Deep Learning- CSE641

Assignment - 2 [Part 2]

Name: Akanksha Shrimal	**Roll No:** MT20055
Name: Vaibhav Goswami	**Roll No:** MT20018
Name: Shivam Sharma	**Roll No**: MT20121

FILES SUBMITTED : submitted .py file , .ipynb file and readme.pdf and output.pdf

ASSUMPTIONS :

- The fit function of the network class takes one hot encoded labels and preprocessed data.
- To display the Training and Testing accuracy during each epoch we have scored training dataset only on the first 1000 samples, to reduce the lag for each epoch.
- The Dropout passed to the neural network must be an array stating the drop out probability of each layer in the network (including input and output layers) As we want a classification for 10 classes so dropout probability for output must always be passed 1 and that of input layer must be very close to 0.

UTILITY FUNCTIONS :

1. Save and Load models using Pickle

```
# Saving and Loading models using pickle
def save(filename, obj):
 with open(filename, 'wb') as handle:
     pickle.dump(obj, handle, protocol=pickle.HIGHEST_PROTOCOL)

def load(filename):
 with open(filename, 'rb') as handle:
     return pickle.load(handle)
```

2. Pre Processing data - Normalization and One Hot Encoding

```
# Utility function to normalize the data and one hot encode the
labels
def pre_process_data(train_x, train_y, test_x, test_y):
   # Normalize
   train_x = train_x / 255.
   test_x = test_x / 255.
   enc = OneHotEncoder(sparse=False, categories='auto')
   train_y = enc.fit_transform(train_y.reshape(len(train_y), -1))
   test_y = enc.transform(test_y.reshape(len(test_y), -1))
   return train_x, train_y, test_x, test_y
```

3. Plotting functions

```
# function to plot double line graph
# Plot double line using X1 , Y1 and X2 , Y2
def plot_double_line_graph(X1,Y1,label1 ,X2 ,Y2,label2
,title,y_name):
 fig = plt.figure(figsize=(7,5))
 plt.subplot(111)
  plt.plot(X1,Y1 ,label=label1 ,marker = "x" , color="blue")
 plt.plot(X2, Y2 , label=label2 ,marker = "x" , color="red")
 plt.title(title)
 plt.ylabel(y_name)
 plt.xlabel('Epochs')
 plt.legend( loc='upper left',prop={'size': 13})
 plt.show()



# Plot simgle line using X1 , Y1
def plot_single_line_graph(X1,Y1,label1, title,name_y):
 fig = plt.figure(figsize=(7,5))
 plt.subplot(111)
  plt.plot(X1,Y1 ,label=label1 ,marker = "x" , color="blue")
 plt.title(title)

 plt.ylabel(name_y)
 plt.xlabel('Epochs')
 plt.legend( loc='lower right',prop={'size': 13})
 plt.show()
```

4. Plotting ROC Curve

```
# (7,7)
#https://www.dlology.com/blog/simple-guide-on-how-to-generate-roc-
plot-for-keras-classifier/
def plot_roc(classes, y_test, y_score, figsize=(7,7)):
 n_classes = len(classes)
 # Plot linewidth.
 lw = 2

 # Compute ROC curve and ROC area for each class
 fpr = dict()
 tpr = dict()
 roc_auc = dict()
 for i in range(n_classes):
     fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
     roc_auc[i] = auc(fpr[i], tpr[i])

 # Compute micro-average ROC curve and ROC area
 fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(),
y_score.ravel())
```

```
 roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

 # Compute macro-average ROC curve and ROC area

 # First aggregate all false positive rates
 all_fpr = np.unique(np.concatenate([fpr[i] for i in
range(n_classes)]))

 # Then interpolate all ROC curves at this points
 mean_tpr = np.zeros_like(all_fpr)
 for i in range(n_classes):
     mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

 # Finally average it and compute AUC
 mean_tpr /= n_classes

 fpr["macro"] = all_fpr
 tpr["macro"] = mean_tpr
 roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

 # Plot all ROC curves
 plt.figure(1)
 plt.figure(figsize=figsize)
 plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["micro"]),
         color='deeppink', linestyle=':', linewidth=4)

 plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["macro"]),
         color='navy', linestyle=':', linewidth=4)

 colors = cycle(['aqua', 'darkorange', 'cornflowerblue'])
 for i, color in zip(range(n_classes), colors):
     plt.plot(fpr[i], tpr[i], color=color, lw=lw,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))

 plt.plot([0, 1], [0, 1], 'k--', lw=lw)
 plt.xlim([0.0, 1.0])
 plt.ylim([0.0, 1.05])
 plt.xlabel('False Positive Rate')
 plt.ylabel('True Positive Rate')
 plt.title('Some extension of Receiver operating characteristic to
multi-class')
 plt.legend(loc="lower right")
 plt.show()
```

6. Plotting Confusion Matrix

```
def confusion_matrix_find(y, y_hat, nclasses):

    y = y.astype(np.int64)
    y_hat = y_hat.astype(np.int64)

    conf_mat = np.zeros((nclasses, nclasses))

    for i in range(y_hat.shape[0]):
        true, pred = y[i], y_hat[i]
        conf_mat[true, pred] += 1
    return conf_mat
# Plotting confusion matrix
def confusion_matrix_plot(cm, classes, title='Confusion matrix',
cmap=plt.cm.Blues, figsize=(7,7), path=None, filename=None):

    cm = cm.astype(np.int64)
    plt.figure(figsize=figsize)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = 'd'
```

```
   thresh = cm.max() / 2.
   for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
       plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
   plt.ylabel('True label')
   plt.xlabel('Predicted label')
   plt.tight_layout()
```

Best Model from Part-1:

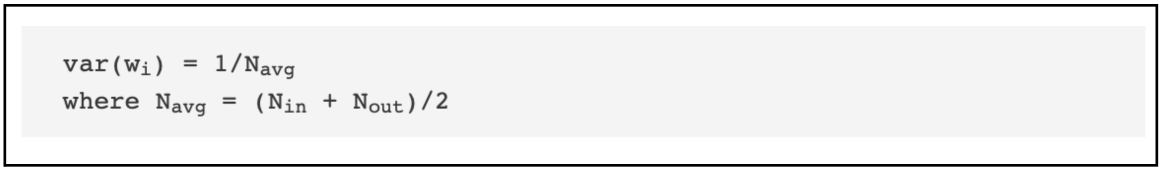
```
Network = [784,128,24,10]
Learning Rate = 0.01
Epochs = 100
Activation = Tanh
Initialization = Random
Optimizer = Adam
```

This model is further used in part too. Though originally Adam was trained for 450 epochs but it was getting overfitted so for convenience we used 100 epocs.

