CSA 250 Deep Learning

Assignment 2

Python Dependencies and Requirements

- python (v3.7)
- tensorflow (v2.1)
- Other libraries used: argparse, matplotlib, numpy, os

Instructions to run

The 'main.py' is the interface to the program. It is programmed to run in two modes – train mode and test mode. The 'main.py', when executed without any arguments, enters into testing the deep networks and produces the output files 'multi-layer-net.txt' and 'convolution-neural-net.txt'. The 'main.py' when executed with the (optional argument) '--train-model' enters into training mode and saves the model after training.

- Train mode : python main.py -train-model
- Test mode : python main.py

Program details

This program is an implementation of the Fashion-MNIST clothing images classification using multilayer neural network/perceptron (MLP) model and using convolutional neural network (CNN) model. The Fashion-MNIST is a dataset of Zalando's article images, consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255. The dataset can be simply loaded using tensorflow API. The program has four major components:

- 'main.py' : This python file acts as a client file for the user to interact
- analytics : This is a python package consisting of python files for analysis and plotting purposes
- CNN : This is a python package consisting of python files for implementation and testing using CNN network.
- MLNN : This is a python package consisting of python files for implementation and testing using MLNN network.

What made you choose your current architecture? How many different architectures have you considered?

The deep network model – CNN and MLNN – need the architecture to be defined appropriately to perform the task efficiently. This involves choosing the number of layers (apart from input and output layers), their hidden units (for MLNN), kernel filters and sizes (for CNN) etc. The network architecture should be deep enough to extract the key essential features from the input, which would help in properly learning the characteristics for each class. At the same time, the network should not be too deep so that the gradients diminish moving towards the input layer, which would result in improper or no learning. The actual rule of thumb to select the number of hidden layers depends on the complexity and nature of the task that we are expecting the network to do.

In this project, we are trying to classify clothing based on the FashionMNIST dataset, which is an image based classification. So our CNN and MLNN should be deep enough to extract distinguishing features such as edges, shape, contour etc., but should not be too deep that it does not learn or it overfits. I had tried numerous architecture, with varying hyperparameters, from very shallow to very deep network trying to find an optimal model. But there was not any key improvement over the accuracy using deep models – which might be due to the nature of the problem. The largest contributor to the fall in accuracy was the class corresponding to shirt which gets classified to its extremely similar counterparts like Tshirt, Pullover and Coat. Unless we are able to specifically learn the charecteristics of these classes, there might not be any significant improvement in the performance of the model. The data augmentation helps in training a model that generalises better – but in this case, there was only a slight change.

How did you arrive at your choice of hyperparameters - number of neurons/layers, activation function, learning rate etc.?

As mentioned in answer to the previous question, the number of layers should be large enough to extract the key essential features from the input but should not be too large so that the gradients diminish moving towards the input layer. The number of layers in both CNN and MLNN was chosen after numerous trials and experiments. There is always a trade off between the model complexity (depth) which essentially decides the execution time and the accuracy. There were few papers and models (see #) that results in superlative performance compared to my model, but they come with the additional cost of more compute time and power. The activations functions used in the model were ‘LeakyRelu’ for hidden layers and ‘Softmax’ for the output layer, which are the standard ones used today. The optimiser used is RMSprop with an initial learning rate alongside ReduceLROnPlateau() – which effectively reduces the learning rate by a factor when the specified metric (here, validation accuracy) plateaus for a specified number of iterations. The number of training epochs is chosen after letting the model train for a large number of epochs and then plotting the loss and finding an optimal early stopping point. It has to be so chosen that the model has enough number of epochs to learn, but not so many that it overfits.

https://github.com/zalandoresearch/fashion-mnist

How did you validate your model?

The training dataset obtained from the Tensorflow API is split into two parts – 80% of which is used to training the model and 20% of which is used for validating the model. Both the training and validation dataset is passed through a data augmentor (ImageDataGenerator) to obtained augmented image data. As a result of this, we always train and validate our model against randomly augmented data, which typically changes at each epoch. This validation provides robustness to the model, especially when we are testing the model with real-world data which almost always would be noisy.

