ECE 544NA Fall 2016 Assignment 2

Jifu Zhao Date: 09/30/2016

I. Pencil-and-Paper

1. Derivative of Softmax
Recall the softmax function:

$$\vec{y}[k] = \frac{e^{\vec{z}[k]}}{\sum_{l=1}^{C} e^{\vec{z}[l]}} \tag{1}$$

So, consider two cases, j = k and j != k.

When j = k:

$$\frac{\partial \vec{y}[k]}{\partial \vec{z}[j]} = -\frac{e^{\vec{z}[k]} \cdot \sum_{l=1}^{C} e^{\vec{z}[l]} - e^{\vec{z}[k]} \cdot e^{\vec{z}[k]}}{(\sum_{l=1}^{C} e^{\vec{z}[l]})^2} = \vec{y}[k] - \vec{y}[k] \cdot \vec{y}[k] \tag{2}$$

When j != k:

$$\frac{\partial \vec{y}[k]}{\partial \vec{z}[j]} = -\frac{e^{\vec{z}[k]} \cdot e^{\vec{z}[j]}}{(\sum_{l=1}^{C} e^{\vec{z}[l]})^2} = -\vec{y}[k] \cdot \vec{y}[j] \tag{3}$$

So, we can conclude that:

$$\frac{\partial \vec{y}[k]}{\partial \vec{z}[j]} = \begin{cases} -\vec{y}[k] \cdot \vec{y}[j] & \text{if } k! = j \\ \vec{y}[k] - \vec{y}[k] \cdot \vec{y}[k] & \text{if } k = j \end{cases} \tag{4}$$

2. Negative Log Likelihood loss for Multi-Class.
Recall the negative log likelihood:

$$L = -\sum_{i}^{N} \sum_{k}^{K} \mathbf{1}[y_i = k] \cdot log(\hat{\vec{y}}_i[k]) \tag{5}$$

In this way, we have:

$$\frac{\partial L}{\partial \hat{\vec{y}}_i[j]} = -\mathbf{1}[y_i = k] \cdot \frac{\partial log(\hat{\vec{y}}_i[j])}{\partial \hat{\vec{y}}_i[j]} = -\frac{\mathbf{1}[y_i = j]}{\hat{\vec{y}}_i[j]} \tag{6}$$

3. Avg-pooling (1D)
Recall Avg-pooling (1D) operation with window size W:

$$\vec{y}[i] = \frac{1}{W} \sum_{j=0}^{W} \vec{x}[i + j] \tag{7}$$

Then, we have:

$$\frac{\partial \vec{y}[i]}{\partial \vec{x}[j]} = \begin{cases} \frac{1}{W} & \text{if } i \le j \le i + W \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

4. Max-pooling (1D)
Recall Max-pooling (1D) operation with window size W:

$$\vec{y}[i] = \max_{j=0}^{W} \vec{x}[i+j] \tag{9}$$

Then, we have:

$$\frac{\partial \vec{y}[i]}{\partial \vec{x}[j]} = \begin{cases} 1 & \text{if } \max_{k=0}^{W} \vec{x}[i+k] = \vec{x}[j] \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

5. Convolutional layer (1D)
Recall Convolution (1D) operation, assume $\vec{w}$ is length 3, and zero index at the center:

$$\vec{y}[i] = (\vec{w} * \vec{x})[i] = \sum_{j=-1}^{1} \vec{x}[i-j] \cdot \vec{w}[j] \tag{11}$$

From above equation, we have:

$$\frac{\partial \vec{y}[i]}{\partial \vec{x}[j]} = \begin{cases} \vec{w}[i-j] & \text{if } i-1 \leq j \leq i+1 \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

$$\frac{\partial \vec{y}[i]}{\partial \vec{w}[j]} = \begin{cases} \vec{x}[i-j] & \text{if } j = -1 \textit{ or } 0 \textit{ or } 1 \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

II. Code-from-Scratch

1. Methods

(1). The overall structure of the main part (Neural Network) is shown in Figure 1. And some helper function can be found in the file named helper.py.