2. Weight Initialization:

XAVIER:

1. It is used to initialize weights randomly in such a way that inputs of the activation function falls in range of the activation function.
2. This method makes sure that inputs have zero mean and unit variance.
3. The variance remains constant throughout the network, passing through each layer.
4. It also initializes the weights with a smaller value so that the model does not start in state of saturation.
5. Variance through each layer is:

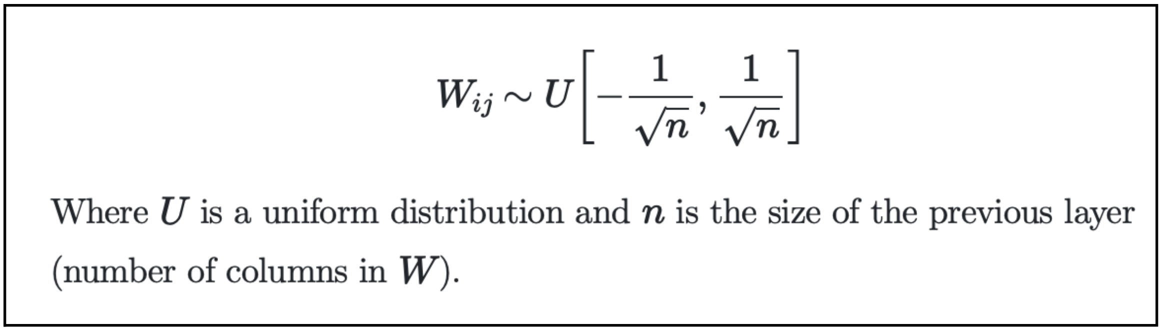


$$\text{var}(w_i) = 1/N_{avg}$$
$$\text{where } N_{avg} = (N_{in} + N_{out})/2$$

which can also be written as:

$$\sigma^2 = \frac{1}{n_{in}}$$

6. Theoretically, weights are initialized as:



$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$

Where U is a uniform distribution and n is the size of the previous layer (number of columns in W).

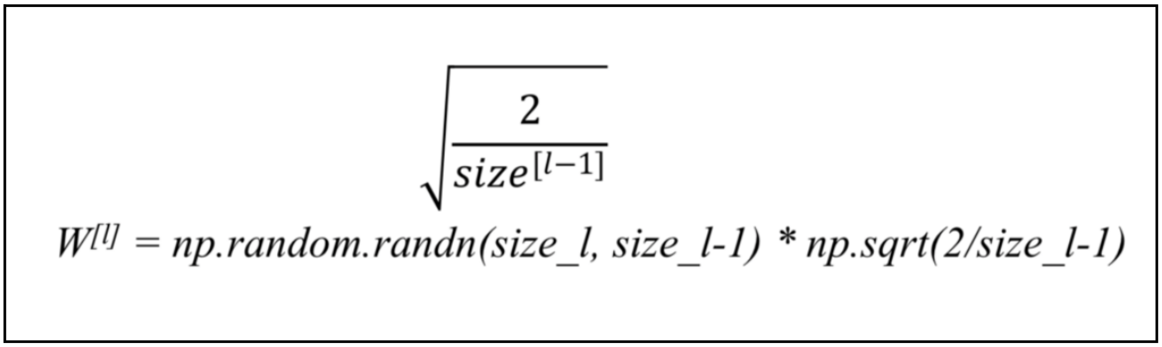
7. Xavier initialization is used for tanh activation function.
8. Overall, scaling factor and weights are updated as:

$$\sqrt{\frac{1}{size^{[l-1]}}}$$

$W^{[l]}$ = *np.random.randn(size_l, size_l-1) * np.sqrt(1/size_l-1)*

HE:

1. It initializes weights with zero mean and variance of sqrt(2/input_layer_dimension).
2. This weight initialization is similar to Xavier, differing in the variance by factor of 2.
3. It is used for ReLU activation function as it converges faster compared to Xavier for ReLU.
4. The scaling factor and weight update:



$$\sqrt{\frac{2}{size^{[l-1]}}}$$

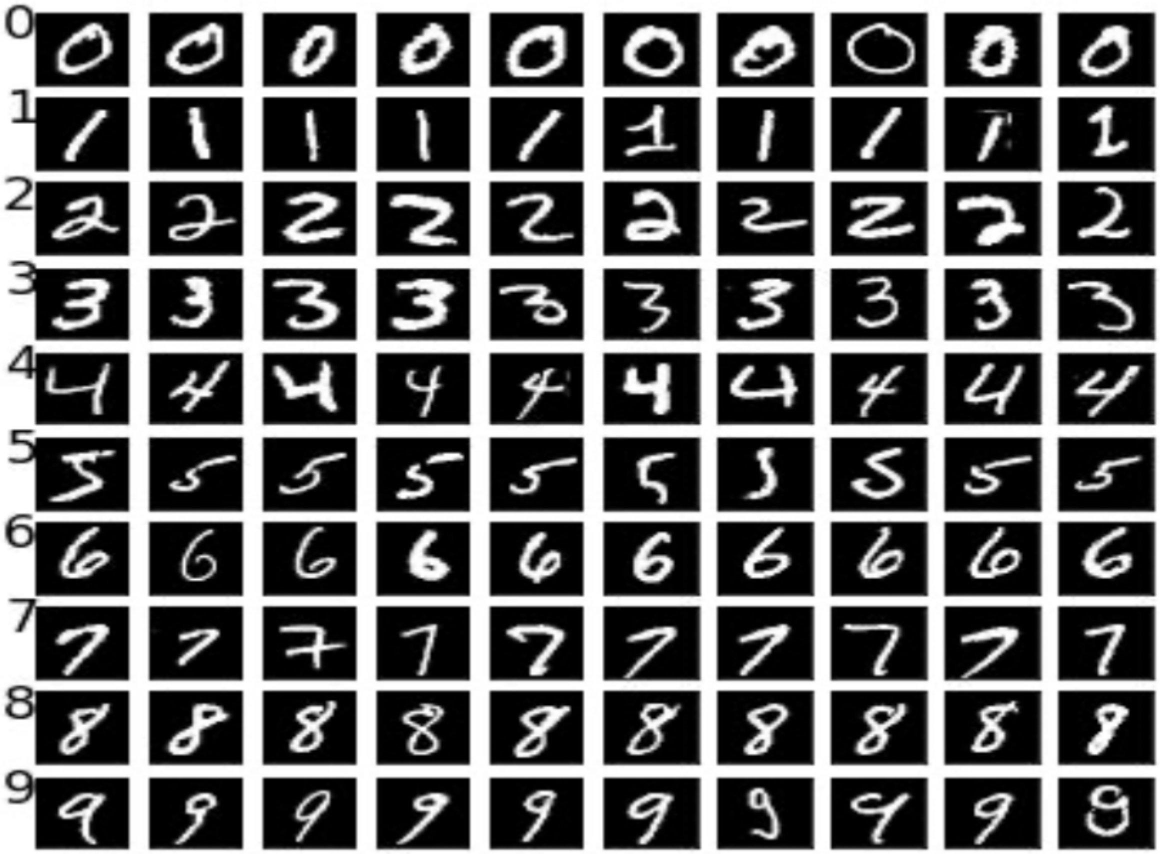
$W^{[l]}$ = np.random.randn(size_l, size_l-1) * np.sqrt(2/size_l-1)

