Project 2 – Classification of Fashion MNIST images

Name: Eshwar S R
Email ID: eshwarsr@iisc.ac.in

Description: The goal of the project is to classify a given image into a fashion category.
Dataset: Fashion MNIST
Tasks:
1. Train a Multilayer network classifier
2. Train a CNN based classifier

Dataset:
The dataset consists of 70000 images and labels. The pytorch API divides the data into 60k and 10k samples as training and test data respectively. (The splits are class balanced.)
As the end classifier is supposed to test on test set, I could not use it for hyper parameter tuning and I had to create a separate validation set from the training set.
I used the train_test_split method from sklearn.model_selection package to do a stratified split for training and validation dataset. I used 10k samples out of 60k from training data as validation dataset.
I had to spend some good amount of time in researching about stratified spliting. The default option was to use random split of samples from the training set for validation set, but having a balanced split made more sense to me.

Once the train and validation splits are created, I started of with the first task of training a Multilayer network classifier.

Preprocessing:
The dataset consists of 28x28 pixel greyscale images. So each pixel value ranged from 0 to 255.
As most of the neural networks work well with normalized data between 0 and 1, I divided the pixel values with 255 to get normalized representation.

For the first task, the network expects a vector input, I flattened the 28x28 matrix to a single vector of 784 dimensions.

For the second task, I just used the normalized 28x28 matrix as CNNs can work with matrices.

Loss function: CrossEntropy Loss
Optimizer: Stochastic Gradient Descent

Multilayer network classifier

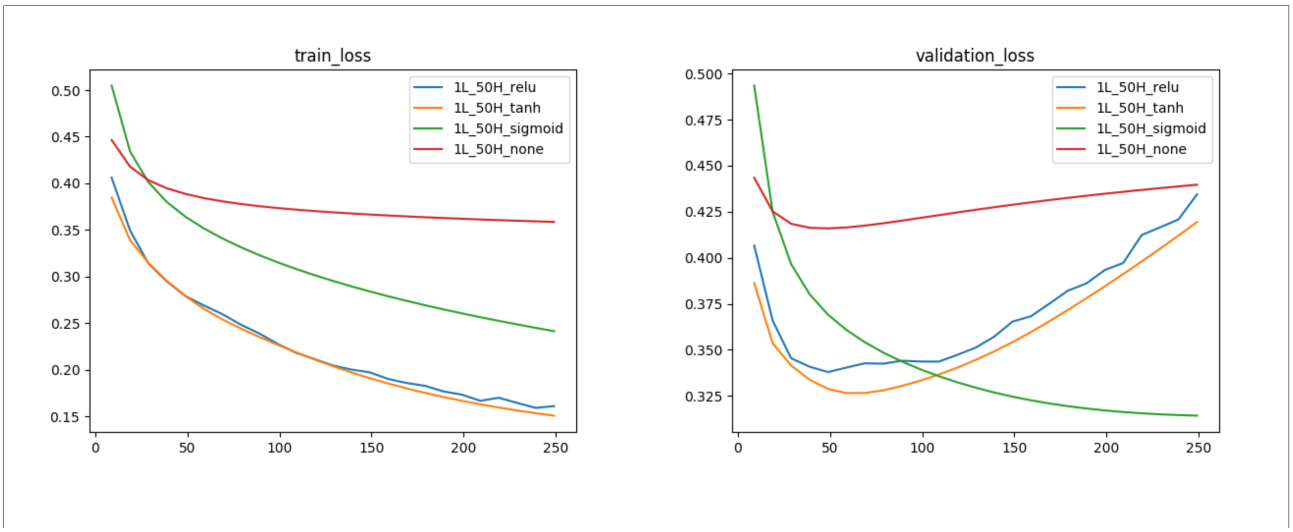
Input: 784 units.
Output: 10 units.

Experiment 1:

As part of the first experiment, I trained 4 models each with one hidden layer consisting of 50 units, followed by relu, tanh, sigmoid or no activation. Trained the models for 250 epochs with a learning rate of 0.05 and batch size of 128.

The graphs are as follows:

Note: In all the graphs, x axis represent the number of epochs and y axis represent the metric mentioned in title of the graph.

