UNIVERSITY OF MARYLAND

ENEE633/CMSC828C PROJECT 2 Report

SUBMITTED BY,

SAKET SESHADRI GUDIMETLA HANUMATH
UID: 116332293
EMAIL: saketsgh@umd.edu

Overview

The objective of this project is to utilize the following machine learning classifiers for classifying the fashion MNIST dataset, study how they work, compare their accuracies and discuss in detail how these perform in different scenarios. The classifiers used in this project are -

1. Support Vector Machine (SVM)
    a. Implemented using Scikitlearn and Python
2. Convolutional Neural Network (CNN)
    a. Custom model built using Tensorflow/Keras

Experimentation and Results

SVM

The steps employed for implementing this classifier are listed below –

a) Load the dataset and pre-process the data.
    i) First normalize the intensities of each training sample
    ii) Apply Dimensionality reduction technique like *Linear Discriminant Analysis* to reduce the effective number of dimensions from 784 to 9(84 in case of *PCA*).
b) Fit different types of SVM including *linear* and *kernel* SVMs
    i) Fine tune the hyperparameters of SVM and observe how it affects the SVM accuracy score
c) Compare the accuracy scores of the various models.

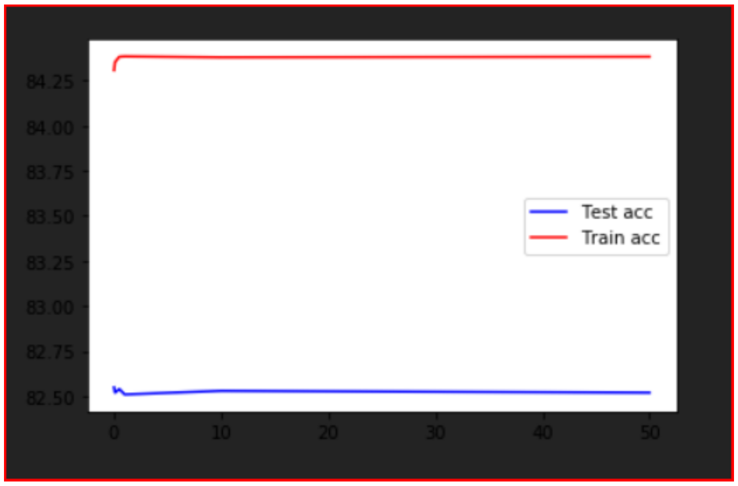
For implementing the SVM I used the *scikitlearn'*s in-built function called *SVC*.

Firstly, I started of experimenting with different types of kernels. SVM algorithms use a set of mathematical functions that are defined as the *kernel*. The function of kernel is to take data as input and transform it into the required form. Different SVM algorithms use different types of kernel functions. These functions can be different types. For example, ***linear, polynomial*** *and* ***radial basis function (RBF).***

Linear SVMs find the optimal hyperplane that maximizes the separation between classes. Linear SVM when fit on the given dataset gave the following scores –

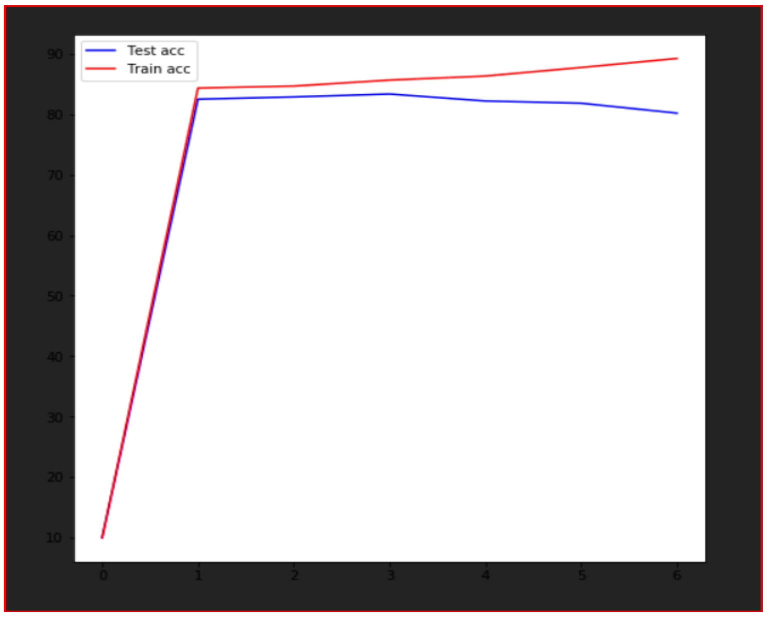
Test/Validation accuracy – 82.51%, Training accuracy – 84.38%

