CSCE 479/879 (Fall 2019) Homework 1

J. Brennan Peace, Suresh Thapa and Praval Sharma

September 29, 2019

1 Introduction

This work deals with the classification of Fashion MNIST dataset using artificial neural network in Python TensorFlow. The dataset contains 60,000 images for training and validation and 10,000 images for testing. Each image is a 28x28 grayscale image which is flattened into an array of 784 pixels. All the images belong to one of the 10 classes as listed below:

- 0 T-shirt/top
- 1 Trouser
- 2 Pullover
- 3 Dress
- 4 Coat
- 5 Sandal
- 6 Shirt
- 7 Sneaker
- 8 Bag
- 9 Ankle boot

In this experiment, we have created two different architectures and compared them by evaluating their performance on the test dataset. We have also tried to find the optimal set of hyperparameters by running both the architectures several times with different combination of hyperparameters.

2 Methodology

2.1 Preprocessing

The Fashion Mnist dataset contains images with integer values in the range of 0 through 255 which has been normalized using the min-max normalization technique. The formula for the normalization is as follows:
$x_{norm} = (x - x_{min})/(x_{max} - x_{min})$
The data labels are one hot encoded so that the deep learning architecture can generate a probability distribution for all ten categories. Furthermore, the dataset given to us is only the train dataset (only 60,000 images) which is further partitioned into train and test set in the ratio of 4:1 such that there are 48,000 images for training and 12,000 images for testing the model.

2.2 Model Architecture

Neural network comprises an input layer, intermediate hidden layers and an output layer. Each layer contains a certain number of nodes/neurons which are connected to every neurons in the subsequent layer as shown in Figure 1.

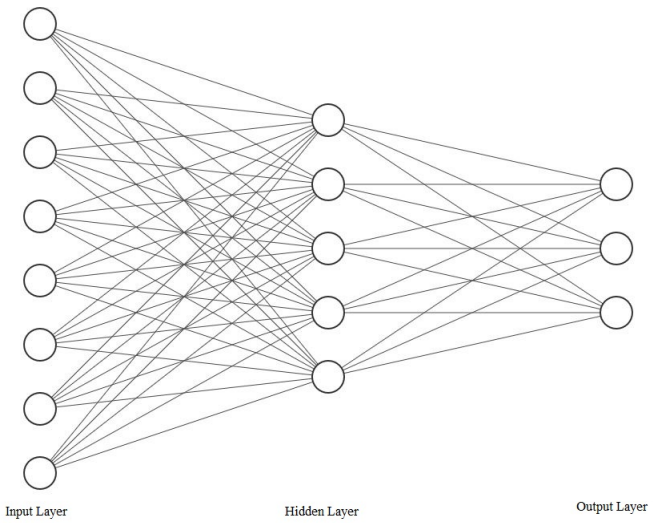

Figure 1: General architecture of neural network

In this paper, two different architectures of neural networks have been implemented. The first architecture is relatively simple and acts as a baseline model, so that the other advanced architectures can be compared with it. It consists of an input layer, first hidden layer with 300 nodes, second hidden layer with 200 nodes and an output layer with 10 nodes representing the values for the 10 corresponding classes. Exponential linear unit (ELU) is used as the activation function and Adam optimizer is used for model optimization. The second architecture is more sophisticated than the first one as it is much deeper and contains larger number of parameters. It contains 8 hidden layers where each hidden layer consists of 200 nodes. Initially, all the hyper-parameters are kept constant as in the first architecture.

First, both architectures are tested in their vanilla form. There is no any regularization except the early stopping technique. At every epoch, the current loss of the model is compared with the best loss.If the current loss is smaller or equal to the best loss, the best loss value is updated with the current loss. The model is also saved. On the other hand if the current loss is higher than the best loss, a value called patience parameter is increased by 1. If the patience parameter > 15, the training stops due to over-fitting. The pseudo code for early stopping is as follows:

Best_loss = 1000

Patience_parameter = 0

if train_loss $\leq$ Best_loss:

    Best_loss = train_loss

    Save the Model

    Patience_parameter = 0

else

    Patience_parameter = Patience_parameter + 1

    if Patience_parameter $>$ 15

    Stop Training.

After that, both architectures are run with some regularization. This is done by adding dropout layers after each hidden layer and also assigning a kernel and bias regularizer. Upto this point, no any other hyper-parameters have been tuned. Finally, all these training steps are repeated with different set of hyperparameters.