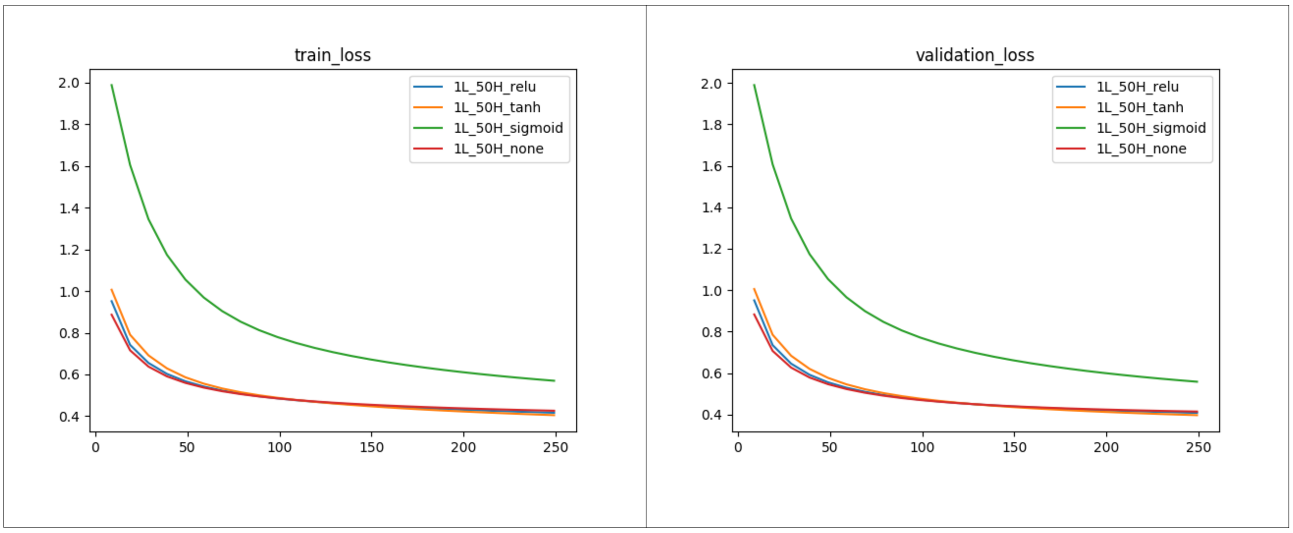


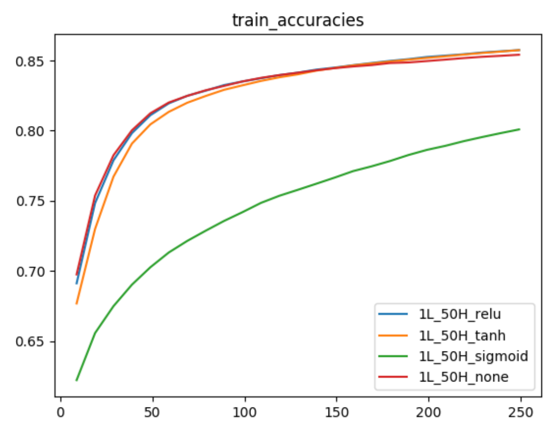
Observe the validation loss plot, initially it starts to decrease and then after certain epochs it starts to increase, though the training loss decrease. This is a clear case of overfitting to training data.

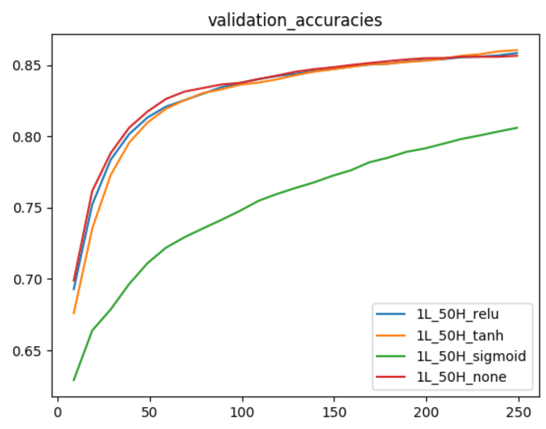
Experiment 2:

A simple model with just one hidden layer of 50 units started to overfit to the data, for just little more than 50 epochs. Hence tried decreasing the learning rate to 0.01 and increased the batch size to 1024.

The graphs are as follows:





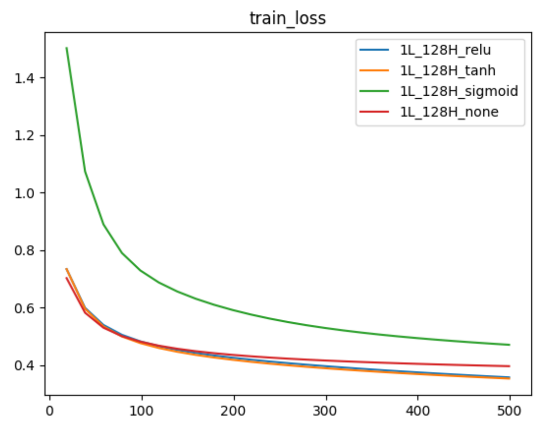


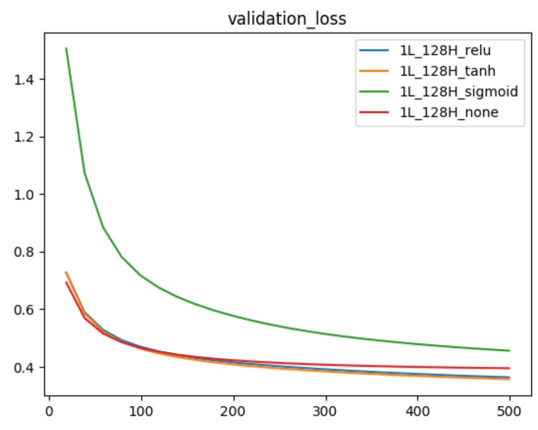
As the plot suggests, the issues of overfitting is resolved by chosing a batchsize of 1024 and learning rate of 0.01. The best accuracy is around 85.

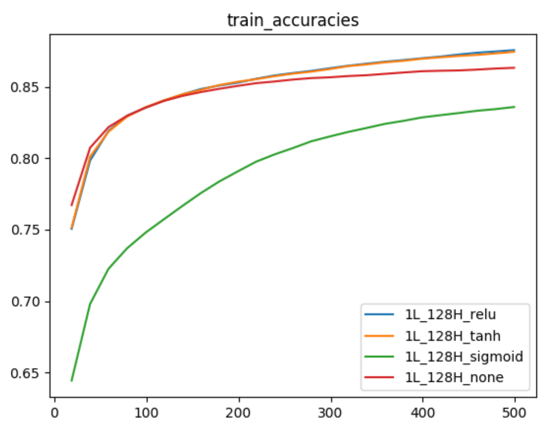
Experiment 3:

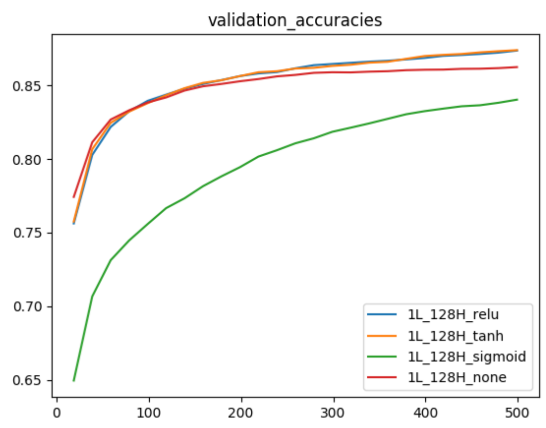
Both the train loss and validation loss are decreasing consistently, I thought of increasing the number of units in the layer to 128 and number of epochs to 500.