I then tuned the hyperparameter of SVM called the regularization parameter (C) to observe how it impacts Linear SVM performance. The C parameter tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of C will cause the optimizer to look for a larger margin separating hyperplane, even if that hyperplane misclassifies more points. The following plot of ‘accuracy score vs C ‘ summarizes the impact C has on this type of classifier.



Inference – The value of C did not have a significant impact on linear SVM performance.

I then tried polynomial SVM on the dataset. This kernel uses a non-linear hyperplane to separate the different classes. We can specify the degree of the polynomial as a parameter to the SVC. I tried different values of degree to see which one is the most optimal. The plot below of ‘accuracy vs degree or complexity of model’ shows how different values of degree causes changes in accuracy.



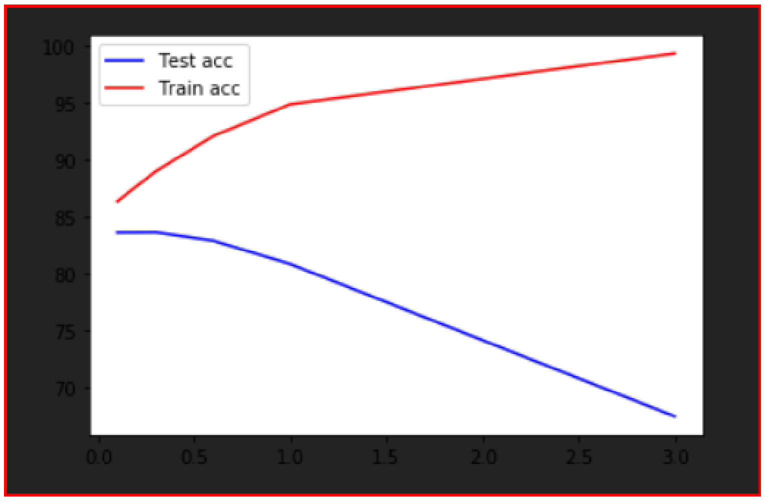
Inference – degree = 3 is the most optimal value as the validation/test accuracy peaks at this point and from that point onwards starts dipping. Choosing higher degree values would result in overfitting. Accuracy scores(best) for Poly SVM, deg = 3

Training accuracy - 85.68%, Test/Validation accuracy - 83.37%

After ‘poly’ I used the ‘rbf’ kernel which is called *Radial Basis Function.* The rbf also uses a non-linear hyperplane. The scores observed were as follows –

Test/Validation acc - 83.71%, Training accuracy - 86.545%

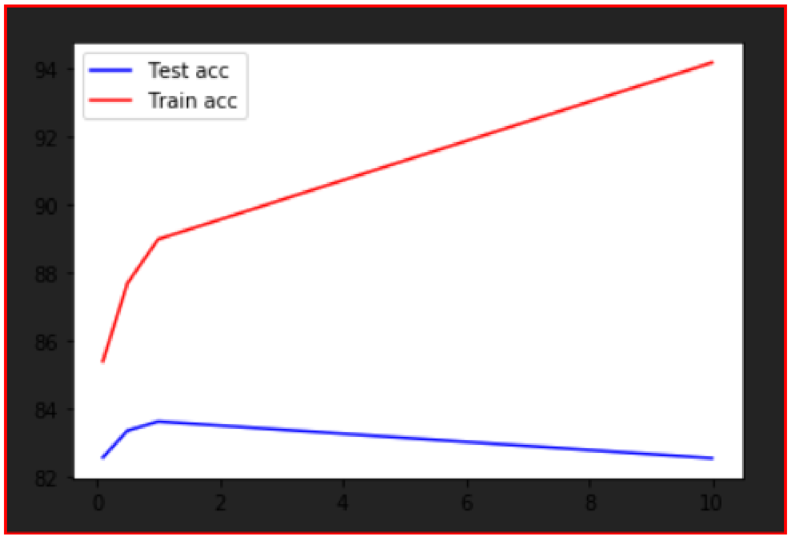
I then tried tweaking another hyperparameter called ‘gamma’. Intuitively, the gamma parameter defines how far the influence of a single training example reaches, with low values meaning ‘far’ and high values meaning ‘close’. The gamma parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors. The following plot shows how it affects the performance -