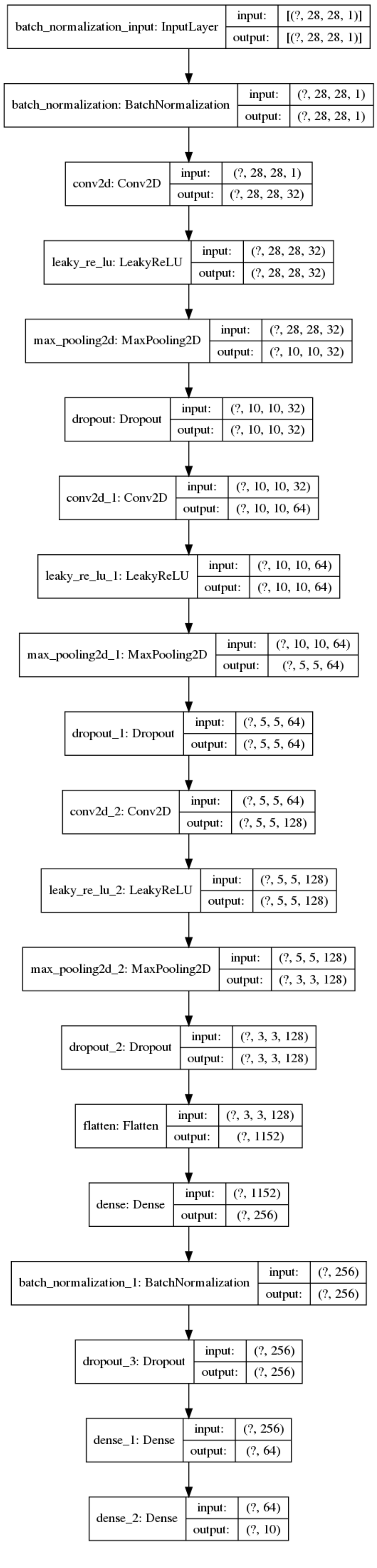
Results

The classification of FashionMNIST dataset was implemented using deep multilayer perceptron/neural network and deep convolutional neural network. The classification of test dataset resulted in an accuracy of 91.76% for CNN and 89.06% for MLNN. Both models performed well for classes such as Trouser, Sandal, Bag, and Ankle Boot, whereas the performance was low for classes such as Shirt, Tshirt, Pullover and Coat. This weak performance can be attributed to similar characteristics exhibited by these classes in terms of its appearance and its shape.

More details about the models and different plots are explained in following pages.

Model architecture – Convolutional Neural Network

(Network architecture)


batch_normalization_input: InputLayer input: [(?, 28, 28, 1)] output: [(?, 28, 28, 1)]
batch_normalization: BatchNormalization input: (?, 28, 28, 1) output: (?, 28, 28, 1)
conv2d: Conv2D input: (?, 28, 28, 1) output: (?, 28, 28, 32)
leaky_re_lu: LeakyReLU input: (?, 28, 28, 32) output: (?, 28, 28, 32)
max_pooling2d: MaxPooling2D input: (?, 28, 28, 32) output: (?, 10, 10, 32)
dropout: Dropout input: (?, 10, 10, 32) output: (?, 10, 10, 32)
conv2d_1: Conv2D input: (?, 10, 10, 32) output: (?, 10, 10, 64)
leaky_re_lu_1: LeakyReLU input: (?, 10, 10, 64) output: (?, 10, 10, 64)
max_pooling2d_1: MaxPooling2D input: (?, 10, 10, 64) output: (?, 5, 5, 64)
dropout_1: Dropout input: (?, 5, 5, 64) output: (?, 5, 5, 64)
conv2d_2: Conv2D input: (?, 5, 5, 64) output: (?, 5, 5, 128)
leaky_re_lu_2: LeakyReLU input: (?, 5, 5, 128) output: (?, 5, 5, 128)
max_pooling2d_2: MaxPooling2D input: (?, 5, 5, 128) output: (?, 3, 3, 128)
dropout_2: Dropout input: (?, 3, 3, 128) output: (?, 3, 3, 128)
flatten: Flatten input: (?, 3, 3, 128) output: (?, 1152)
dense: Dense input: (?, 1152) output: (?, 256)
batch_normalization_1: BatchNormalization input: (?, 256) output: (?, 256)
dropout_3: Dropout input: (?, 256) output: (?, 256)
dense_1: Dense input: (?, 256) output: (?, 64)
dense_2: Dense input: (?, 64) output: (?, 10)

The network architecture for the convolutional neural network is shown on the left. The input data is obtained from the Tensorflow API for the FashionMNIST dataset. The dataset loaded is feature scaled to range [0,1] for normalization.