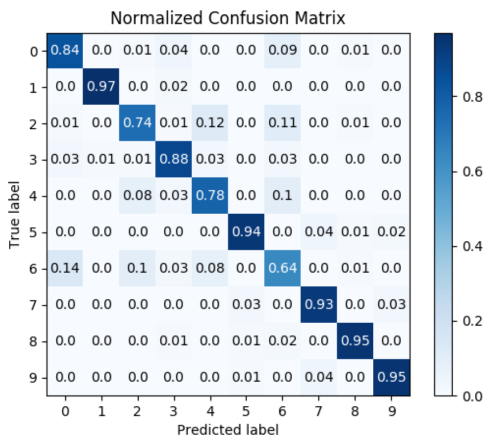
The graphs are as follows:



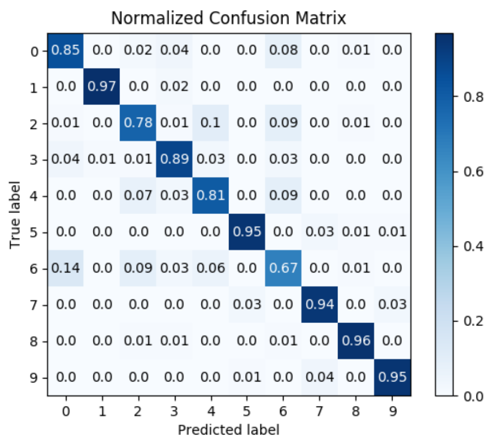




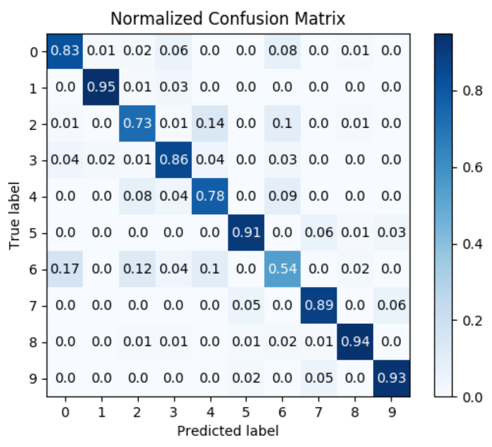




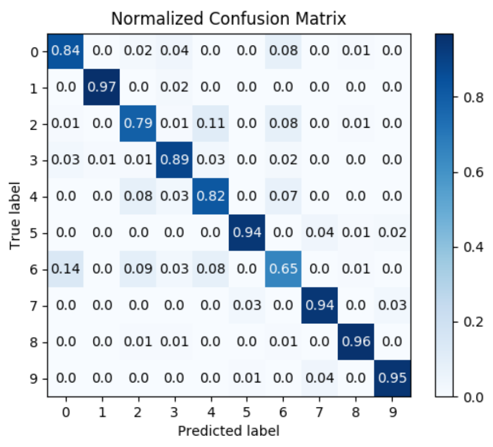
1L_128H_none



1L_128H_relu



1L_128H_sigmoid



1L_128H_tanh

The accuracies did not increase much even after increasing the hidden units and number of epochs.

Experiment 4:

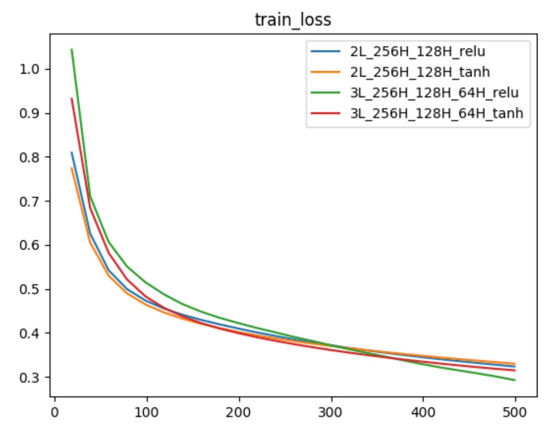
As the accuracies did not improve much, I tried adding more layers.
I ignored the None activation function while increasing the layers as it will be of no use to increase the layers in a linear setting, its just same as having more units in the same layer.
I also ignored the sigmoid activation function as it is performing less compared to relu and tanh.

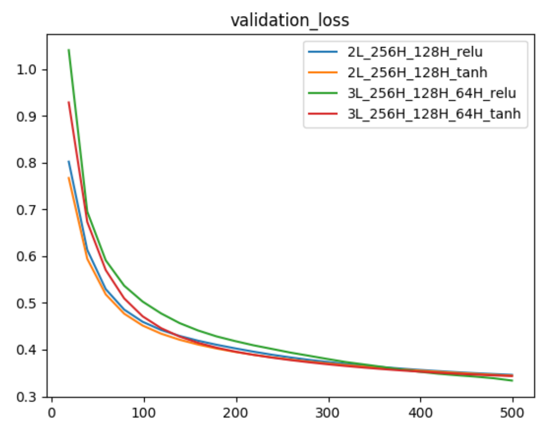
In the next experiment, I tried with 4 models.

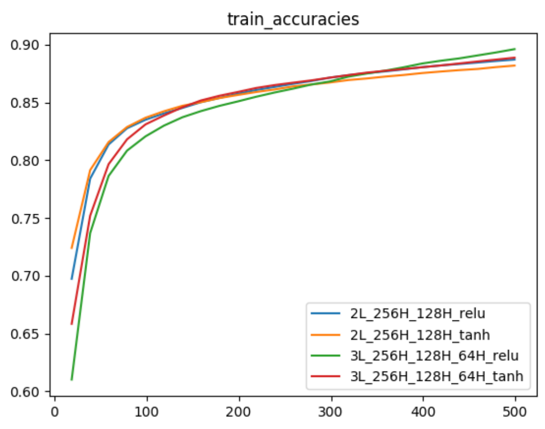
Two models, each with 2 hidden layers of 256 and 128 units respectively. I used tanh in one model and relu and another model for hidden layer activations.

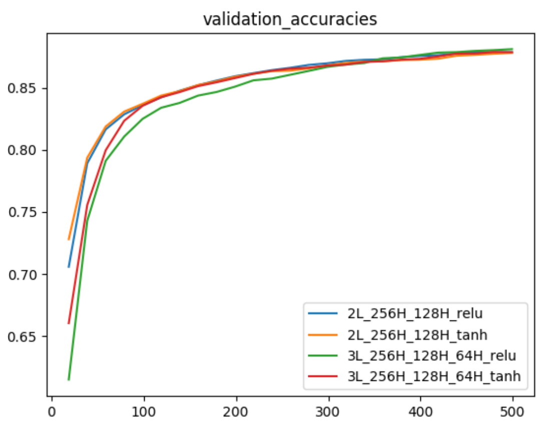
Two models, each with 3 hidden layers of 256,128 and 64 units respectively. I used tanh in one model and relu and another model for hidden layer activations.