5. It takes into account the non-linearity of ReLU activation function.

1.1 About dataset

A brief description of the dataset : MNIST

No of samples	10,000 (Training) , 2000 (Testing)
dimensions of each sample	(28,28)
Unique Labels	0,1,2,3,4,5,6,7,8,9



The following observations are made for the data :

- The data is categorical
- Each sample is a handwritten digit which are gray scale images with 28*28 pixels
- Every MNIST data point, every image can be thought as an array of numbers describing
- how dark these pixels is so each value in 28*28 represent intensity value between 0 and 1
- All the images are given labels among [0,1,2,3,4,5,6,7,8,9]

Training data is normalised using standard scaler from sklearn. And the same scaler is used to transform the Testing data.
One hot encoding is used for y-labels to do multi classification in MNIST._

1.2 Preprocessing dataset

Pre Processing Data

1. PIL Image is converted to np array
2. Image data is normalised using standard scalar form numpy.
3. Labels are one-hot encoded for multi-classification.

1.3 Weight Initialization Methodology

- ***Weight is Initialised using Xavier***

  Weights at a respective layer are initialised using the number of nodes at the previous and next layer. Here we have initialized the bias as zero and only weights initialised using xavier.

```
if self._weight_init == 'xavier':
    for i in range(len(self._layers)-1):
        self._weights[i] = np.random.randn(self._layers[i+1],
self._layers[i])*np.sqrt(1/self._layers[i])
        self._bias[i] = np.zeros((self._layers[i+1], 1))
        self._db[i] = np.zeros((self._layers[i+1], 1))
        self._dw[i] = np.zeros((self._layers[i+1],
self._layers[i]))
        self._optimizer_weight[i] = np.zeros((self._layers[i+1],
self._layers[i]))
        self._optimizer_bias[i] = np.zeros((self._layers[i+1], 1))
```

- ***Weight is Initialised using He***

He initialization used for weights and biases are kept as zero.

```
  if self._weight_init == 'he':
     for i in range(len(self._layers)-1):
       self._weights[i] = np.random.randn(self._layers[i+1],
self._layers[i])*np.sqrt(2/self._layers[i])
       self._bias[i] = np.zeros((self._layers[i+1], 1))
       self._db[i] = np.zeros((self._layers[i+1], 1))
       self._dw[i] = np.zeros((self._layers[i+1],
self._layers[i]))
       self._optimizer_weight[i] = np.zeros((self._layers[i+1],
self._layers[i]))
       self._optimizer_bias[i] = np.zeros((self._layers[i+1], 1))
```

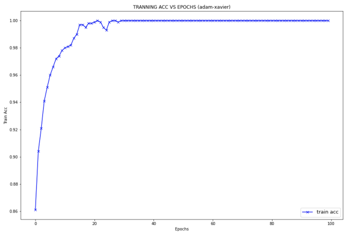
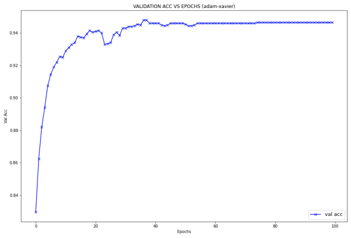
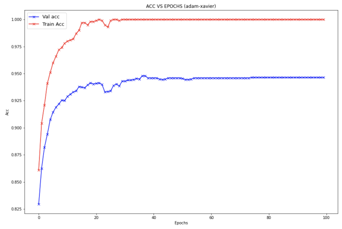
1.3 Results and Observation from Weight Initialization

1. Xavier

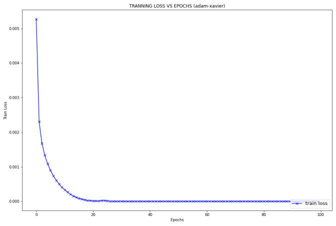
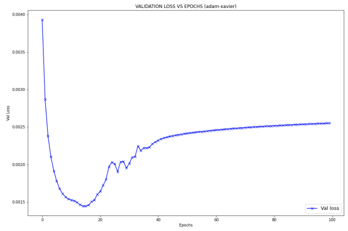
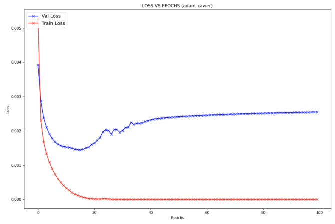
Plots

From the plots it can be observed that after approximately 19 epochs model starts overfitting and learning the training and and doing less generalization. At this time Xavier gives the accuracy which is training 98% and testing 94% (approx). Also we know Xavier initialization works very well for tanh and He is preferred for Relu.

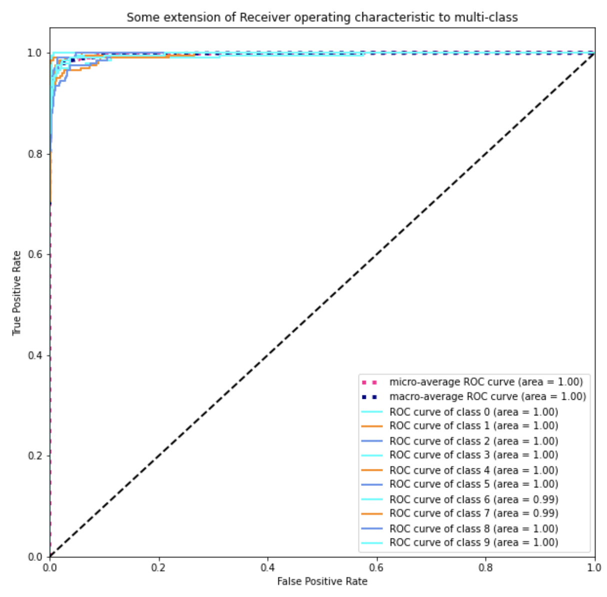
Accuracy vs Epoch

Train	Test	Combined
		

Loss vs Epoch

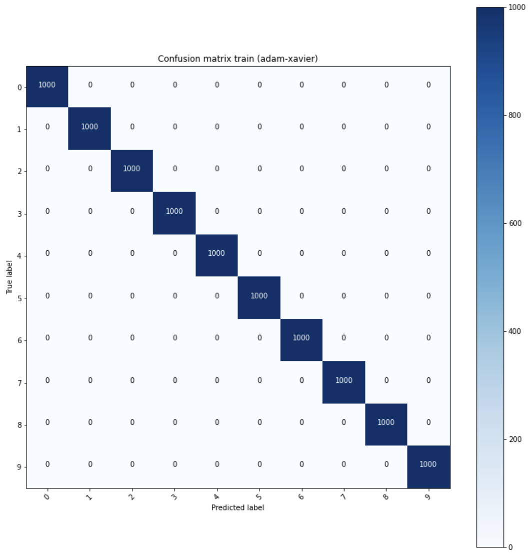
Train	Test	Combined
		

ROC curves (Val set)