Inference - Higher values of gamma tend to overfit the model. The best value of gamma and the associated scores are below -

gamma = 0.3, Test/Validation accuracy - 83.64%, Training accuracy - 88.99%

I also tried changing the regularization parameter with rbf to see any improvement. The plot below summarizes the effort -



Inference - Higher values of C (beyond 6) were causing the model to overfit on the dataset. Thus, a lower value like C = 1 was chosen as optimal with the accuracies being nearly identical to before, peaking at around 83.64%.

I also tried the same kernels but instead of applying LDA I tried PCA using the same parameters as before.

Following table summarizes the best results of the kernels discussed till now -

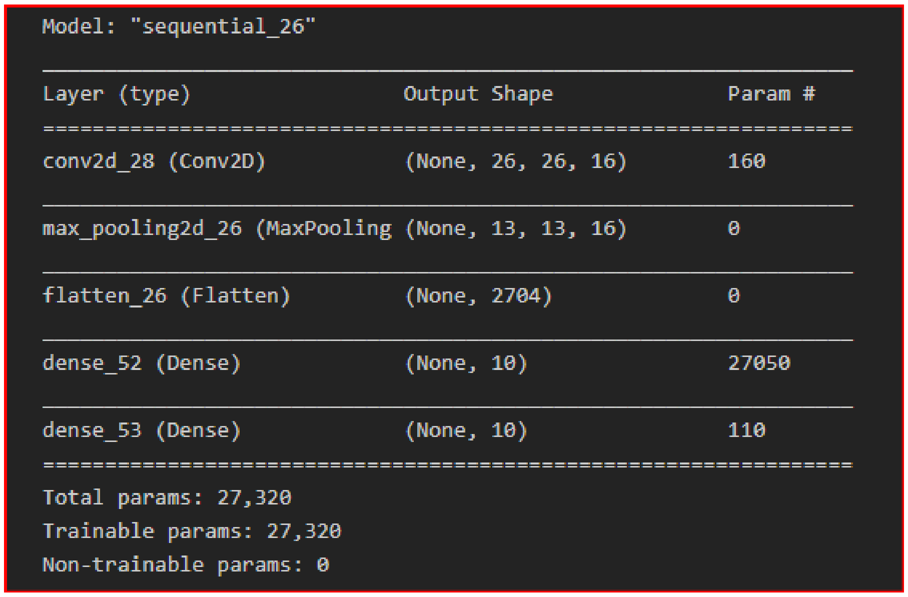
Kernel	**Train acc (LDA)**	**Train acc (PCA)**	**Test acc (LDA)**	**Test acc (PCA)**
Linear	82.51%	86.685%	84.38%	84.93%
Polynomial	83.37%	88.225%	85.68%	86.16%
RBF	83.64%	89.84%	88.99%	87.84%

CNN –

The steps employed for implementing this classifier are listed below –

a) Load the dataset and pre-process the data.
   i) Reshape the train and test variables to match the requirements of Keras/Tensorflow
   ii) Normalize the intensities of each training sample.
b) Record the performance and establish a baseline score
c) Improve model performance by tweaking and turning different knobs of the CNN architecture as follows –
   i) Epochs
   ii) Dense Units
   iii) Number of filters
   iv) Learning Rate
   v) Optimizers
   vi) Number of Conv Layers
   vii) Regularization & Dropout
d) Logging the data of the different models built using *Tensorboard.*
   i) TensorBoard provides the visualization and tooling needed for machine learning experimentation.