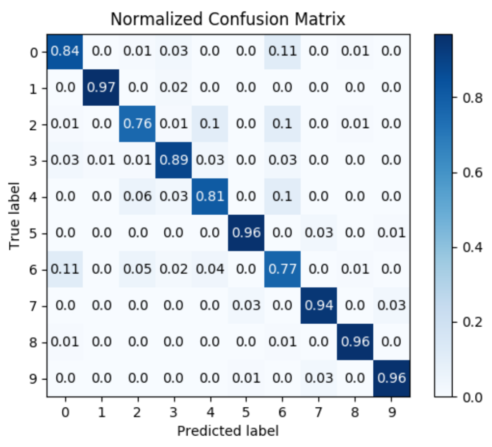
The graphs are as follows:



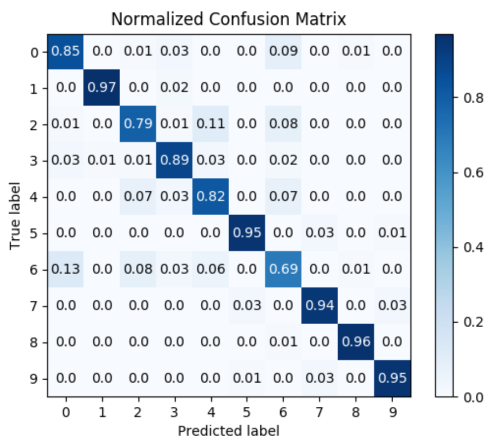




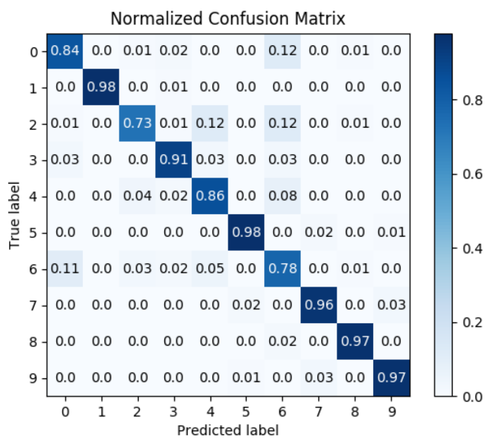




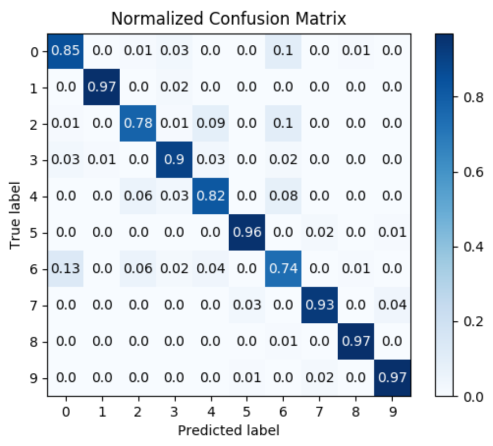
2L_256H_128H_relu



2L_256H_128H_tanh



3L_256H_128_64H_relu



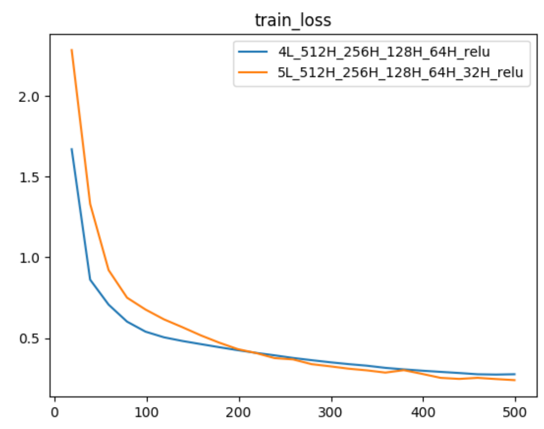
3L_256H_128_64H_tanh

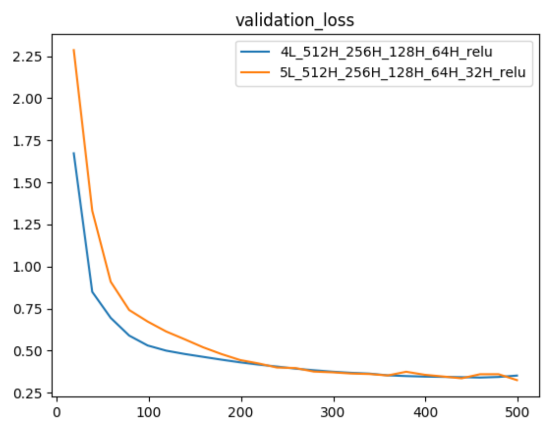
The accuracies increased a little by adding more layers. Out of all the 4 models, 3 Layer Relu model seems to provide best results.

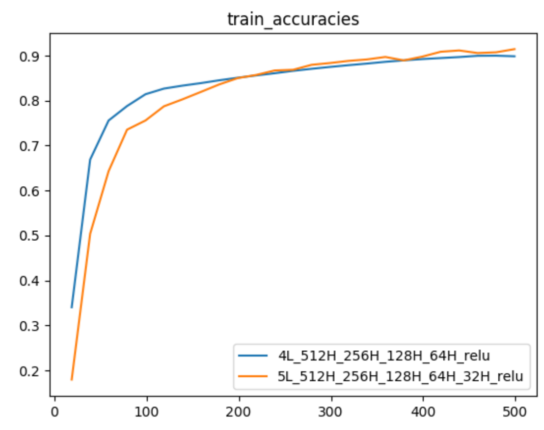
Experiment 5:

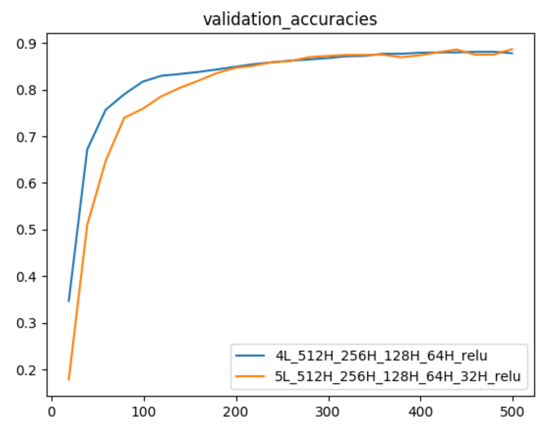
As a final experiment I wanted to try training deeper models of relu. Tried with 2 models.
One model of 4 hidden layers with 512, 256, 128 and 64 units respectively in each layer.
Another model of 5 hidden layers with 512, 256, 128, 64 and 32 units respectively in each layer.