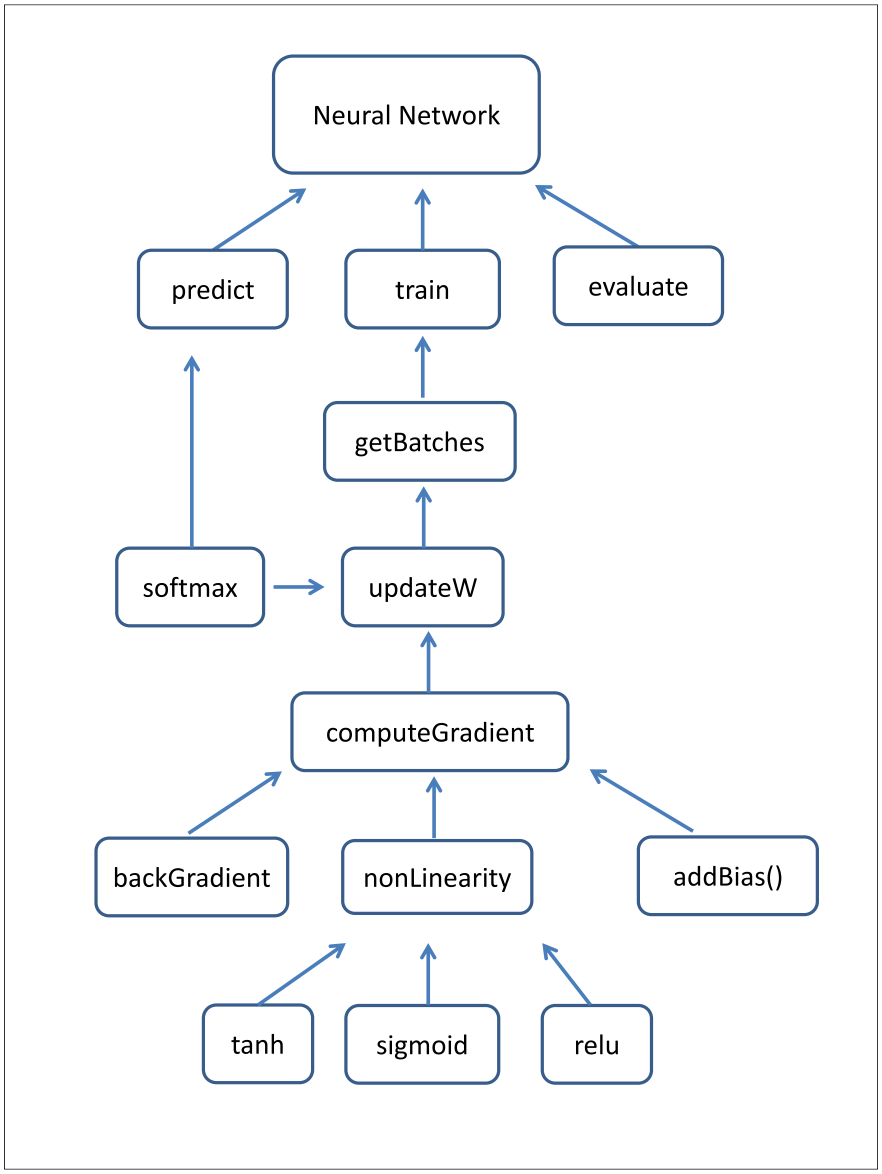


Figure 1: Algorithm Structure

(2). The whole program has train, predict, evaluate, getBatches, updateW and computeGradient. These functions are based above each other.

On the hidden nodes, you can choose the activation functions such as Sigmoid, Tanh and ReLU functions. The final output layer is calculated using Softmax function. On the back-propagation part, we choose negative log likelihood loss and update the weight one by one.