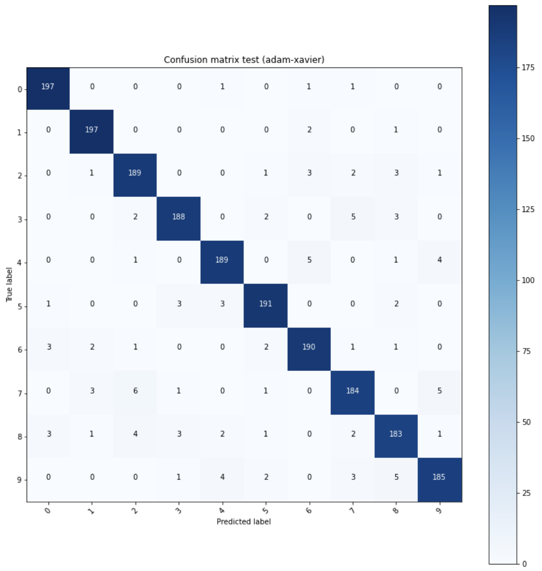


Confusion Matrix of Validation set

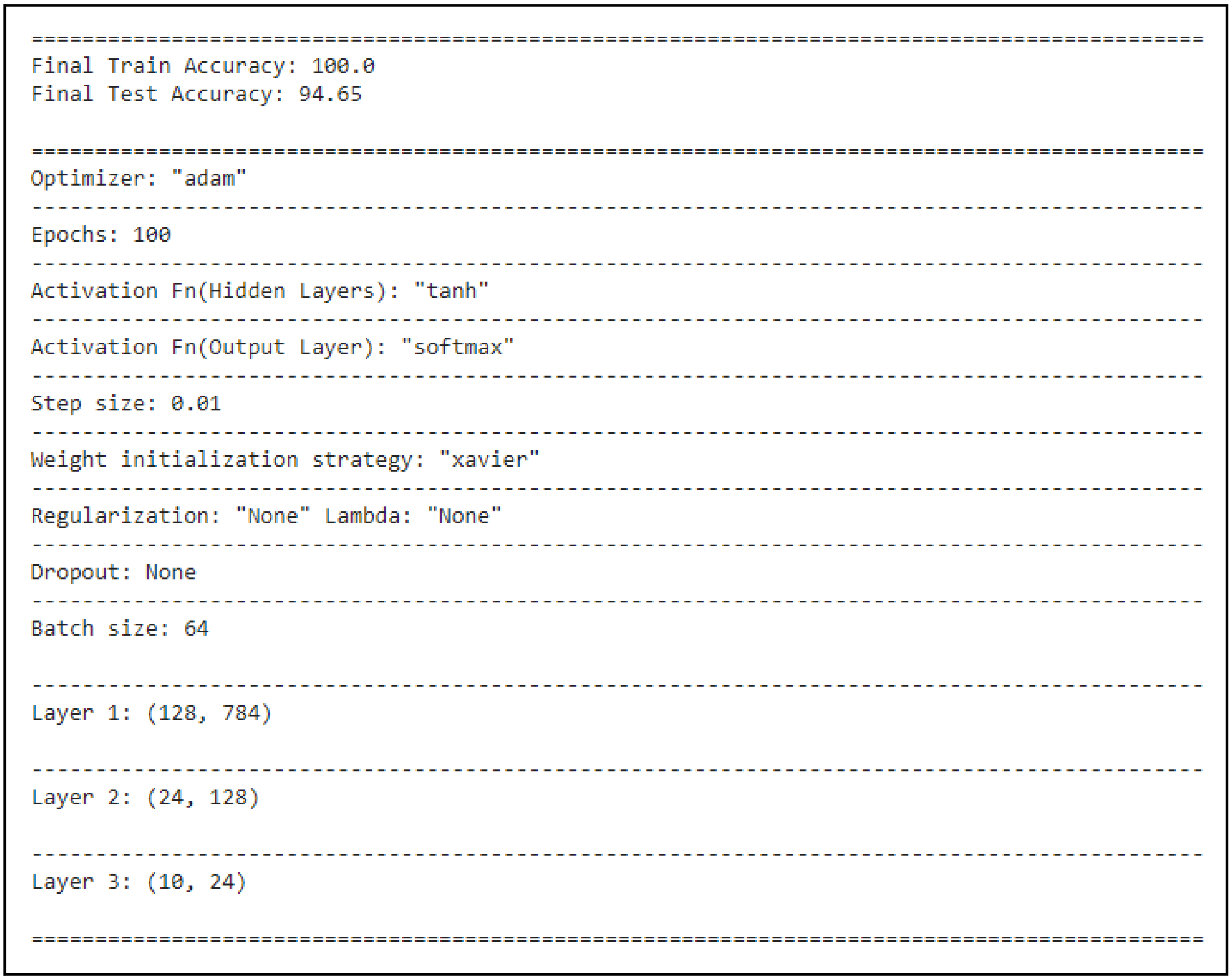
Train



Test



Observation



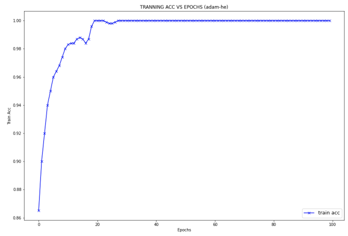
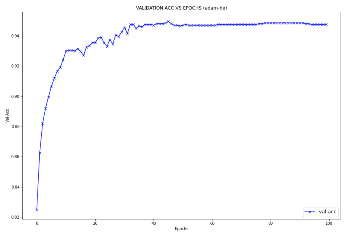
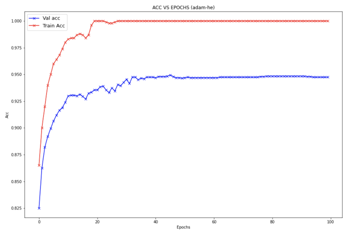
```
============================================================================================
Final Train Accuracy: 100.0
Final Test Accuracy: 94.65

============================================================================================
Optimizer: "adam"
--------------------------------------------------------------------------------------------
Epochs: 100
--------------------------------------------------------------------------------------------
Activation Fn(Hidden Layers): "tanh"
--------------------------------------------------------------------------------------------
Activation Fn(Output Layer): "softmax"
--------------------------------------------------------------------------------------------
Step size: 0.01
--------------------------------------------------------------------------------------------
Weight initialization strategy: "xavier"
--------------------------------------------------------------------------------------------
Regularization: "None" Lambda: "None"
--------------------------------------------------------------------------------------------
Dropout: None
--------------------------------------------------------------------------------------------
Batch size: 64

--------------------------------------------------------------------------------------------
Layer 1: (128, 784)

--------------------------------------------------------------------------------------------
Layer 2: (24, 128)

--------------------------------------------------------------------------------------------
Layer 3: (10, 24)

============================================================================================
```