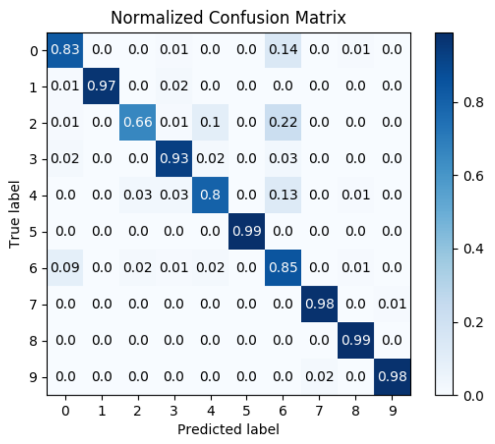
The graphs are as follows:



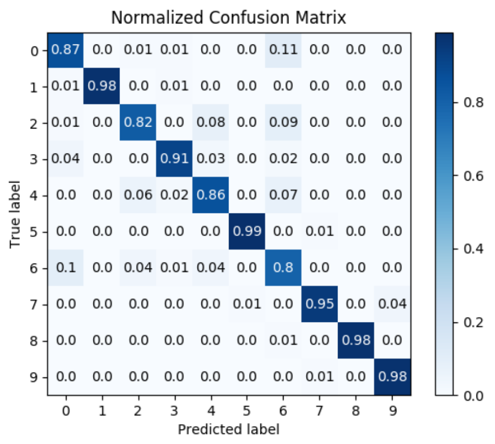








4L_512H_256H_128_64H_relu



5L_512H_256H_128_64H_32H_relu

Conclusion:

The 5 layer model performs slightly better than the 4 layer model in terms of overall accuracies. But it outperforms to a greater extent when it comes to class 2 (over 15%). It has better performance in class 0, 2 and 4. It has a slight decrease in class 6 and 7

Finally chosing the 5 layer model as it is better than 4 layer model as mentioned above.

Model Name	**Train Accuracy**	**Validation Accuracy**
4L_512H_256H_128_64H_relu	0.89	0.87
5L_512H_256H_128_64H_32H_relu	**0.91**	**0.88**

Convolutional Neural Network classifier

Input size: 1x 28 x 28
Output size: 10

The usual way of training CNN classifiers is to have Convolution layers in the beginning and as the depth increases, model is flattened and connected with fully connected layers. The intuition here is CNN layers will capture the lines, curves etc irrespective of their location in the image and later the FC layers are used to classify the observed patterns into its appropriate class.

Experiment 1:

For the first experiment, I started off with the same approach. I trained 2 models, each with 1 Convolutional layers followed by 1 FC layers.
Each Convolution layer has 2D convolution, follwed by relu activation followed by pooling. The FC layer has 128 units with relu activations.

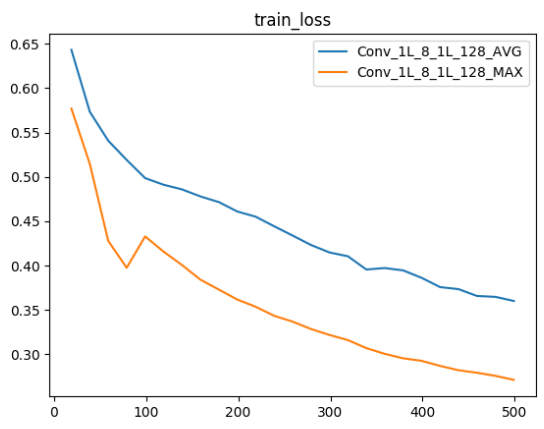
The recent literature says Relu is the default activation for training deeper networks. And from Task 1, I also observed the same. Hence I used relu activation in all of the task 2 models.

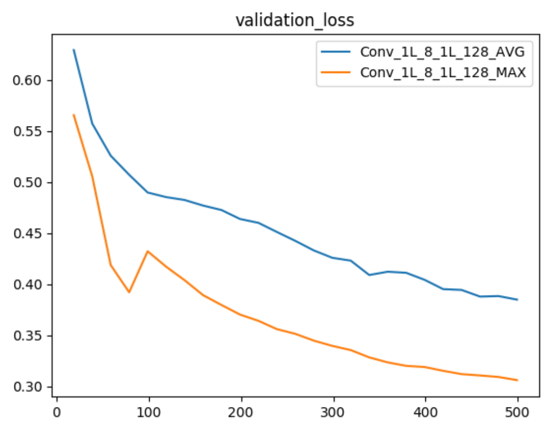
Below is the actual configuration of the models:

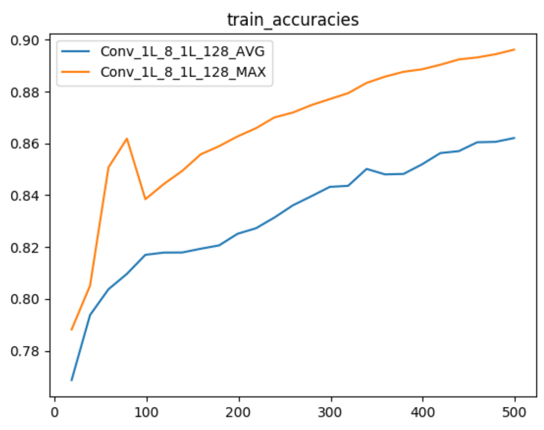
1st CNN layer:
Conv2d: number of kernels = 8, kernel_size=5, stride=1, padding=2
Relu activation
Pool2d: kernel_size=4, stride=2, padding=2 (AvgPool/MaxPool)

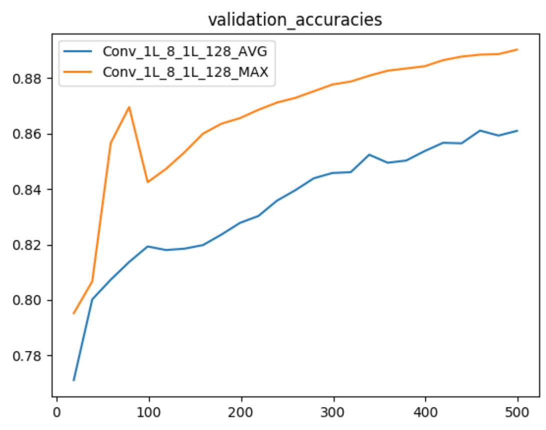
FC layer:
1 Hidden layers with 128 units with relu activations.