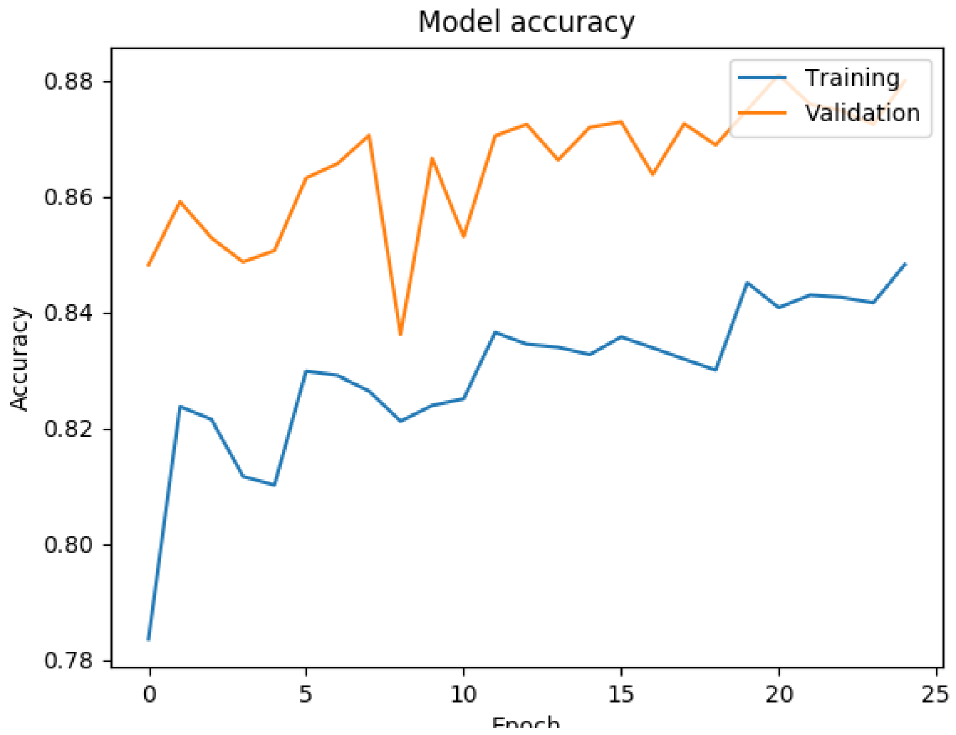
The training image data is used to produce augmented image data on-the-fly using ImageDataGenerator utility (from Keras). Since this on-the-fly generation continues infinitely, we need to explicitly specify the number of steps per epoch for training and validation. This augmented data is used to train and validate the model. The optimizer used for training is 'RMSprop()' optimizer with ReduceLROnPlateau() callback. This callback gets executed at the end of each epoch and reduces the learning rate by a specified factor if the chosen metric (here validation accuracy) does not change for specified number of consecutive epochs. The loss function used for the model was 'sparse_categorical_crossentropy' since we are dealing with integer valued classes.