2. He

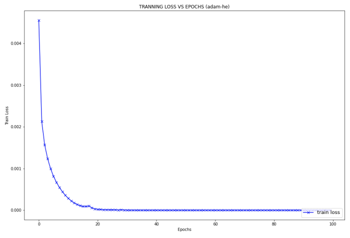
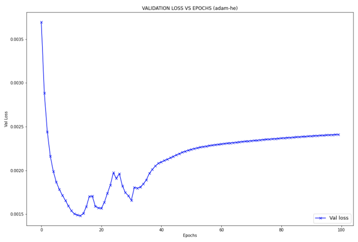
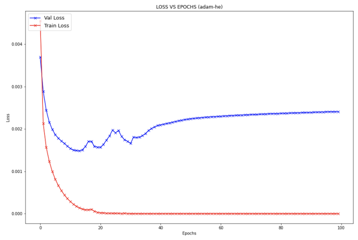
Plots

From the plots it can be observed that after approximately 19 epochs model starts overfitting and learning the training and and doing less generalization. At this time He gives the accuracy which is training 98% and testing 93% (approx). Also we know Xavier initialization works very well for tanh and He is preferred for Relu.

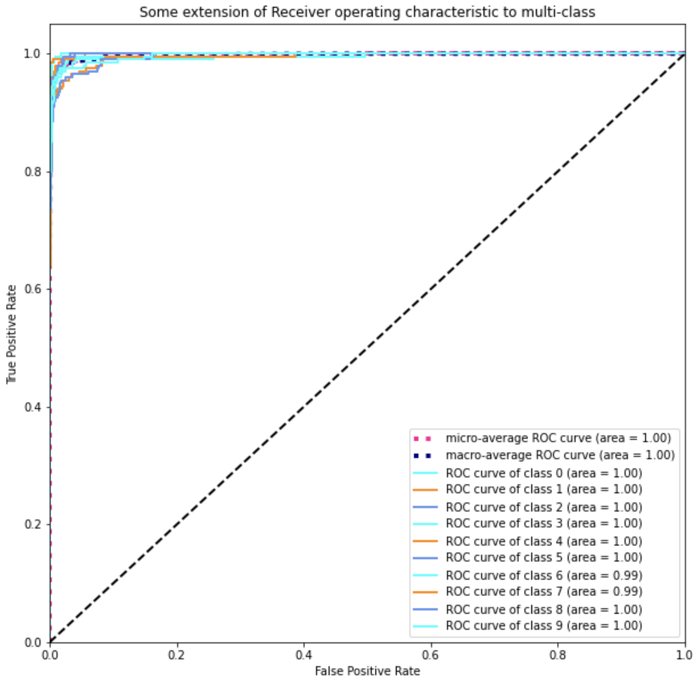
Accuracy vs Epoch

Train	Test	Combined
		

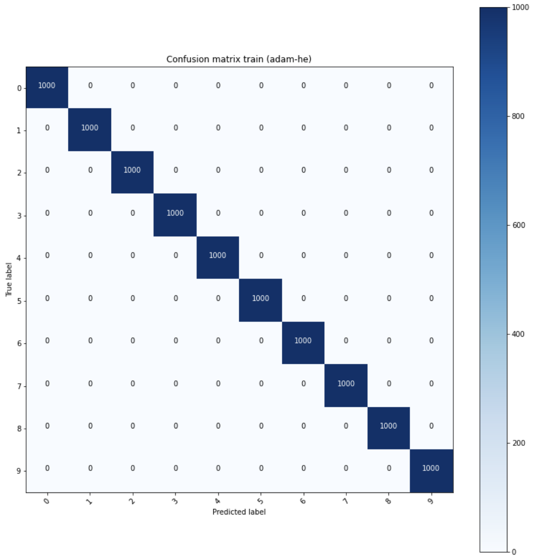
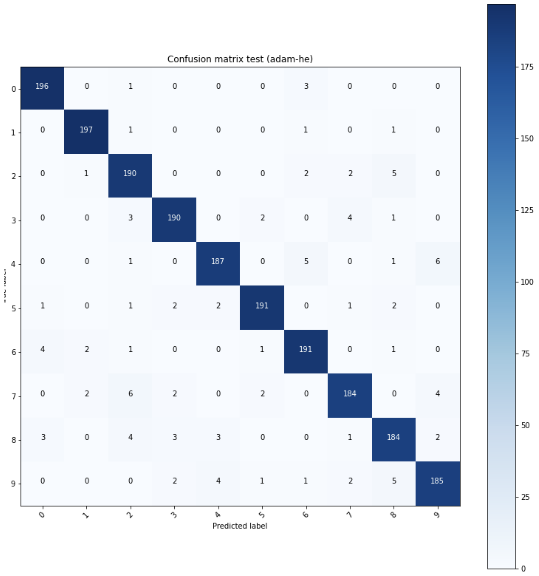
Loss vs Epoch