Due to the large parameters and many training set, we choose mini-batch gradient descent for optimization. On each global iteration, the training set is randomly shuffled and then divided into different batches according predefined batch size. In this work, we studied the influence of different batch sizes, the result will be discussed in next section.

The whole architecture of this part is a fully connected feed-forward multi-layer neural network. Besides the input layer, there are two hidden layers which have same hidden nodes, and the final output layer has 9 nodes corresponding to 9 different labels.

In the part, since we set the bias term to be 1 for all models and treat bias as an additional dimension, learning rate is one important hyper-parameter. During the experiment, we find that, the learning rate cannot be too large and will be different for different models and different hidden nodes. For example, the learning rate for ReLU activation function is almost always smaller than the learning rate for Sigmoid and Tanh activation function. Specifically, for this work, we choose around 0.6 for ReLU function, 0.3 for Sigmoid function and 0.3 for Tanh function.

(3). The total number of weights is different for different hidden nodes. Here, we show two numbers for the case that there are 10 hidden nodes and 50 hidden nodes.

When there are 10 hidden nodes:

$$\#\ weights = (1120 + 1) \times 10 + (10 + 1) \times 10 + (10 + 1) \times 9 = 11419$$

When there are 50 hidden nodes:

$$\#\ weights = (1120 + 1) \times 50 + (50 + 1) \times 50 + (50 + 1) \times 9 = 59059$$

Note: here we set bias to be 1 all the time and treat bias as an additional dimension.

2. Results

(1). In this section, we change the hidden nodes from 10 to 50, and choose batch size to be 50, through choosing different activation functions, we find that:

For Sigmoid function, with 50 hidden nodes, we can get the highest testing accuracy.

In this case, the training accuracy is 83.67% and the testing accuracy is 46.31%.

Note: here the highest testing accuracy is chosen based the cross validation, in other words, the development set accuracy. The testing accuracy is calculated on the evaluation set. With enough iterations, the training accuracy can always reach 95% or higher.

(2). In Table 1 and Table 2, we have listed the training accuracy and testing accuracy for all combinations between nonlinearities and number of hidden nodes.

Table 1: Comparison of training accuracy

Activation Function	ReLU	Sigmoid	Tanh
(10, 10) hidden nodes	66.42%	57.25%	63.16%
(20, 20) hidden nodes	87.94%	76.72%	59.21%
(30, 30) hidden nodes	60.71%	78.08%	75.35%
(40, 40) hidden nodes	56.84%	83.69%	77.91%
(50, 50) hidden nodes	69.90%	83.67%	79.72%

Table 2: Comparison of test accuracy

Activation Function	ReLU	Sigmoid	Tanh
(10, 10) hidden nodes	39.38%	43.01%	41.03%
(20, 20) hidden nodes	43.34%	46.09%	44.44%
(30, 30) hidden nodes	43.23%	44.44%	44.11%
(40, 40) hidden nodes	35.42%	45.10%	46.09%
(50, 50) hidden nodes	40.70%	46.31%	43.01%

(3). During the experiment, we also noticed that the running time for each inner iteration varies with batch size and number of hidden nodes, as well as activation functions. In Table 3, we have listed the averaged running for different activation functions and different number of hidden nodes.

Table 3: Average time for one iteration (in seconds)

Activation Function	ReLU	Sigmoid	Tanh
(10, 10) hidden nodes	0.00067	0.00309	0.00509
(20, 20) hidden nodes	0.00073	0.00330	0.00535
(30, 30) hidden nodes	0.00082	0.00352	0.00555
(40, 40) hidden nodes	0.00092	0.00366	0.00582
(50, 50) hidden nodes	0.00105	0.00383	0.00614

It is easy to notice that, ReLU function takes less time than Sigmoid and Tahn function. And with more hidden nodes, the running time becomes larger.