I had built my own Convolutional Neural Network model which is shown below –



```
Model: "sequential_26"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_28 (Conv2D)           (None, 26, 26, 16)        160
_________________________________________________________________
max_pooling2d_26 (MaxPooling (None, 13, 13, 16)        0
_________________________________________________________________
flatten_26 (Flatten)         (None, 2704)              0
_________________________________________________________________
dense_52 (Dense)             (None, 10)                27050
_________________________________________________________________
dense_53 (Dense)             (None, 10)                110
=================================================================
Total params: 27,320
Trainable params: 27,320
Non-trainable params: 0
```

Initially, I chose to observe the impact on model performance of number of epochs, so I went ahead and tested my model for epochs = [5, 10, 20, 30, 40].

This model for 5 epochs performs well with train and test accuracies as follows –

Test/Validation acc - 89.25%, Training accuracy – 90.66%

This can be used as a baseline model which can be improved upon.

Upon further evaluation of performance vs number of epochs, I had the following observations which can be explained sufficiently with the help of the Accuracy and Loss plots below –

The performance increase was not significantly varied as I altered the number of epochs. Although, a general trend of increase in accuracy with epochs was obserced . The best performance in this case, however, was obtained around ***20 epochs*** were the following accuracies were observed –

Test acc – 89.94%, Train acc – 94.03%

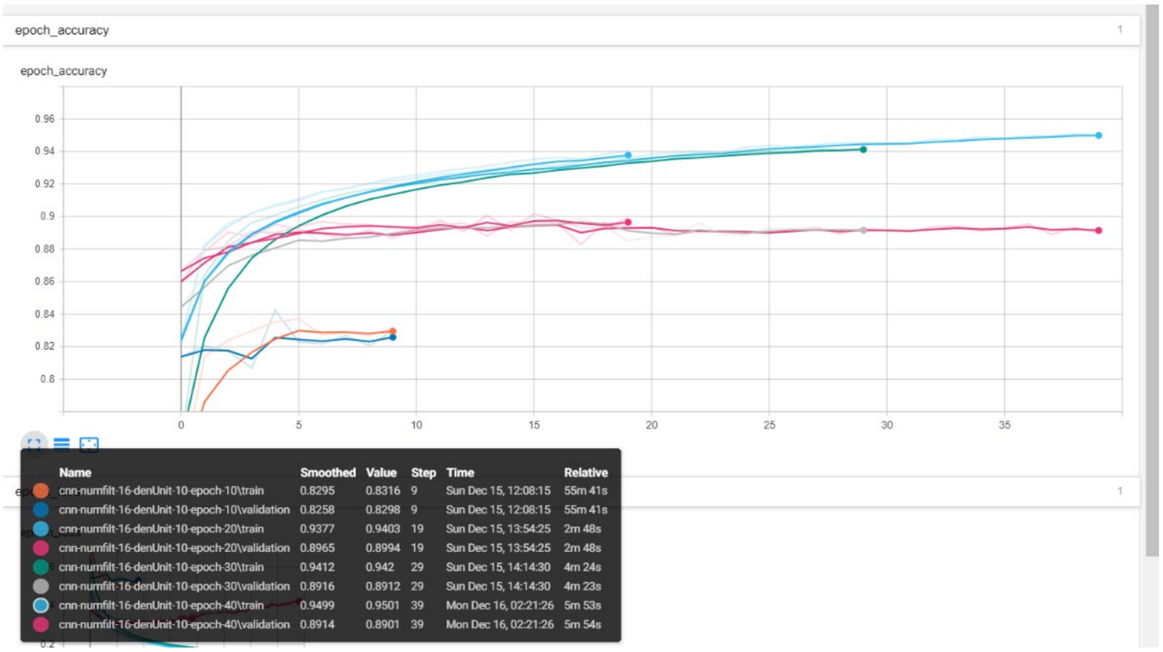
the other model parameters were left untouched.

(P.S. in order to understand the legends of the plots I have explained the terminology as follows)
Lr – learning rate