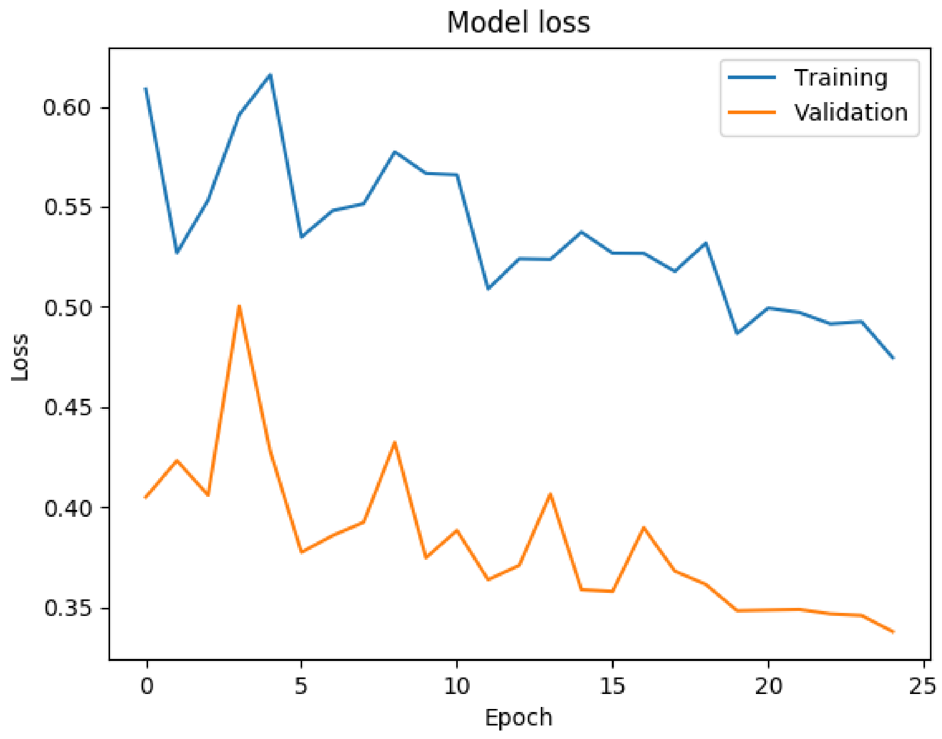
After the training for the specfied number of epochs, the final model is saved under 'model' directory with the name 'CNN.h5'.

During testing the model is loaded form the 'model' directory and the test data (from API) is used to evaluate the model. The loss, accuracy and (true class, predicted class) values are written to a file named 'convolution-neural-net.txt'

In the next pages, we can see how the network performance improved during training with the loss and accuracy vs epoch plots and how it performed on test data with the confusion matrix and classification report.