In addition, we also studied the influence of batch size on running time. Through changing the batch size in 50, 100, 150, 200, 250 and 300, we find that with larger batch size, the running time for one iteration becomes larger.

(4). In this section, using the best model we have got (as mentioned above), we can calculate the confusion matrix for training and testing set. The training classification confusion matrix and testing classification confusion matrix are shown in Figure 2 and Figure 3.

Note: the function to plot the confusion matrix is modified based on some online example.

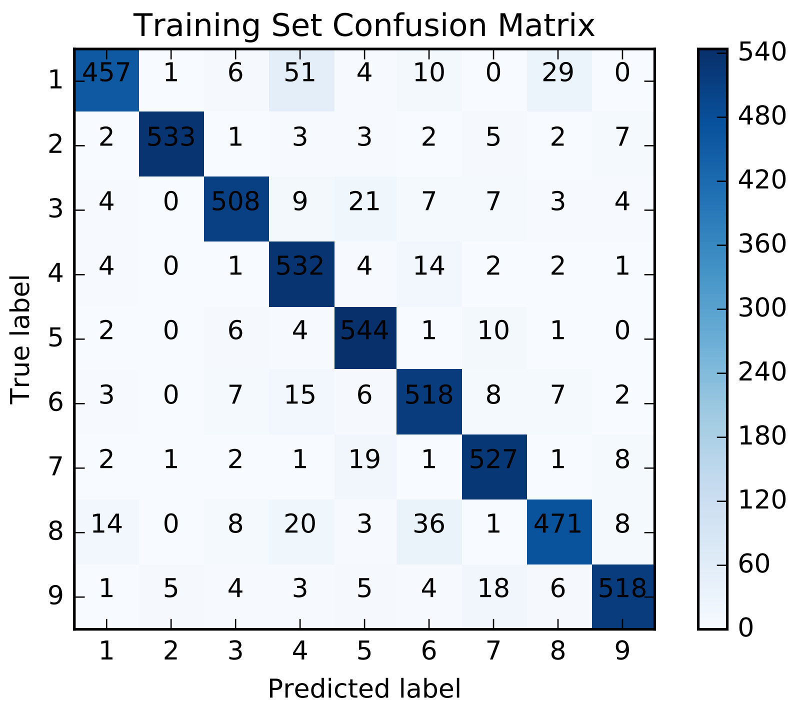


Figure 2: Training Set Confusion Matrix

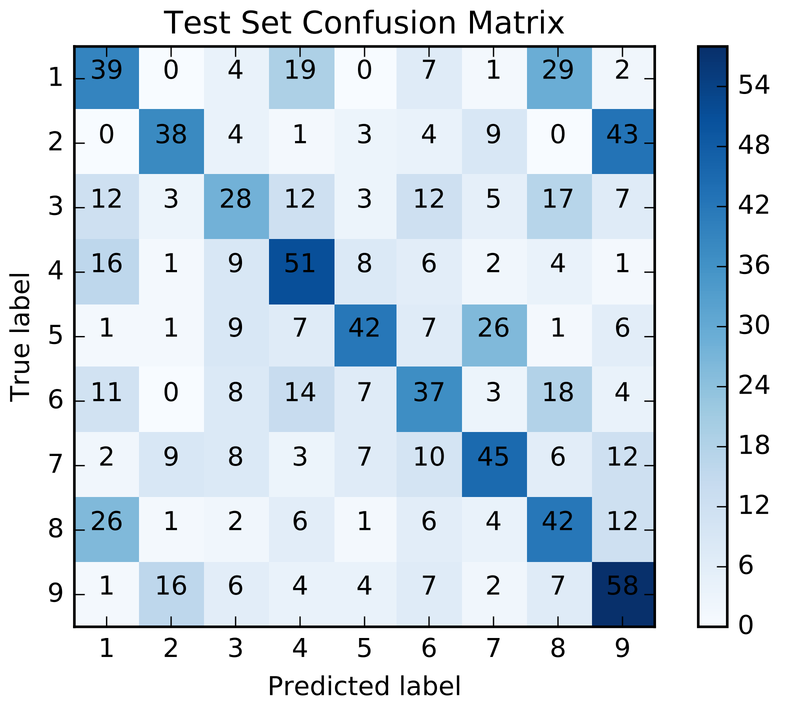


Figure 3: Testing Set Confusion Matrix

III. TensorFlow

1. Methods

(1). In this section, we use TensorFlow to build a TDNN model following the architecture proposed by Waibel, Hanazawa, Hinton, Shikano and Lang to perform 9 class classification.

At first, we followed the procedure in the literature, choosing the following network architecture:
Input Layer: (J=16, N=2)
Hidden Layer 1: (J=8, N=4)
Hidden Layer 2: (J=3)
Output Layer: fully connected feed-forward neural network with 9 output.

More specifically, we choose ReLU function as the activation function. For the final layer, we first flatten the output of hidden layer 2 to be one column vector and train a fully connected neural network to classify 9 classes. And final output is calculated through Softmax function and the loss function is negative log likelihood loss and train the model through ADAM optimizer. The learning rate is chosen to be 0.002 and the batch size is setting to be 50.

The used TensorFlow function includes: tf.reshape(), tf.nn.relu(), tf.nn.softmax(), tf.nn.conv2d() and so on. The final training accuracy is 55.91% and final testing accuracy is 45.76%.

Note: Here, one important thing is that, in each iteration, we don't train the model on all the batches. Instead, on each iteration, we first randomly shuffle the training set and split them into different subsets according to given batch sizes. Then, we select the first 5 batches to perform mini-batch training rather than using all batches to train the model.