Numfilt – number of filters

denUnit – number of dense units used in the penultimate FC dense layer

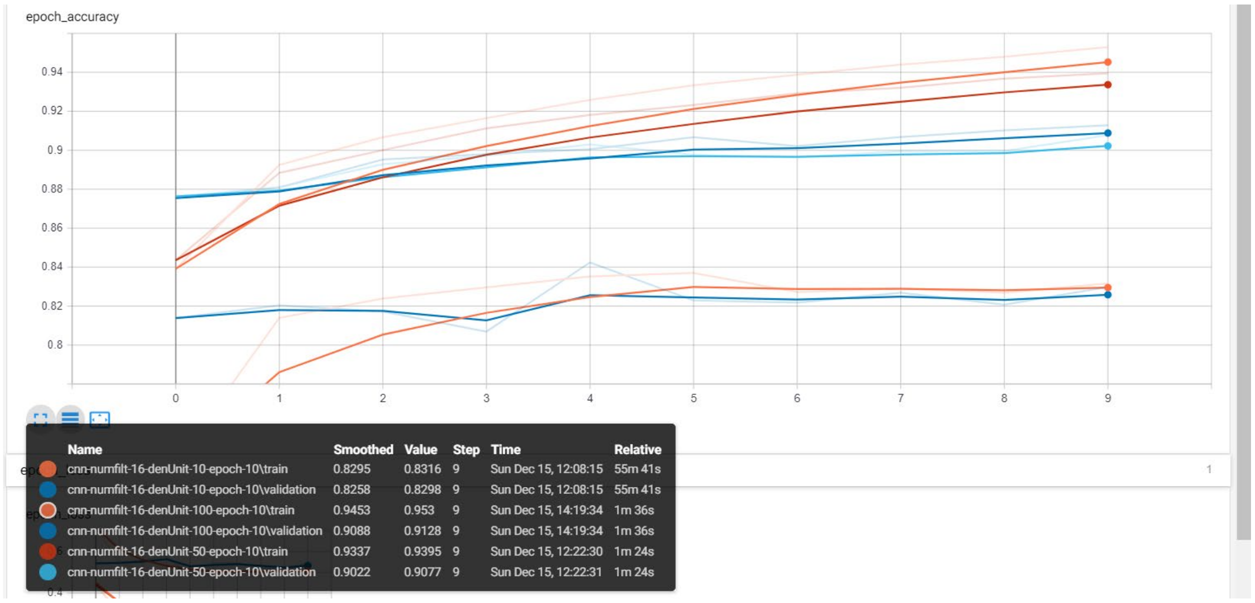
cnnx2 – 2 CNN layers



	Name	Smoothed	Value	Step	Time	Relative
●	cnn-numfilt-16-denUnit-10-epoch-10\train	0.8295	0.8316	9	Sun Dec 15, 12:08:15	55m 41s
●	cnn-numfilt-16-denUnit-10-epoch-10\validation	0.8258	0.8298	9	Sun Dec 15, 12:08:15	55m 41s
●	cnn-numfilt-16-denUnit-10-epoch-20\train	0.9377	0.9403	19	Sun Dec 15, 13:54:25	2m 48s
●	cnn-numfilt-16-denUnit-10-epoch-20\validation	0.8965	0.8994	19	Sun Dec 15, 13:54:25	2m 48s
●	cnn-numfilt-16-denUnit-10-epoch-30\train	0.9412	0.942	29	Sun Dec 15, 14:14:30	4m 24s
●	cnn-numfilt-16-denUnit-10-epoch-30\validation	0.8916	0.8912	29	Sun Dec 15, 14:14:30	4m 23s
●	cnn-numfilt-16-denUnit-10-epoch-40\train	0.9499	0.9501	39	Mon Dec 16, 02:21:26	5m 53s
●	cnn-numfilt-16-denUnit-10-epoch-40\validation	0.8914	0.8901	39	Mon Dec 16, 02:21:26	5m 54s

The next experiment I did was changing the dense units in the penultimate layer of the CNN. I experimented with different values ranging from 10 to 100. I observed that 100 dense units gave the best performance compared to the previous results with the following scores – This was expected as more units mean more weights to learn feature maps thus would contribute to increments in accuracies.