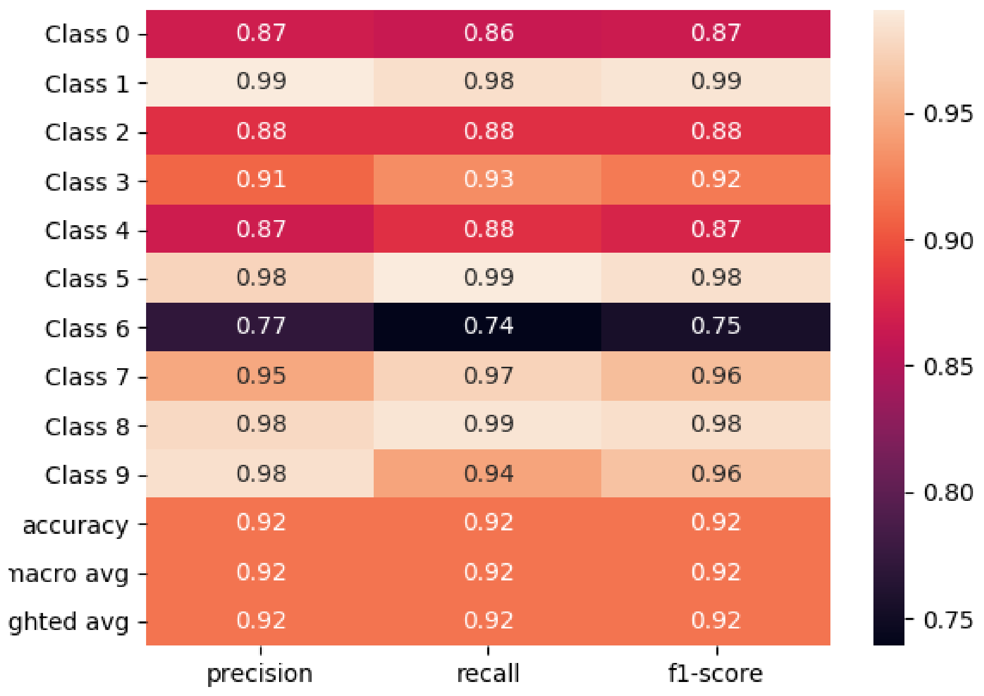
During training

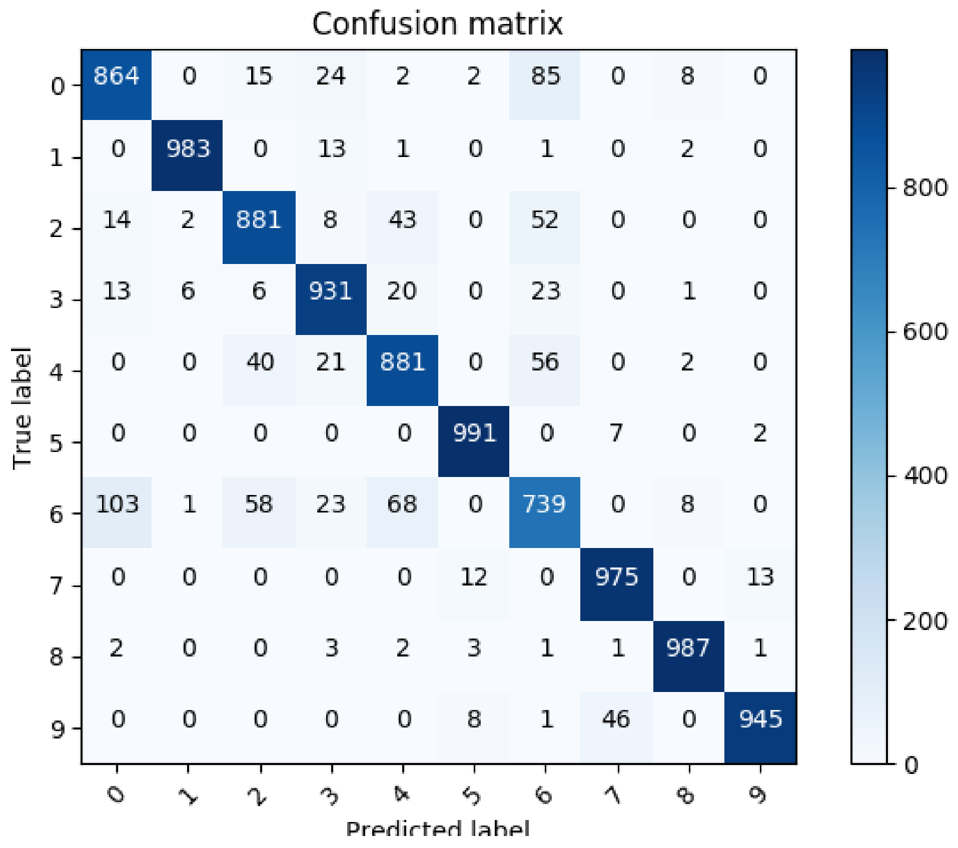




We can see that as training progresses, the model accuracy increases and, loss decreases with the number of epochs of training. They both stabilize after a specific number of epochs, after which they both degrade the performance. The training stopped before reaching that point (early stopping) to avoid overfitting the training data.