2.3 Evaluation metrics

The models will be evaluated using accuracy and confusion matrix.

3 Results and Discussion

The training and testing result from all of these different variations in the architecture and hyperparameters are tabulated in Table 1.

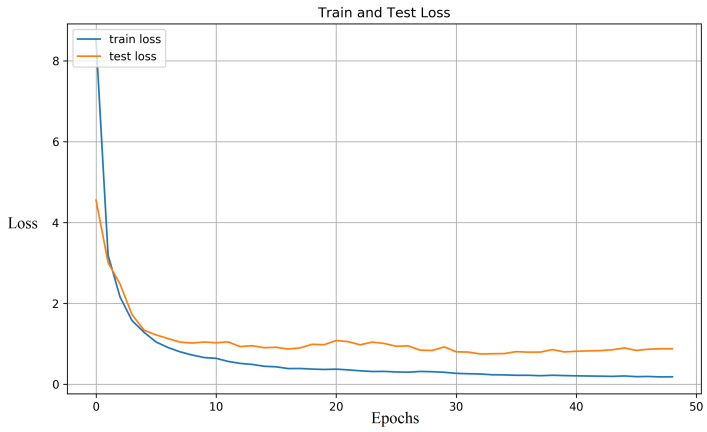

Figure 2: Training and testing loss for Run 1

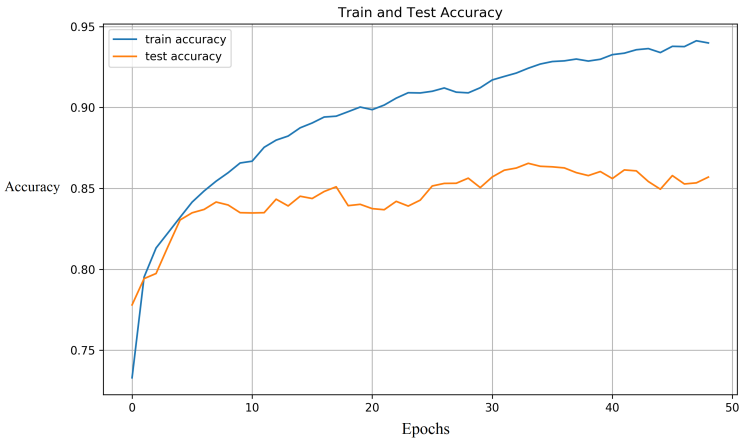

Figure 3: Training and testing accuracy for Run 1

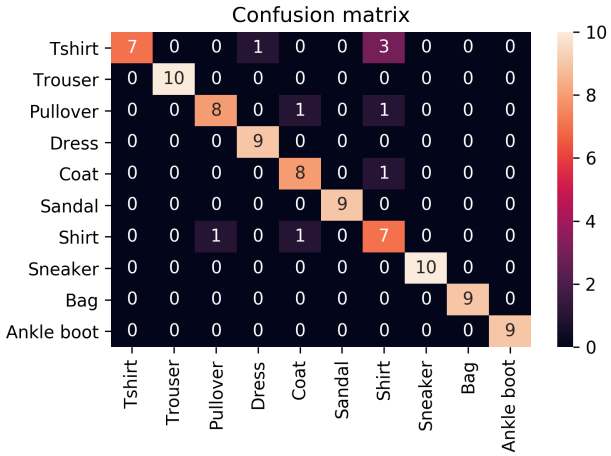

Figure 4: Confusion matrix for Run 1

Table 1: Result

Run	Architecture	Regularization	Activation	Learning Rate	Train Acc	Test Acc	Train Loss	Test Loss	Upper Bound	Lower Bound	Epoch
1	First	Leaky_relu	None	0.0001	93.98	85.7	0.1842	0.8783	14.93	13.67	49
2	First	Leaky_relu	L2(0.1)	0.0001	95.925	86.93	0.1109	0.4948865	13.67	12.46	107
3	Second	Leaky_relu	None	0.0001	88.67	86.81	0.3137	0.5671	13.79	12.59	20
4	Second	Leaky_relu	L2(0.1)	0.0001	90.52	87.28	0.2638	0.3609	13.31	12.12	40
5	First	elu	None	0.00005	96.06	85.27	0.1129	0.9059	15.36	14.01	15
6	First	elu	L2(0.05)	0.00005	98.63	87.07	0.0392	0.6744	13.53	12.33	130
7	Second	elu	None	0.00005	90.79	86.90	0.2544	0.5744	13.69	12.49	22
8	Second	elu	L2(0.05)	0.00005	90.83	88.21	0.2499	0.3887	12.37	11.21	32