Test acc – 89.94%, Train acc – 94.03%

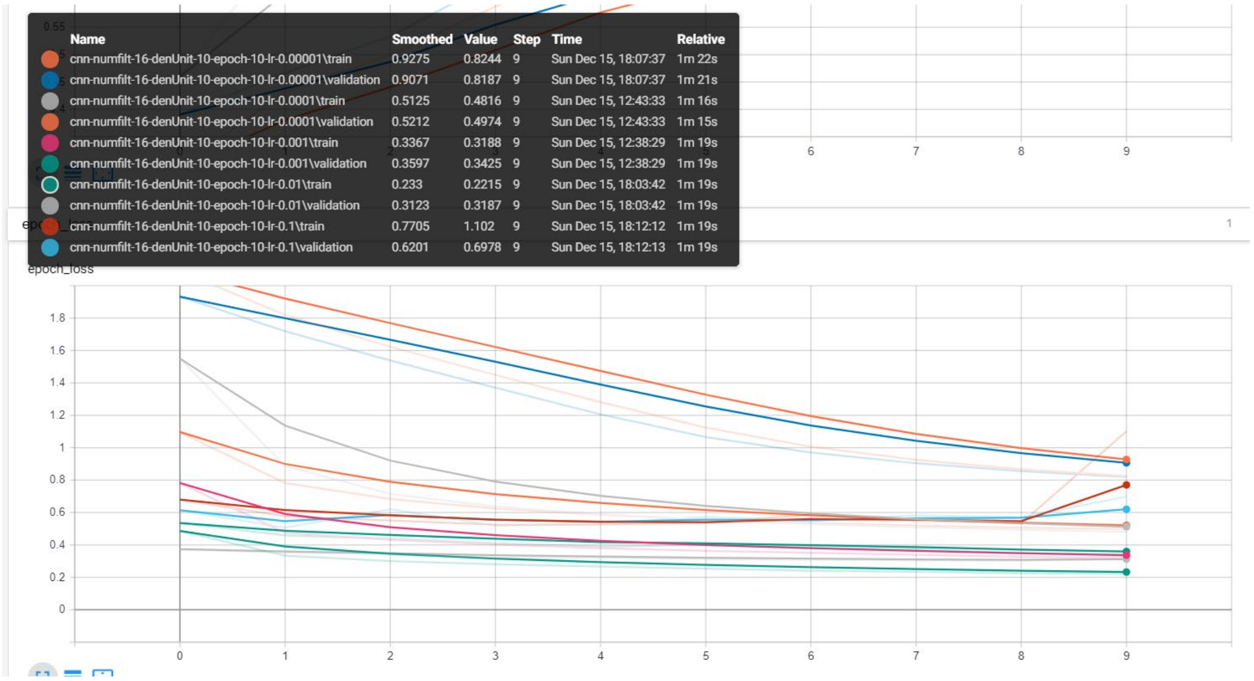


	Name	Smoothed	Value	Step	Time	Relative
●	cnn-numfilt-16-denUnit-10-epoch-10\train	0.8295	0.8316	9	Sun Dec 15, 12:08:15	55m 41s
●	cnn-numfilt-16-denUnit-10-epoch-10\validation	0.8258	0.8298	9	Sun Dec 15, 12:08:15	55m 41s
●	cnn-numfilt-16-denUnit-100-epoch-10\train	0.9453	0.953	9	Sun Dec 15, 14:19:34	1m 36s
●	cnn-numfilt-16-denUnit-100-epoch-10\validation	0.9088	0.9128	9	Sun Dec 15, 14:19:34	1m 36s
●	cnn-numfilt-16-denUnit-50-epoch-10\train	0.9337	0.9395	9	Sun Dec 15, 12:22:30	1m 24s
●	cnn-numfilt-16-denUnit-50-epoch-10\validation	0.9022	0.9077	9	Sun Dec 15, 12:22:31	1m 24s

Till this point I was working with ***Stochastic Gradient Descent Optimizer*** for my model. I wanted to see the impact of ***learning rate*** for the given problem at hand. The following plot summarizes my observations –

I varied the learning rate between 0.1 and 0.00001. Generally higher values tend to pose the risk of overshooting from the minima. Thus, I tried smaller values to see how the convergence of the algorithm is affected. The plot above show the ***cross-entropy loss*** for the various learning rates. The observation is that 0.01 is the most optimal learning rate since on either side of it we can see worse loss values. Also, the accuracies are as follows –