(2). Besides the above model, we also tried different models, for example, in the above model, the output of the second hidden layer is an image with dimension (64, 3). However, since we have 9 classed, one direct idea is the change the output from (64, 3) to (64, 9), and then perform fully connected feed-forward neural network. Our result shows that, this change can improve the testing accuracy from 45.76% to 52.25%. Besides, the second hidden layer, we also tried to use different filters, such as more delay in input layer and/or less delay in first hidden layer. Some changes have clear influence, but some changes have little influence.

In addition, we also noticed that in the original paper, the author setting the final output to be the Sigmoid function of the sum of the output of hidden layer 2. Here, we also edit our model so that the final output is not a fully connected feed-forward neural network. Instead, since the output of second hidden layer has dimension (64, 9). We use the mean value of each column as the input to final output layer and use Softmax function to perform 9 class prediction. In this model, we noticed that our model has testing accuracy of 57.32%.

Note: During the training process, we also noticed that some of our model is sensitive to the initialization. The final output accuracy may varies a lot. The reported accuracy mentioned above is the highest accuracy we ever get. (All testing accuracy is determined through development set, rather than decided directly.)

(3). In the original TDNN model we set:
Input Layer: (J=16, N=2)
Hidden Layer 1: (J=8, N=4)

Hidden Layer 2: (J=3)
Output Layer: fully connected feed-forward neural network with 9 output.
So, we can calculate the total number of weights as follows:

$$\#\ weights = 16 \times 3 \times 8 + 8 \times 5 \times 3 + 64 \times 3 \times 9 = 2232$$

Note: Here, we didn't consider bias as the additional weights, if bias is considered, then:

$$\#\ weights = 16 \times 3 \times 8 + 8 + 8 \times 5 \times 3 + 3 + 64 \times 3 \times 9 + 9 = 2252$$

(4). For the original TDNN model, at the input layer, the activation dimensions are:

$$16 \times 70 = 1120$$

At the first hidden layer, the activation dimensions are:

$$8 \times 68 = 544$$

At the second hidden layer, the activation dimensions are:

$$3 \times 64 = 192$$

At the final output layer, since we use the fully connected neural network, the activation dimensions are 9.