Train	Test	Combined
		

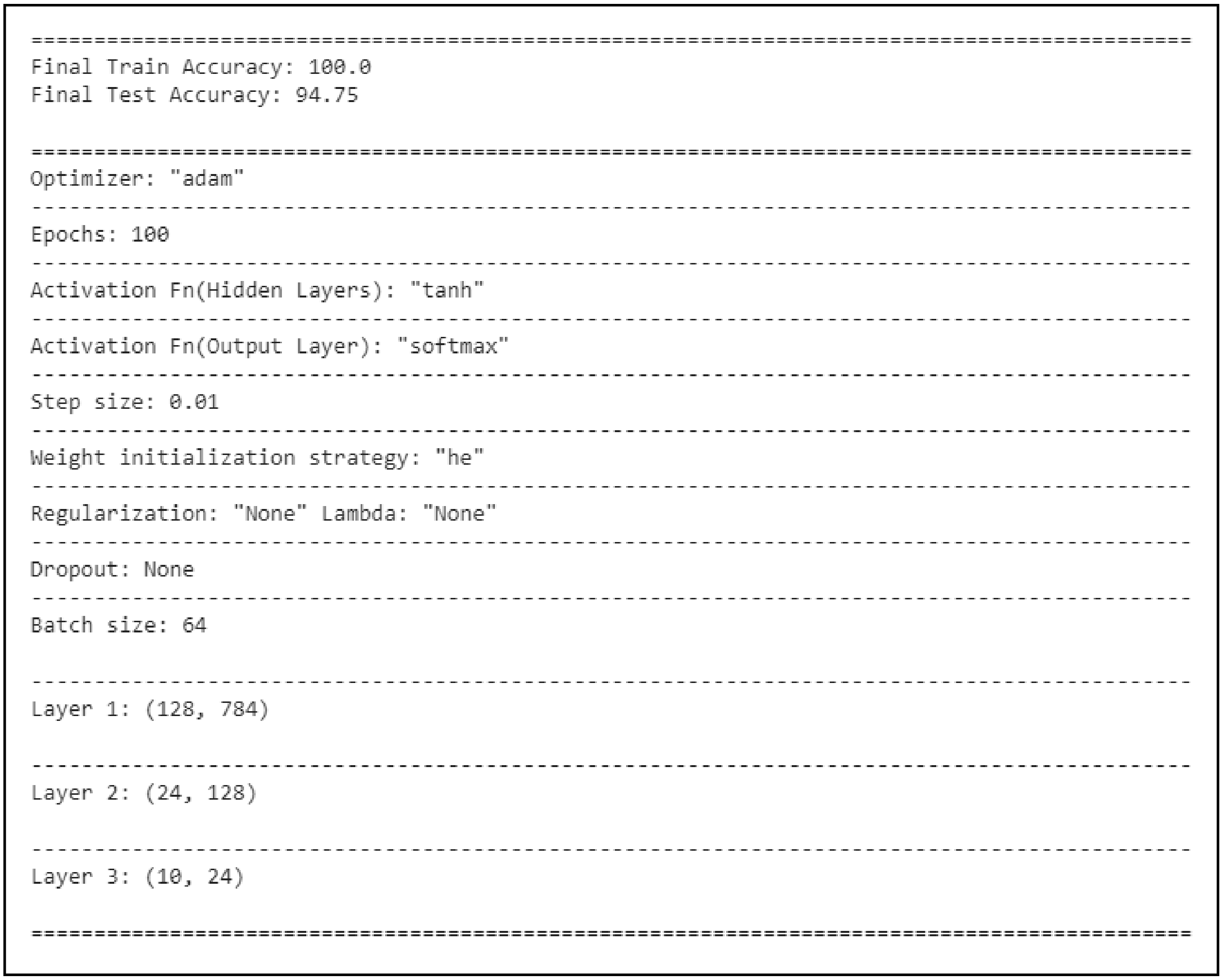
ROC curves (Val set)



Confusion Matrix of Validation set

Train	Test
	

Observation

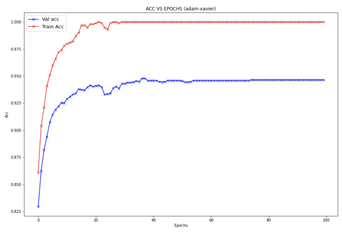
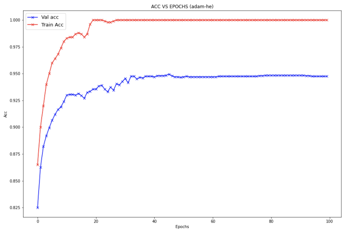
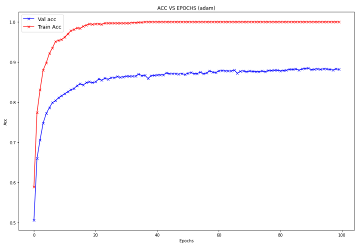


```
============================================================================================
Final Train Accuracy: 100.0
Final Test Accuracy: 94.75

============================================================================================
Optimizer: "adam"
--------------------------------------------------------------------------------------------
Epochs: 100
--------------------------------------------------------------------------------------------
Activation Fn(Hidden Layers): "tanh"
--------------------------------------------------------------------------------------------
Activation Fn(Output Layer): "softmax"
--------------------------------------------------------------------------------------------
Step size: 0.01
--------------------------------------------------------------------------------------------
Weight initialization strategy: "he"
--------------------------------------------------------------------------------------------
Regularization: "None" Lambda: "None"
--------------------------------------------------------------------------------------------
Dropout: None
--------------------------------------------------------------------------------------------
Batch size: 64

--------------------------------------------------------------------------------------------
Layer 1: (128, 784)

--------------------------------------------------------------------------------------------
Layer 2: (24, 128)

--------------------------------------------------------------------------------------------
Layer 3: (10, 24)

============================================================================================
```

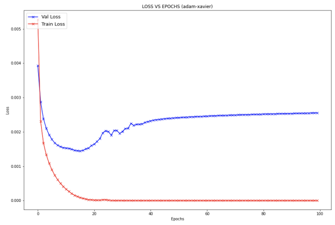
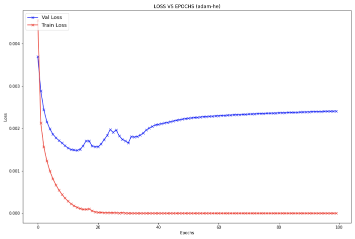
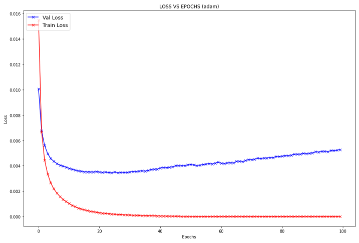
- ***Comparison of Random Initialization , Xavier , He***

```
Network = [784,128,24,10]
Learning Rate = 0.01
Epochs = 100
Activation = Tanh
Optimizer = Adam
Regularization = None
Dropout = None
Initialization = Variable
```

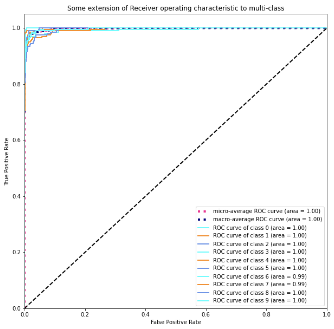
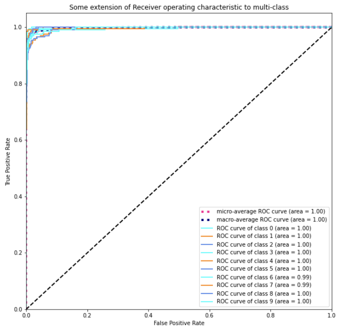
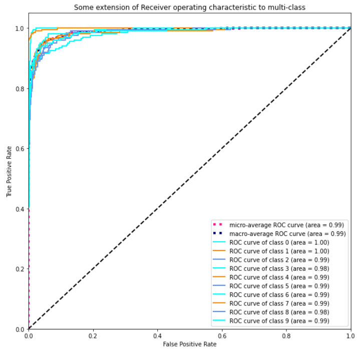
Accuracy vs Epoch