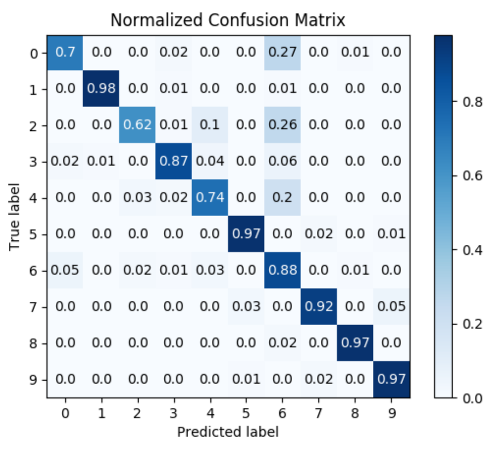
Below are the graphs for the same.



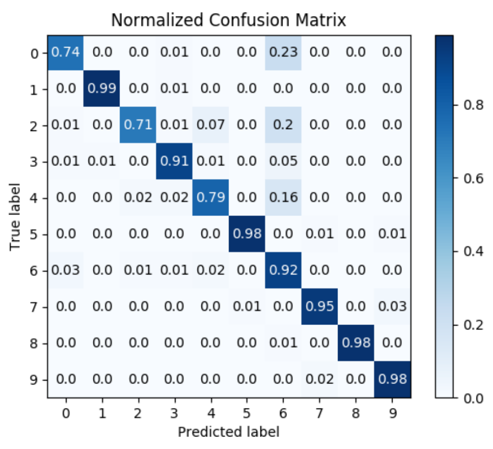








Conv_1L_8_1L_128_AVG



Conv_1L_8_1L_128_MAX

The models seems to learn the data well. The max pooling model has an accuracy of .90 which seem to be quite good with just 1 CNN followed by 1 FC layer.

Experiment 2:

Next, I trained 4 models, each with 2 Convolutional layers followed by 2 FC layers.
Each Convolution layer has 2D convolution, follwed by relu activation followed by pooling and finally regularization layer. The FC layers have 128 and 32 units respectively with relu activations.

The 4 models are combinations of Average Pooling / Max pooling in pooling layer and BatchNormalization / Dropout in regulariation layer.

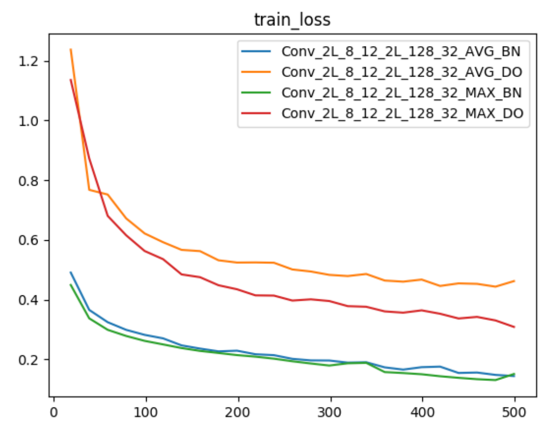
Below is the actual configuration of the models:

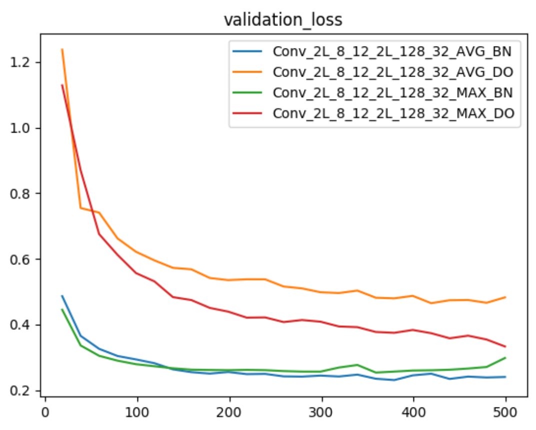
1st CNN layer:
Conv2d: number of kernels = 8, kernel_size=5, stride=1, padding=2
Relu activation
Pool2d: kernel_size=4, stride=2, padding=2 (AvgPool/MaxPool)
Regularization (Dropout/BatchNorm)

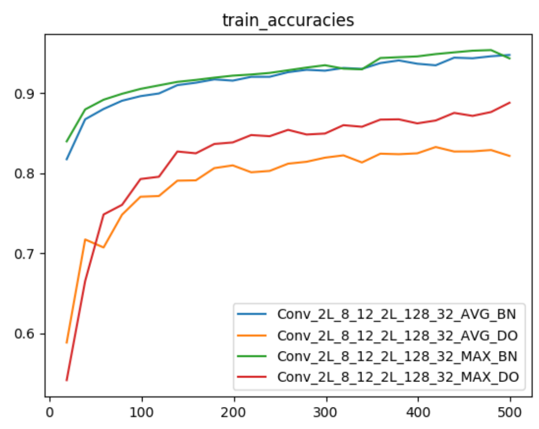
2nd CNN layer:
Conv2d: number of kernels=12, kernel_size=4, stride=1,padding=1
Relu activation
Pool2d: kernel_size=4, stride=2 (AvgPool/MaxPool)
Regularization (Dropout/BatchNorm)

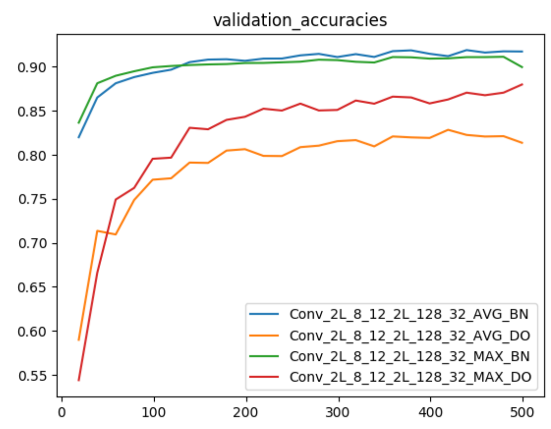
FC layers:
2 Hidden layers with 128 and 32 units respectively with relu activations.