Figures 2 and 3 show the loss and accuracy curve for the vanilla model(Run 1). The training accuracy for the Run was 93.98 % while the testing accuracy was only 85.7%. From the accuracy curve, it can be seen that the training accuracy continuously increases as the epochs increase. However, the test accuracy has small oscillations over a mean value and does not increase significantly over time. The model starts to overfit around 25-30 epoch and then finally the training stops early at epoch 49. This can be verified from Figure 2, where the test loss gradually starts increasing while the training loss is decreasing rapidly. Figure 4 shows the normalized confusion matrix for Run 1. The high values in the diagonal represent the correct predictions of the model. The numbers in the diagonal add up to 86 which is equivalent to the accuracy of the model. However, the numbers outside of the main diagonal add only up to 9 instead of 14, which is due the rounding error while calculating normalized confusion matrix.Also, the matrix shows that the model is having difficulty in identifying the tshirt. The model misclassifies shirt as a shirt or a dress. This makes sense as sometimes, with such low resolution images, a human might actually be also confused between them. Similar misclassifications happen for Pullover(row 3) and Shirt(row 7).

In Run 2, we regularized the model in Run 1 adding L2 regularization for kernel and bias, and as well as adding a drop out layer will drop probability 0.5. The overall coefficient for L2 regularizaiton was 0.1. Theoretically, this should help to minimize overfitting and improve generalization accuracy. As seen from table, although insignificant, there is slight improvement of the accuracy.

In Run 3, we trained the second architecture without any regularization. This architecture is significantly deeper than the one used in Run 1 and contains much more parameters. Hence, this model is also expected to overfit the data very quickly. As shown from the table, the results are similar to Run 1 but the model stops much earlier at just epoch 20 due to overfitting.

In Run 4, the second architecture is trained with L2 regularization and dropout layer, with same regularization parameters as in Run 3. Regularization in combination with deeper architecture seem to have slightly reduced the problem of over fitting and improved the accuracy.

Run 5 to Run 8 are exactly similar to Run 1 through Run 4, except the hyperparameters used in them. We played with 4 hyperparameters - activation function, batch size, learning rate and regularization coefficient. We changed one parameter at a time and saw their effect on accuracy, and kept the best one. For activation fucntions, elu was the better than relu and leaky relu. Learning rate of 0.00005 was found optimum. A learning rate higher than 0.0001 was a bigger step for gradient descent and did not lead to minimum error. Learning rate much smaller than 0.00005 would slow down the training process drastically and thus was not chosen. Also, previous regularization coefficient 0.1 was found to be harsh, and 0.05 was a better one. Finally, batch size of 64 gave better accuracy than 32,128 or 256.

Run 8 gave the best performance. Fig 5 and Fig 6 show the test loss and test accuracy for the best case. The best test accuracy is 88.21 %. Confusion matrix shown in Fig. 7 also verifies that the model has learnt better. At least now, the model is much less confused in identifying Tshirt over other similar apparels.

Overall, the deeper network performed slightly better than the shallower network. However, it comes at the expense of too many parameters which slows down the training process and is prone to overfitting. Thus, depending upon the application and resources available, a simpler model with slightly less accuracy may be more suitable. The model might learn better provided more data. At the end, we could use all 60,000 images for training with the optimal parameters obtained. Or, we could also use a smaller test size(9:1 split), so that the model gets larger data for the learning task.