Xavier	He	Random
		

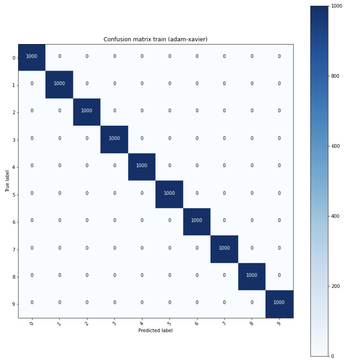
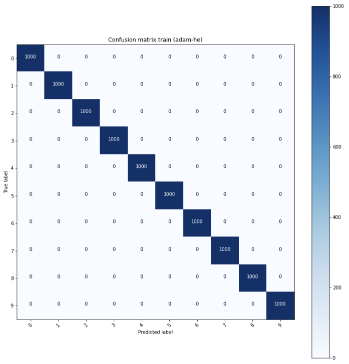
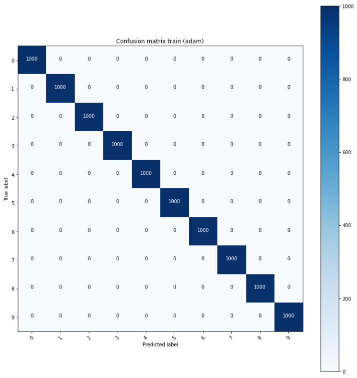
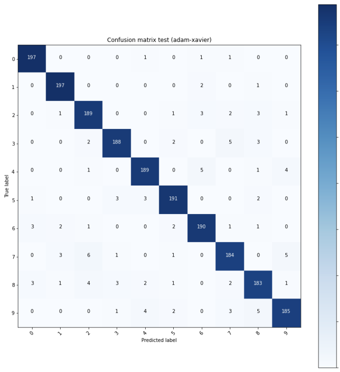
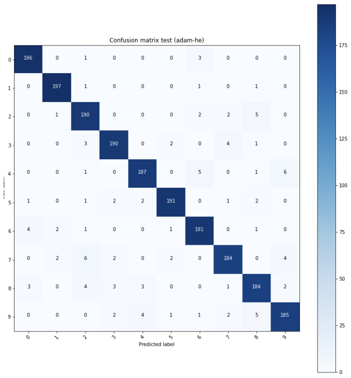
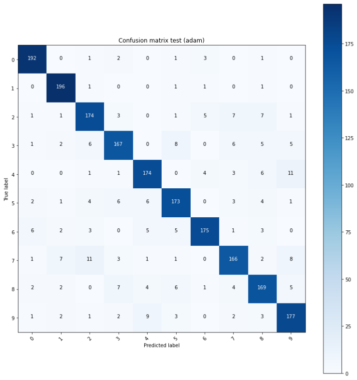
Loss vs Epoch

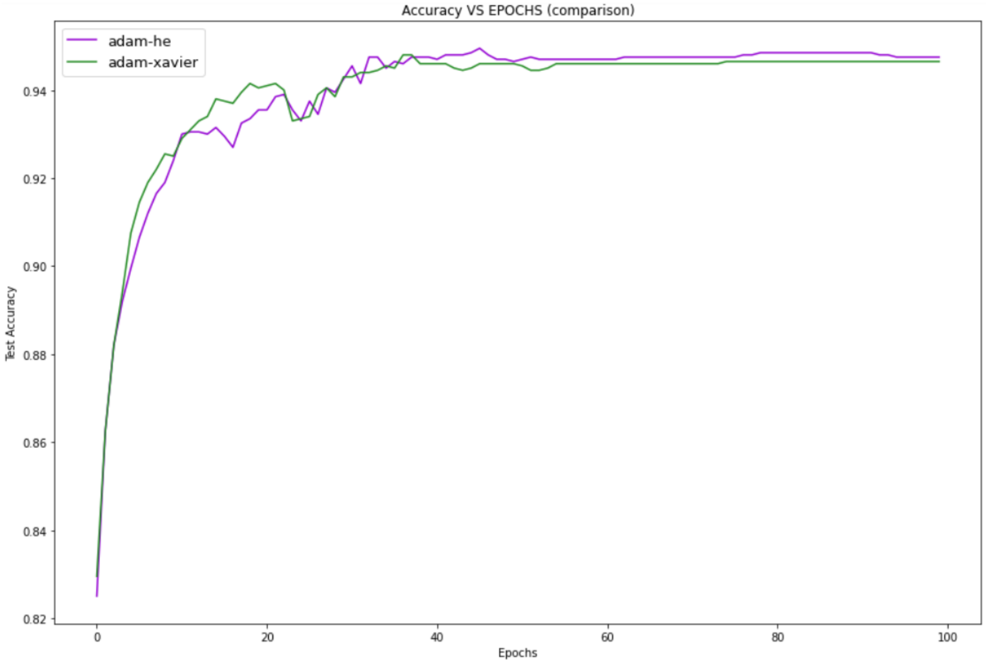
Xavier	He	Random
		

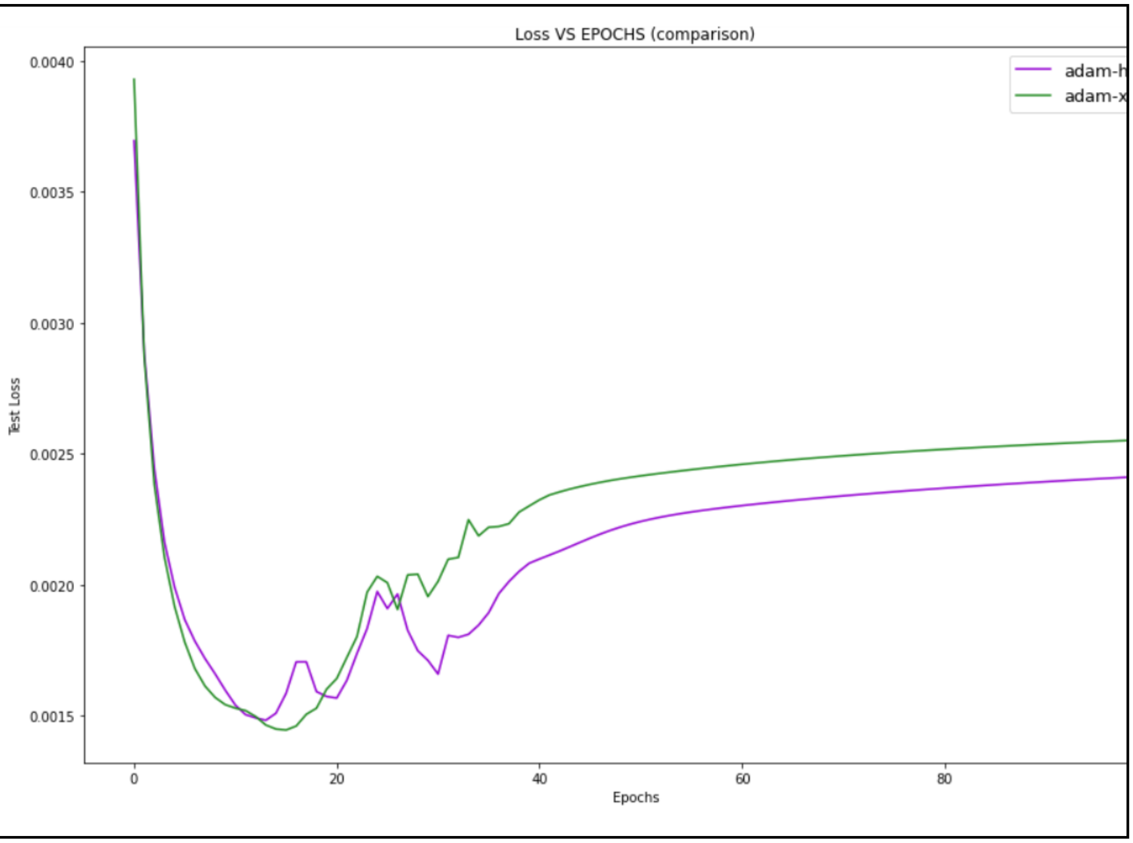
ROC curves

Xavier	He	Random
Test		
		

Confusion Matrix of Validation set

Xavier	He	Random
Train		
		
Test		
		

1. 