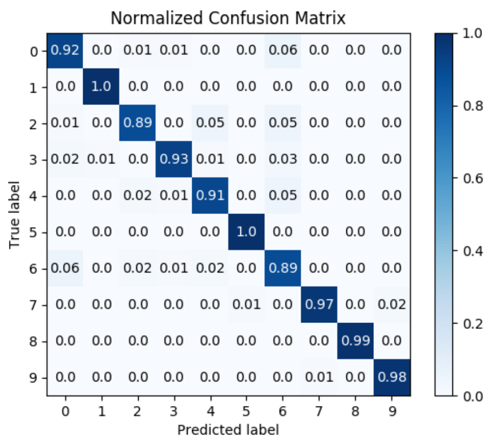
Below are the graphs for the same.



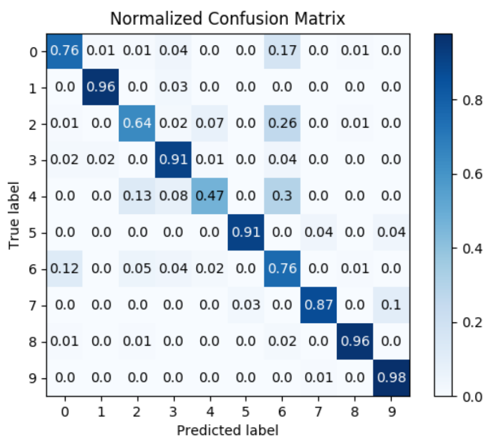




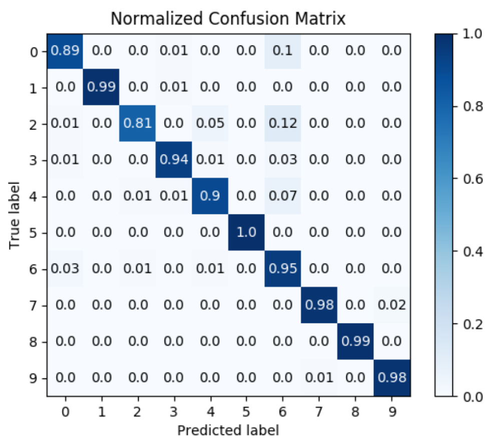




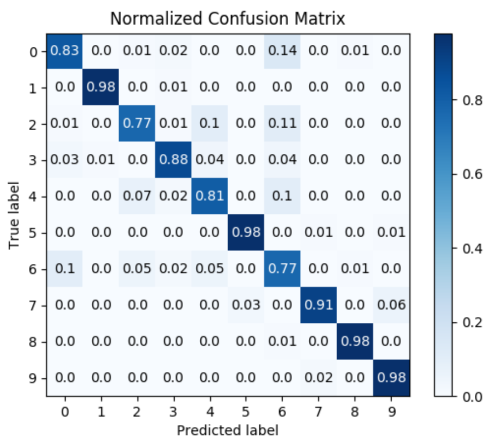
Conv_2L_8_12_2L_128_32_AVG_BN



Conv_2L_8_12_2L_128_32_AVG_DO



Conv_2L_8_12_2L_128_32_MAX_BN



Conv_2L_8_12_2L_128_32_MAX_DO

As the plot suggests, BatchNormalization is performing better than Dropout. I guess this is because of the pooling activity and the FC layers later on, the representation in the layers seem to be of quite a big range. Having the Batch Normalization in the CNN layer helped.

Even with 2 Layer of convolution followed by 2 fc layers, the accuracies are prety good for BatchNormalized models. Seems like they have generalized well. The training and validation accuracies are around .94 and .90 respectively.

Experiment 3:

The 2 layer CNN follwed by 2 layer FC layers seems to perform well. I wanted to see if adding one more layer of Convolution and FC might change the performance.
Hence I added one more layer of convolution and FC. Below are the network architectures.

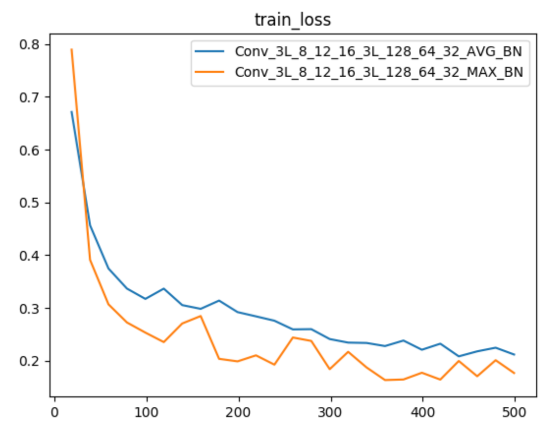
1st CNN layer:
Conv2d: number of kernels = 8, kernel_size=5, stride=1, padding=2
Relu activation
Pool2d: kernel_size=4, stride=2, padding=2 (AvgPool/MaxPool)
BatchNorm2d

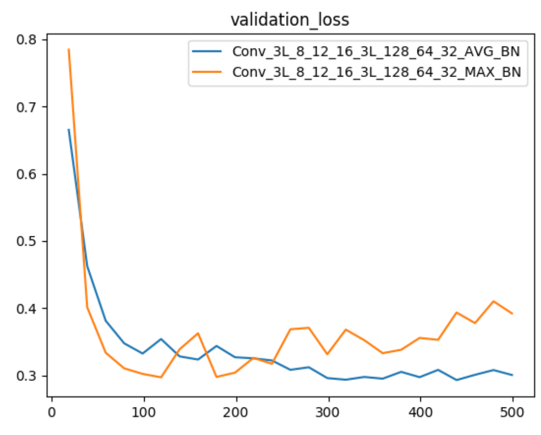
2nd CNN layer:
Conv2d: number of kernels=12, kernel_size=4, stride=1,padding=1
Relu activation
Pool2d: kernel_size=4, stride=2 (AvgPool/MaxPool)
BatchNorm2d