Train accuracy - 91.89%, Test accuracy - 88.76%



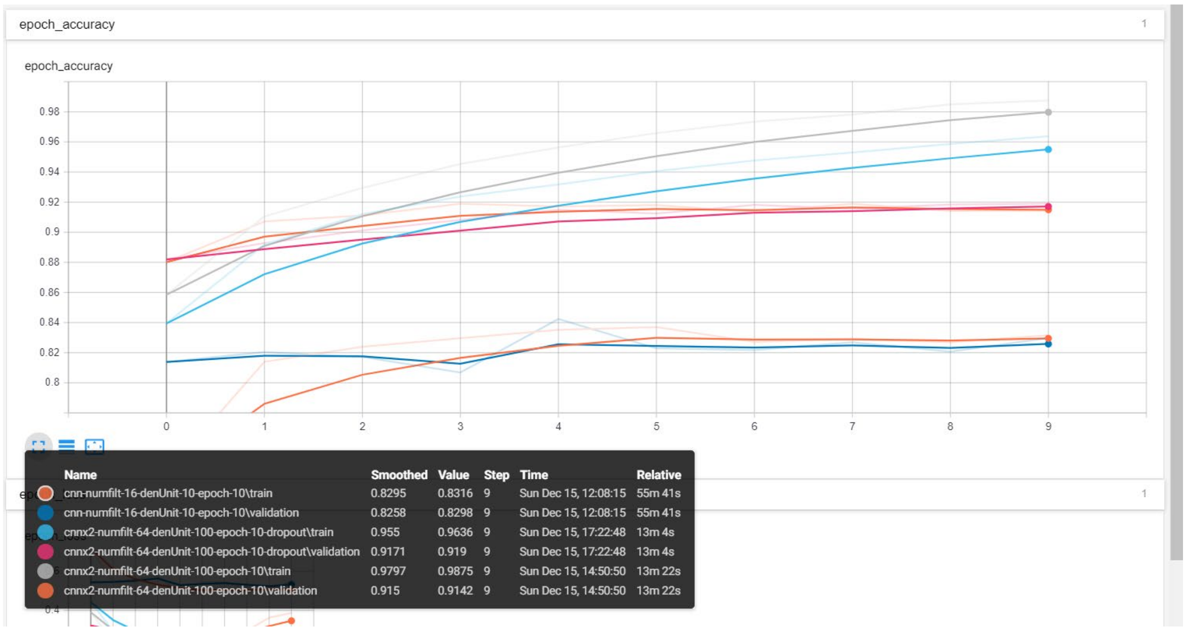
	Name	Smoothed	Value	Step	Time	Relative
●	cnn-numfilt-16-denUnit-10-epoch-10-lr-0.00001\train	0.9275	0.8244	9	Sun Dec 15, 18:07:37	1m 22s
●	cnn-numfilt-16-denUnit-10-epoch-10-lr-0.00001\validation	0.9071	0.8187	9	Sun Dec 15, 18:07:37	1m 21s
●	cnn-numfilt-16-denUnit-10-epoch-10-lr-0.0001\train	0.5125	0.4816	9	Sun Dec 15, 12:43:33	1m 16s
●	cnn-numfilt-16-denUnit-10-epoch-10-lr-0.0001\validation	0.5212	0.4974	9	Sun Dec 15, 12:43:33	1m 15s
●	cnn-numfilt-16-denUnit-10-epoch-10-lr-0.001\train	0.3367	0.3188	9	Sun Dec 15, 12:38:29	1m 19s
●	cnn-numfilt-16-denUnit-10-epoch-10-lr-0.001\validation	0.3597	0.3425	9	Sun Dec 15, 12:38:29	1m 19s
◉	cnn-numfilt-16-denUnit-10-epoch-10-lr-0.01\train	0.233	0.2215	9	Sun Dec 15, 18:03:42	1m 19s
●	cnn-numfilt-16-denUnit-10-epoch-10-lr-0.01\validation	0.3123	0.3187	9	Sun Dec 15, 18:03:42	1m 19s
●	cnn-numfilt-16-denUnit-10-epoch-10-lr-0.1\train	0.7705	1.102	9	Sun Dec 15, 18:12:12	1m 19s
●	cnn-numfilt-16-denUnit-10-epoch-10-lr-0.1\validation	0.6201	0.6978	9	Sun Dec 15, 18:12:13	1m 19s

Other nuances that I have leant and understood from various experiments are as follows (the supporting plots have not been shown in order to keep the report concise)–