(5). The used TensorFlow function includes: tf.reshape(), tf.nn.relu(), tf.nn.softmax(), tf.nn.conv2d() and so on. Specifically, for tf.nn.conv2d(), we choose "VALID" mode, as shown in line 54 of TensorFlow.py. In addition, we initialize the weight variable through tf.truncated_normal() and initialize the b variable to be constant 0.1.

The overall structure of the code is simple, in file named TenforFlow.py, there are generally 4 parts. In the first part, we load the data and define some helper functions. Then, in the second part, we apply the model as required by the course website, where:
Input Layer: (J=16, N=2)
Hidden Layer 1: (J=8, N=4)
Hidden Layer 2: (J=3)
Output Layer: fully connected feed-forward neural network with 9 output.

On the third part, we used a flexible model, where you can change the filter size, learning rate and so on. On the last part, we used a similar model similar to part 2, but on the final output layer, we don't use a fully connected feed-forward neural network from hidden layer 2 to output layer, instead, we use the mean value of each dimension from hidden layer 2 as the input to final output layer, and then perform Softmax() function for 9 class classification. This is similar to the original model described in the paper.

On each part of part 2, 3 and 4, the overall structure of the code is similar. First we define the hyper-parameters such as learning rate. Then, we create the tensorFlow session and create some placeholders to hold the variables. Then through define the weight filter and bias term for each layer, we create all the variables. Next, we define the loss and accuracy. Through selecting appropriate batches, we start training the model and use the trained model to calculate the training accuracy and development accuracy. Finally, we find the best model through development set and apply the model on the evaluation set and plot the convergence curve and confusion matrix.

2. Results

(1). In this part, following the procedure described above, we calculate the training and testing accuracy for TDNN model.

Table 4: Training and testing accuracy for TDNN model

Used Set	Accuracy
Training Set	55.91%
Testing Set	45.76%

(2). In this part, following the procedure described above, we calculate the training and testing accuracy for best TDNN model we have got.

Table 5: Training and testing accuracy for best model

Used Set	Accuracy
Training Set	67.64%
Testing Set	57.32%

(3). In this part, the training and testing classification confusion matrix for the TDNN model are shown in Figure 4 and Figure 5.