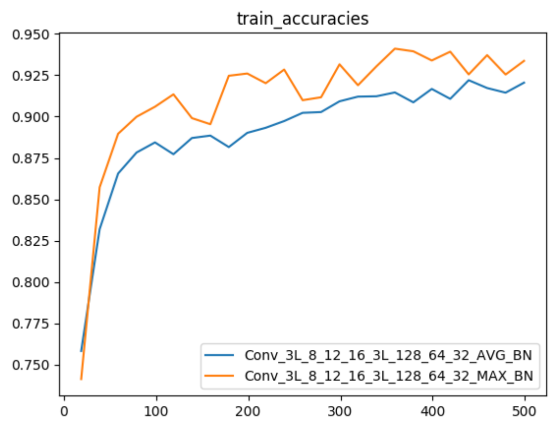
3rd CNN layer:
Conv2d: number of kernels=16, kernel_size=4, stride=1,padding=1
Relu activation
Pool2d: kernel_size=3, stride=2 (AvgPool/MaxPool)
BatchNorm2d

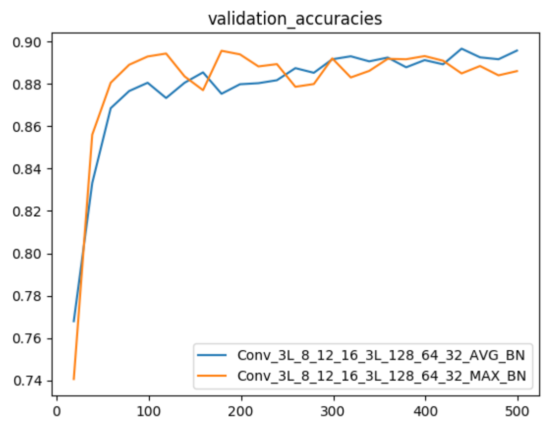
FC layers:
3 Hidden layers with 128, 64 and 32 units respectively with relu activations.

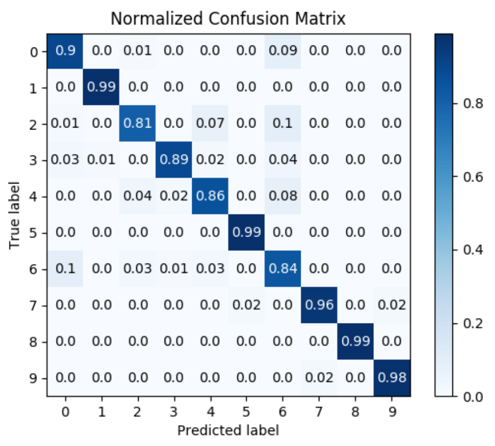
Below are the graphs for the same.





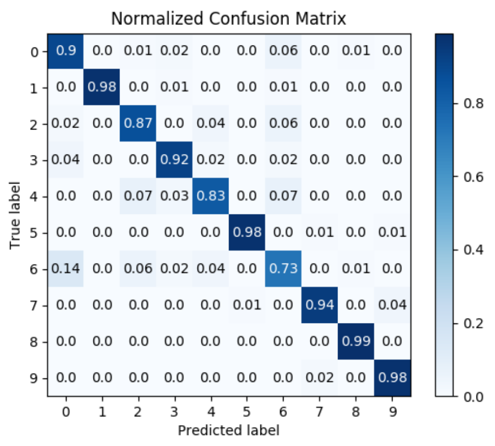






	0	1	2	3	4	5	6	7	8	9
0	0.9	0.0	0.01	0.0	0.0	0.0	0.09	0.0	0.0	0.0
1	0.0	0.99	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.01	0.0	0.81	0.0	0.07	0.0	0.1	0.0	0.0	0.0
3	0.03	0.01	0.0	0.89	0.02	0.0	0.04	0.0	0.0	0.0
4	0.0	0.0	0.04	0.02	0.86	0.0	0.08	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.99	0.0	0.0	0.0	0.0
6	0.1	0.0	0.03	0.01	0.03	0.0	0.84	0.0	0.0	0.0
7	0.0	0.0	0.0	0.0	0.0	0.02	0.0	0.96	0.0	0.02
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.99	0.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.02	0.0	0.98

Conv_3L_8_12_16_3L_128_64_32_AVG_BN



	0	1	2	3	4	5	6	7	8	9
0	0.9	0.0	0.01	0.02	0.0	0.0	0.06	0.0	0.01	0.0
1	0.0	0.98	0.0	0.01	0.0	0.0	0.01	0.0	0.0	0.0
2	0.02	0.0	0.87	0.0	0.04	0.0	0.06	0.0	0.0	0.0
3	0.04	0.0	0.0	0.92	0.02	0.0	0.02	0.0	0.0	0.0
4	0.0	0.0	0.07	0.03	0.83	0.0	0.07	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.98	0.0	0.01	0.0	0.01
6	0.14	0.0	0.06	0.02	0.04	0.0	0.73	0.0	0.01	0.0
7	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.94	0.0	0.04
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.99	0.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.02	0.0	0.98

Conv_3L_8_12_16_3L_128_64_32_MAX_BN

As the plots suggest the MAX pooling model starts to overfit to the model after 120 epochs. I was evaluating and saving the model every 20 epochs, hence the Confusion matrix shown above is that of 120 epochs. Where as the AVG pooling model does not overfit.

Conclusion:

Of all the 6 models, Conv_2L_8_12_2L_128_32_AVG_BN gave the best accuracies overall as well as class level accuracies. The performance of 3Layer model was also very similar, hence choosing the 2L one based on Occam's Razor.

Model Name	**Train Accuracy**	**Validation Accuracy**
Conv_1L_8_1L_128_AVG	0.86	0.86
Conv_1L_8_1L_128_MAX	0.89	0.89
Conv_2L_8_12_2L_128_32_AVG_BN	**0.94**	**0.91**
Conv_2L_8_12_2L_128_32_AVG_DO	0.82	0.81
Conv_2L_8_12_2L_128_32_MAX_BN	0.94	0.89
Conv_2L_8_12_2L_128_32_MAX_DO	0.88	0.87
Conv_3L_8_12_16_3L_128_64_32_AVG_BN	0.92	0.89
Conv_3L_8_12_16_3L_128_64_32_MAX_BN	0.93	0.88

Code Details:

Repo: https://github.com/EshwarSR/IISc-DL-Project-2.git

Please refer to the file README.md for more details on the code base.
PS: I have borrowed the AverageMeter code for tracting the running average. And main.py to write the txt file in the required format, from the sample repository.