During testing

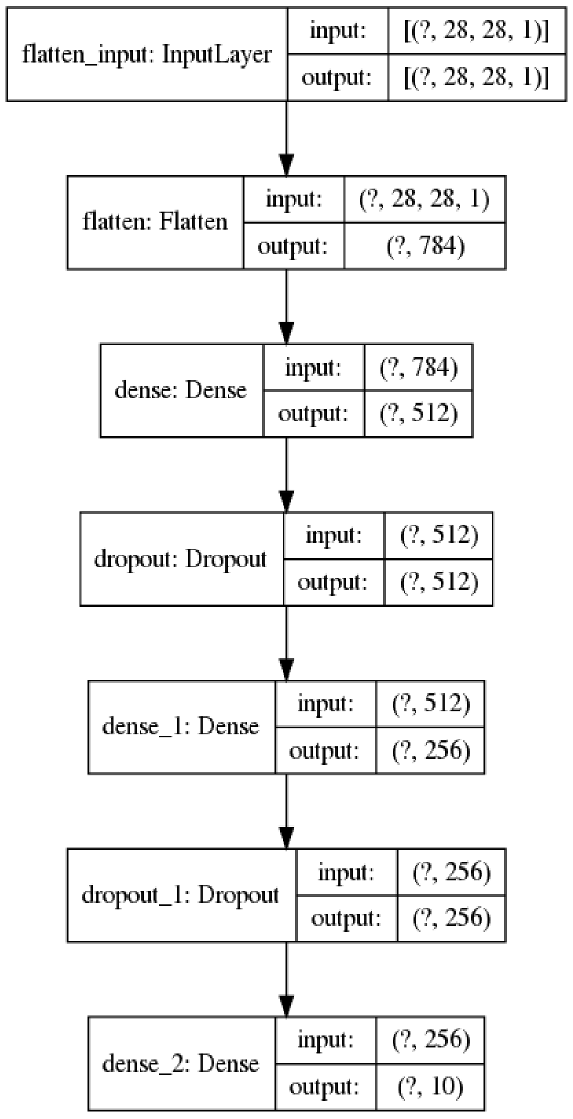




From the plots it is evident that the classifier performs exceptionally well for classes 1(Trouser), 5(Sandal), 8(Bag), and 9(Ankle Boot) whereas the performance is lowest for class 6(Shirt) and decent for classes 2(Pullover) and 4(Coat).

Model architecture – Multilayer Neural Network

(Network architecture)



The network architecture for the multilayer neural network is shown on the left. The input data is obtained from the Tensorflow API for the FashionMNIST dataset. The dataset loaded is feature scaled to range [0,1] for normalization.

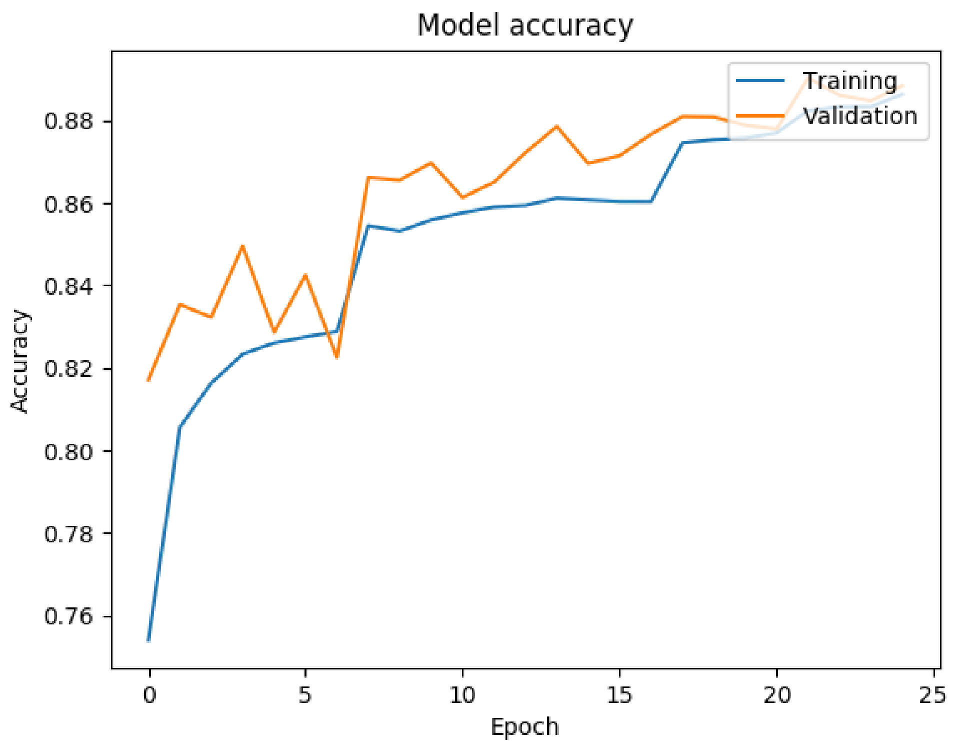
The training image data is used to produce augmented image data on-the-fly using ImageDataGenerator utility (from Keras). Since this on-the-fly generation continues infinitely, we need to explicitly specify the number of steps per epoch for training and validation. This augmented data is used to train and validate the model. The optimizer used for training is 'RMSprop()' optimizer with ReduceLROnPlateau() callback. This callback gets executed at the end of each epoch and reduces the learning rate by a specified factor if the chosen metric (here validation accuracy) does not change for specified number of consecutive epochs.The loss function used for the model was 'sparse_categorical_crossentropy' since we are dealing with integer valued classes.