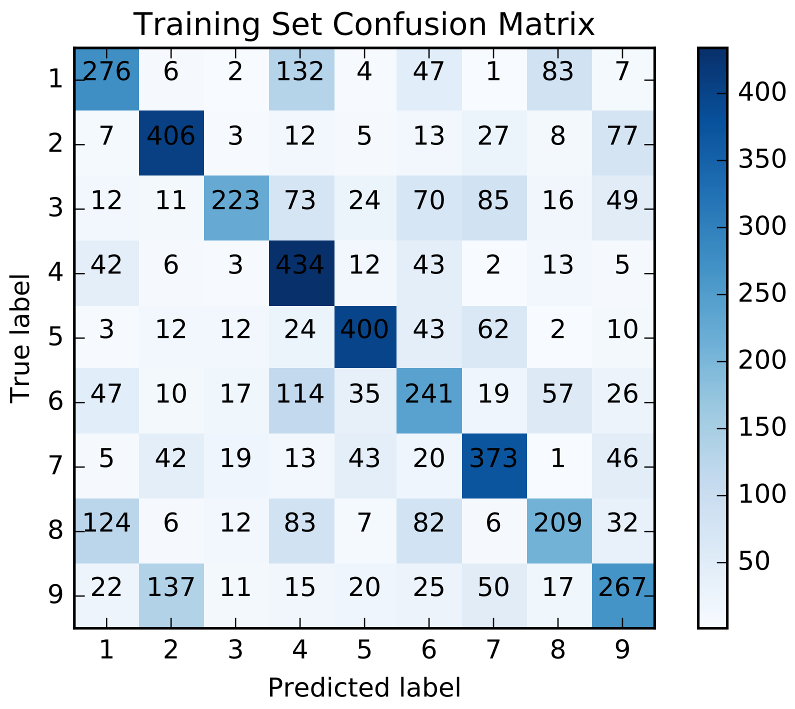
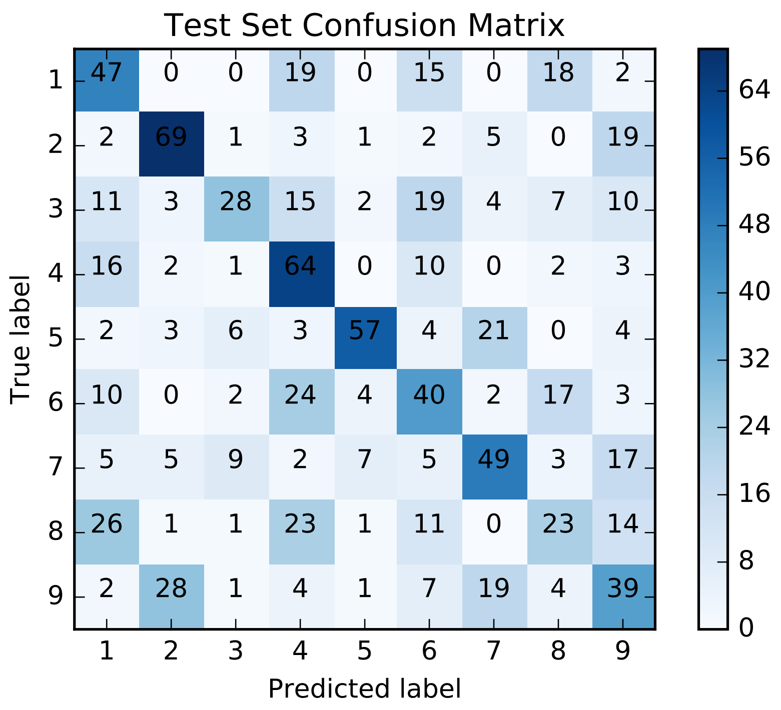


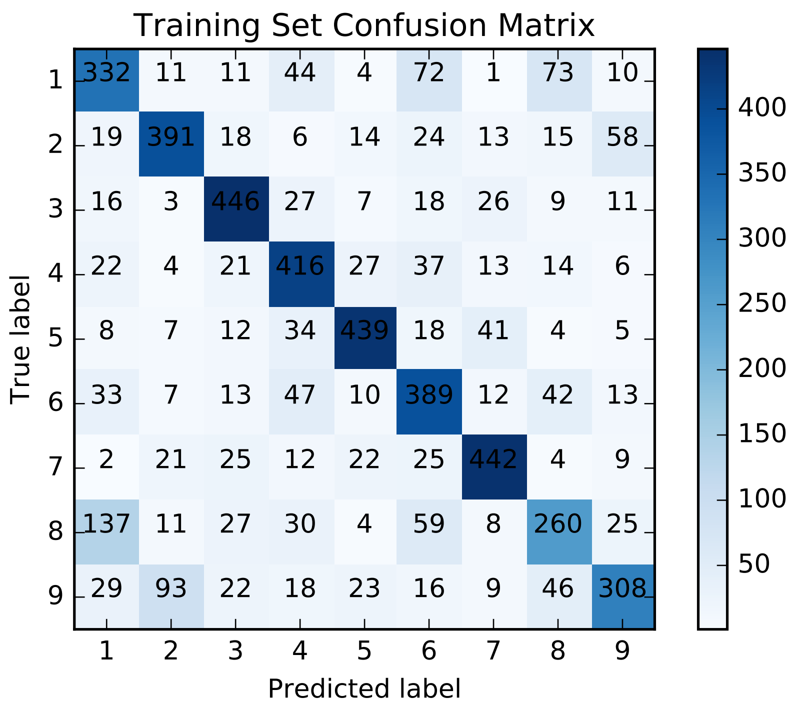
Figure 4: Training Set Confusion Matrix for TDNN model