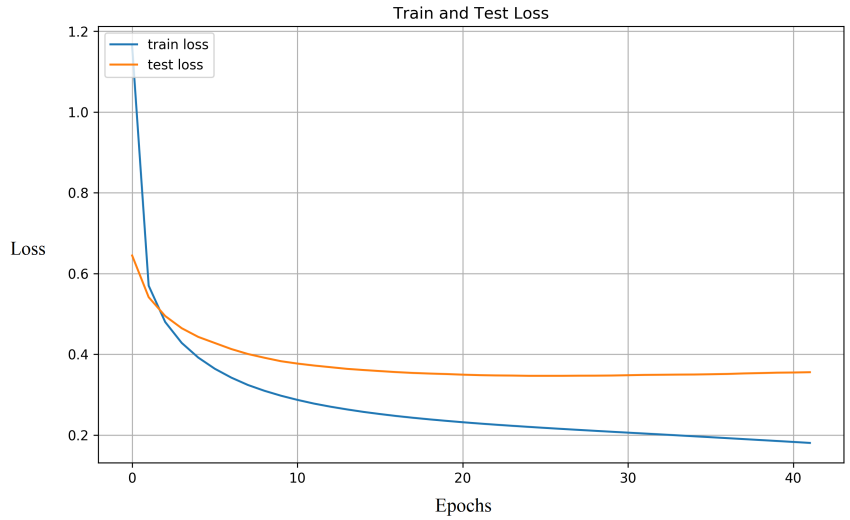

Figure 5: Training and testing loss for Run 8 (Best case)

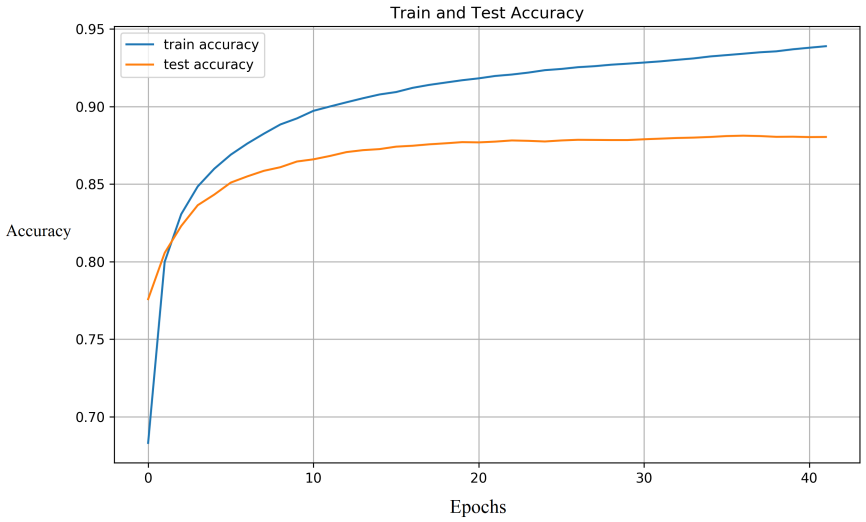

Figure 6: Training and testing accuracy for Run 8 (Best case)

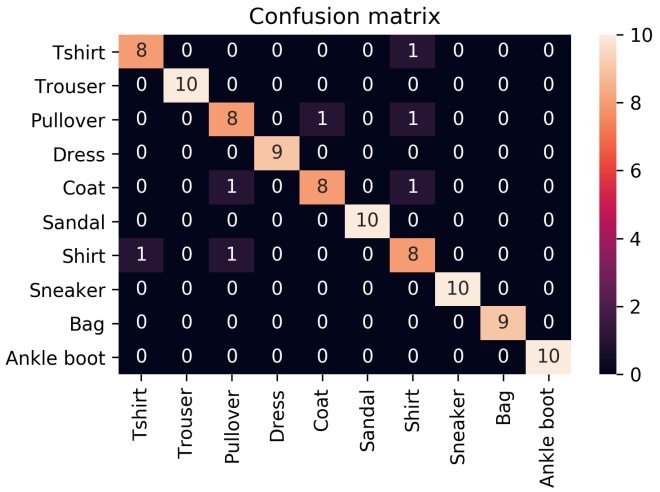

Figure 7: Confusion matrix for Run 8 (Best case)

4 Conclusion

Two different neural network architectures were compared in this work for classifying Fashion MNIST dataset. Between them, one was a shallow network while another was relatively deeper one. The performance of the model is highly dependent on its architecture and the hyperparameters. The deeper network which consisted of 10 hidden layers gave the best performance when L2 regularization was applied. The accuracy of the best model is of 88.21 %.

Instructions to run the code

To run the code, run main.py from the command line. The different architectures implemented can be chosen by giving argument in the command line. For example,

$ python main.py architecture_1

The above command runs the code with architecture_1 defined in model.py file. The four architecture that can be chosen are:

- architecture_1
- architecture_1_with_regularization
- architecture_2
- architecture_2_with_regularization