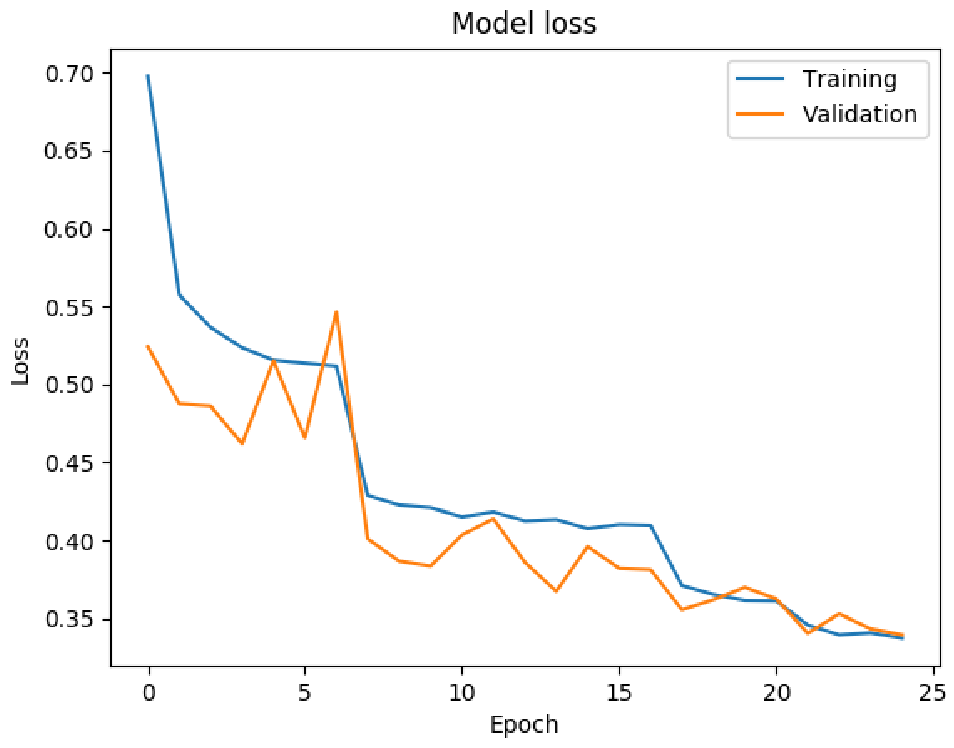
After the training for the specfied number of epochs, the final model is saved under 'model' directory with the name 'MLNN.h5'.

During testing the model is loaded form the 'model' directory and the test data (from API) is used to evaluate the model. The loss, accuracy and (true class, predicted class) values are written to a file named 'multilayer-net.txt'

In the next pages, we can see how the network performance improved during training with the loss and accuracy vs epoch plots and how it performed on test data with the confusion matrix and classification report.