True label \ Predicted label	1	2	3	4	5	6	7	8	9
1	47	0	0	19	0	15	0	18	2
2	2	69	1	3	1	2	5	0	19
3	11	3	28	15	2	19	4	7	10
4	16	2	1	64	0	10	0	2	3
5	2	3	6	3	57	4	21	0	4
6	10	0	2	24	4	40	2	17	3
7	5	5	9	2	7	5	49	3	17
8	26	1	1	23	1	11	0	23	14
9	2	28	1	4	1	7	19	4	39

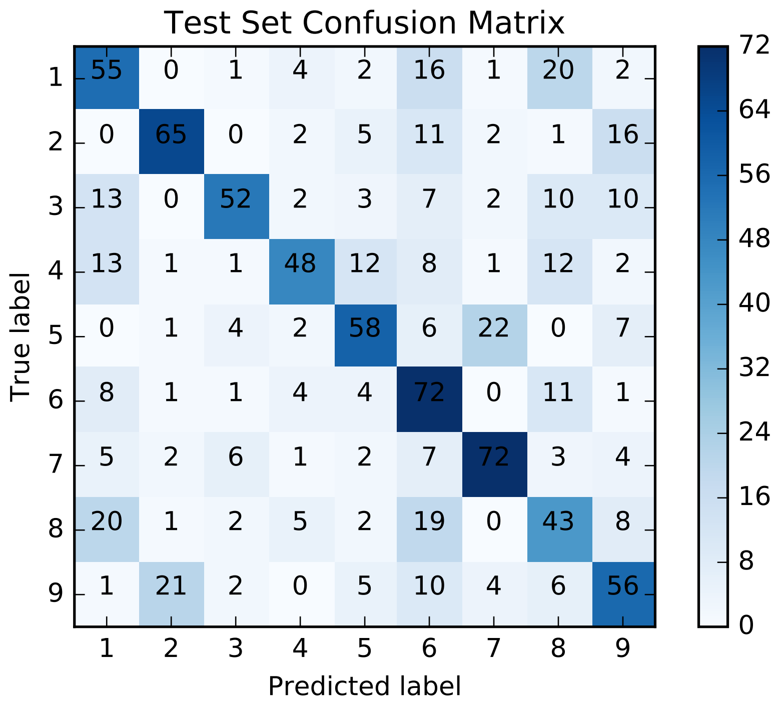
Figure 5: Testing Set Confusion Matrix for TDNN model

The training and testing classification confusion matrix for the best TDNN model are shown in Figure 6 and Figure 7.



True \ Predicted	1	2	3	4	5	6	7	8	9
1	332	11	11	44	4	72	1	73	10
2	19	391	18	6	14	24	13	15	58
3	16	3	446	27	7	18	26	9	11
4	22	4	21	416	27	37	13	14	6
5	8	7	12	34	439	18	41	4	5
6	33	7	13	47	10	389	12	42	13
7	2	21	25	12	22	25	442	4	9
8	137	11	27	30	4	59	8	260	25
9	29	93	22	18	23	16	9	46	308

Figure 6: Training Set Confusion Matrix for best model



True \ Predicted	1	2	3	4	5	6	7	8	9
1	55	0	1	4	2	16	1	20	2
2	0	65	0	2	5	11	2	1	16
3	13	0	52	2	3	7	2	10	10
4	13	1	1	48	12	8	1	12	2
5	0	1	4	2	58	6	22	0	7
6	8	1	1	4	4	72	0	11	1
7	5	2	6	1	2	7	72	3	4
8	20	1	2	5	2	19	0	43	8
9	1	21	2	0	5	10	4	6	56

Figure 7: Test Set Confusion Matrix for best model