89.44	39.12	71.06
64	89.98	64.16	74.40
128	90.32	61.82	69.78
256	91.62	31.68	55.06

Table 3: Test accuracy using different number of hidden neurons and activations

Training time for same set of parameters is shown in Table 4. It can be observed that tanh takes the largest training time whereas ReLU is the fastest. Also training time is increasing with # of neurons as expected.

Activation # Neurons	**Sigmoid**	**ReLU**	**Tanh**
32	133.137	116.511	145.912
64	170.921	152.531	205.702
128	255.141	207.650	298.367
256	371.729	288.749	448.806

Table 4: Training time (s) for different number of hidden neurons and activations