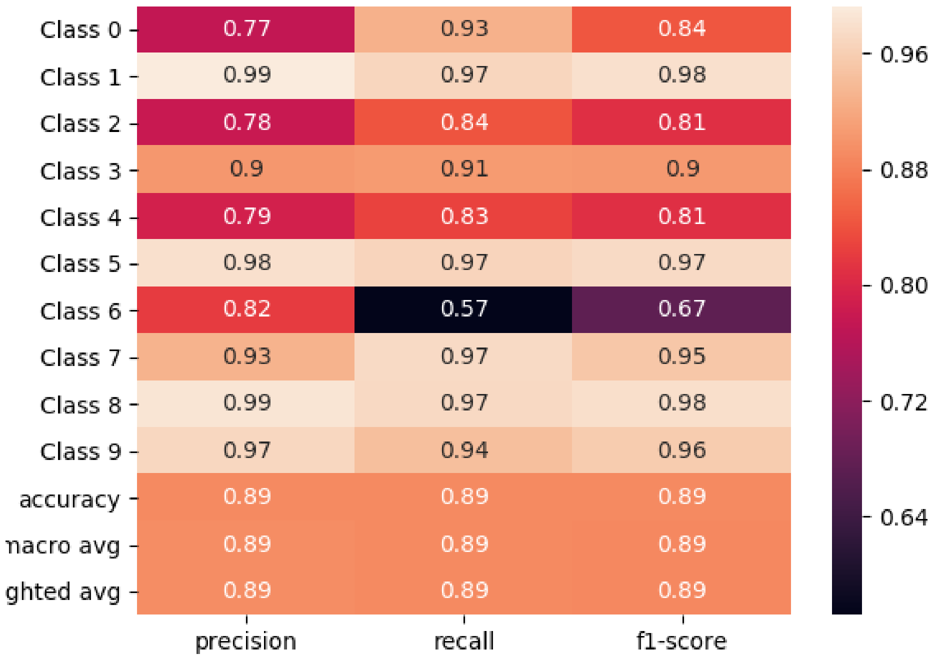
During training



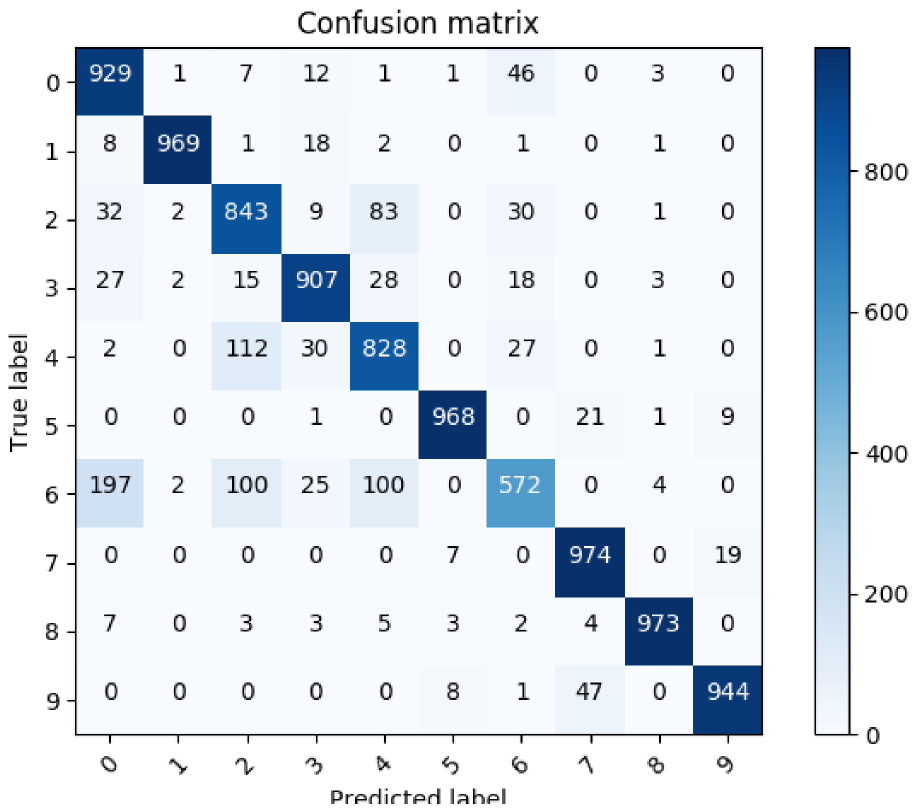


We can see that as training progresses, the model accuracy increases and, loss decreases with the number of epochs of training. They both stabilize after a specific number of epochs, after which they both degrade the performance. The training stopped before reaching that point (early stopping) to avoid overfitting the training data.

During testing



	precision	recall	f1-score
Class 0	0.77	0.93	0.84
Class 1	0.99	0.97	0.98
Class 2	0.78	0.84	0.81
Class 3	0.9	0.91	0.9
Class 4	0.79	0.83	0.81
Class 5	0.98	0.97	0.97
Class 6	0.82	0.57	0.67
Class 7	0.93	0.97	0.95
Class 8	0.99	0.97	0.98
Class 9	0.97	0.94	0.96
accuracy	0.89	0.89	0.89
nacro avg	0.89	0.89	0.89
ghted avg	0.89	0.89	0.89



	0	1	2	3	4	5	6	7	8	9
0	929	1	7	12	1	1	46	0	3	0
1	8	969	1	18	2	0	1	0	1	0
2	32	2	843	9	83	0	30	0	1	0
3	27	2	15	907	28	0	18	0	3	0
4	2	0	112	30	828	0	27	0	1	0
5	0	0	0	1	0	968	0	21	1	9
6	197	2	100	25	100	0	572	0	4	0
7	0	0	0	0	0	7	0	974	0	19
8	7	0	3	3	5	3	2	4	973	0
9	0	0	0	0	0	8	1	47	0	944

From the plots it is evident that the classifier performs exceptionally well for classes 1(Trouser), 5(Sandal), 8(Bag), and 9(Ankle Boot) whereas the performance is lowest for class 6(Shirt) and decent for classes 2(Pullover) and 4(Coat).