- ***Dropout*** is very essential as it helps prevent overfitting. I used one dropout layer before the final fully connected layer. I found it experimentally that 0.25 was optimal. The training-validation accuracies varied gradually. It reduced train acc from ~99% to ~96%.
- I tried different ***Optimizers*** like ***adam*** and ***SGD***, but I did not observe significant change in the convergence time or epochs.
- For pooling ***Max Pooling*** is used. The maxpooling kernel size is set to (2,2). This works as there is still enough information in a (2,2) window to be used down the layers.
- The kernel size is set to minimum of (3,3). This is because the image size is already small (28,28) and by using large kernel size there won't be enough data left to add more convolution layers.
- The activation is set to ***RELU***, as it is the most recommended and as the input is normalized between 0 and 1.
- I tried different number of conv layers without significant improvement in accuracy scores. The most optimal number of layers from my observations were two conv layers with 64 filters that gives both great accuracy and requires lesser training time.

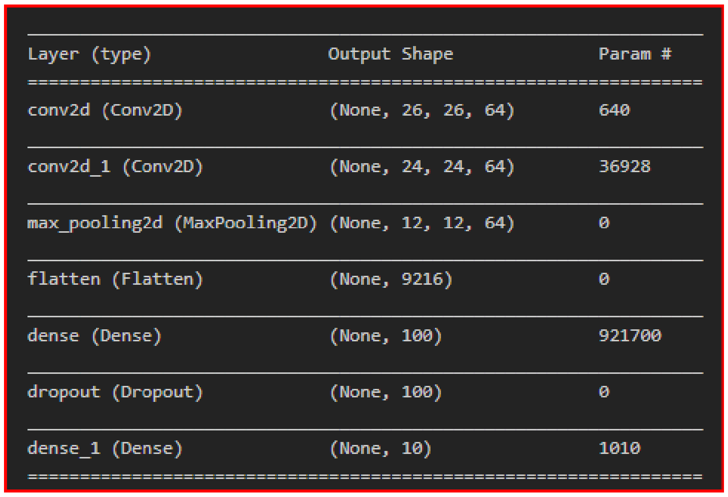
- For final fully connected layer, ***softmax*** is used as activation. Softmax proves to more suited for classification among 10 classes.

Using the above experiments and observations I made changes to my existing model and included one more Conv layer with 64 units, dropout before the last dense layer, 100 dense units and ran it for 10 epochs. The new model that I made gave the best accuracy scores. The following plot shows how this new model performs compared to other models–



	Name	Smoothed	Value	Step	Time	Relative
●	cnn-numfilt-16-denUnit-10-epoch-10\train	0.8295	0.8316	9	Sun Dec 15, 12:08:15	55m 41s
●	cnn-numfilt-16-denUnit-10-epoch-10\validation	0.8258	0.8298	9	Sun Dec 15, 12:08:15	55m 41s
●	cnnx2-numfilt-64-denUnit-100-epoch-10-dropout\train	0.955	0.9636	9	Sun Dec 15, 17:22:48	13m 4s
●	cnnx2-numfilt-64-denUnit-100-epoch-10-dropout\validation	0.9171	0.919	9	Sun Dec 15, 17:22:48	13m 4s
●	cnnx2-numfilt-64-denUnit-100-epoch-10\train	0.9797	0.9875	9	Sun Dec 15, 14:50:50	13m 22s
●	cnnx2-numfilt-64-denUnit-100-epoch-10\validation	0.915	0.9142	9	Sun Dec 15, 14:50:50	13m 22s

It is evident from the scores how the new model is better compared to other models. The final model architecture is shown below –



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 64)        640
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 64)        36928
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0
_________________________________________________________________
dense (Dense)                (None, 100)               921700
_________________________________________________________________
dropout (Dropout)            (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1010
=================================================================
```

It produces the **most accurate model** with the following scores for train and test – 96.36% & 91.9% respectively.

Conclusion –

For the given classification problem on fashion MNIST, I have successfully implemented and experimented with different classifiers such as SVM and CNN. In SVM I learned about different types of kernels, understood how parameters like regularization can affect model performance and saw variation in the outcomes of the same experiment carried out with PCA and LDA. In CNN I learned how to build a network, test it, which activation functions to use, how many filters to apply, how to prevent overfitting, how to select learning rate etc.