2.

Observations :-

From the plots and analysis above the following observations are made.

- All the models using Adam whatever be the initialization are , are overfitting after approximately 20 epochs and loss is increasing.
- So for analysis we consider the first 20 epochs.
- Random Initialization initialises the weights randomly and thus may not get a good start for training, thus with random initialization the accuracy for first epoch is approximately 50 percent while for Xavier and He it is above 80%. With random weights we may not be close to minima also.
- For Xavier and He, both give good start to learning and achieve a good accuracy from the first epoch only that is above 80%. This may happen because they may begin reducing loss close to minima.
- Also we can observe that at 20th epoch, Xavier outperforms He and thus we have chosen Xavier initialization.
- Also it is proved theoretically that Xavier out performs He when the activation function is tanh.

1.5 Best Configuration form Part-2 (b)

To be used in later parts

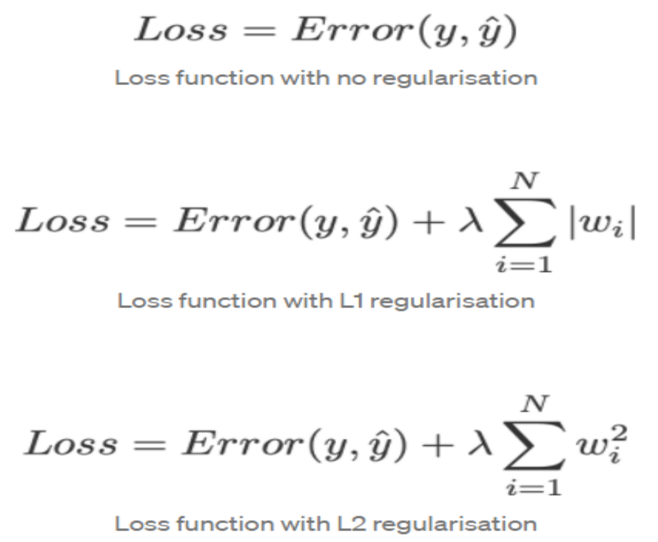
```
Network = [784,128,24,10]
Learning Rate = 0.01
Epochs = 100
Activation = Tanh
Initialization = ‘Xavier’
Regularization = None
Dropout = ‘None’
```

Note :- The model though overfitting cannot be said as best configuration at 100 epochs but it may be helpful in doing a thorough analysis for the different types of regularization techniques in the question part 2

2. Regularization:

Why Regularization ?

Overfitting is a phenomenon that occurs when a model is tailored to a particular dataset and is unable to generalise to other datasets. This usually happens in complex models, like deep neural networks. **Regularization** is a technique used for tuning the function by adding an additional penalty term in the error function. The additional term controls the excessively fluctuating function such that the coefficients don't take extreme values. This in order prevents overfitting. Below is a brief explanation on L1 and L2 regularisation or L1/L2 Norm although it can be of any order and generally known as p-norm.



$$Loss = Error(y, \hat{y})$$

Loss function with no regularisation

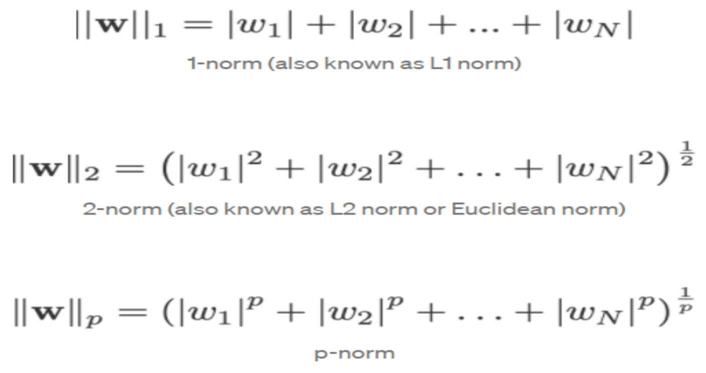
$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Loss function with L1 regularisation

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

Loss function with L2 regularisation

The updated Loss function



$$||\mathbf{w}||_1 = |w_1| + |w_2| + ... + |w_N|$$

1-norm (also known as L1 norm)

$$||\mathbf{w}||_2 = \left(|w_1|^2 + |w_2|^2 + \ldots + |w_N|^2\right)^{\frac{1}{2}}$$

2-norm (also known as L2 norm or Euclidean norm)

$$||\mathbf{w}||_p = \left(|w_1|^p + |w_2|^p + \ldots + |w_N|^p\right)^{\frac{1}{p}}$$

p-norm

Fig. displays what gets multiplied with the lambda term in the error function

- **L1 Regularization**

L1 regularisation or L1 norm is a type of regularisation where we try to stop the overfitting of the model by adding the magnitude of the weights which effectively penalize the larger weights. L1 norm is usually used for feature selection as it can yield sparse models which describe the important features by eliminating some of the coefficients.

The Loss function and weight update is given as

$$L_1(X, \omega) = L(X, \omega) + \lambda \sum |\omega_i|$$

$$\omega_i \leftarrow \omega_i - \eta \cdot [\partial L(X, \omega)/\partial \omega_i + \lambda \cdot sgn(\omega_i))]$$

This variant pushes the parameters towards smaller values, even completely canceling the influence of some input features on the network output, which implies an automatic **feature selection**. The result is a better generalization, but only to a certain extent (the choice of *λ* becomes more significant in this case).

This is the regularization applied by the ***Lasso*** regression.

- **L2 Regularization**

The main idea behind this kind of regularization is to decrease the parameters value, which translates into a variance reduction.This technique introduces an extra **penalty** term in the original loss function (***L***), adding the sum of squared parameters (***ω***).

Loss Function Formula

$$L_2(X, \omega) = L(X, \omega) + \lambda \sum \omega_i^2$$

Weight update on each iteration is given by

$$\omega_i \leftarrow \omega_i - \eta \cdot [\partial L(X, \omega) / \partial \omega_i + 2\lambda\omega_i]$$

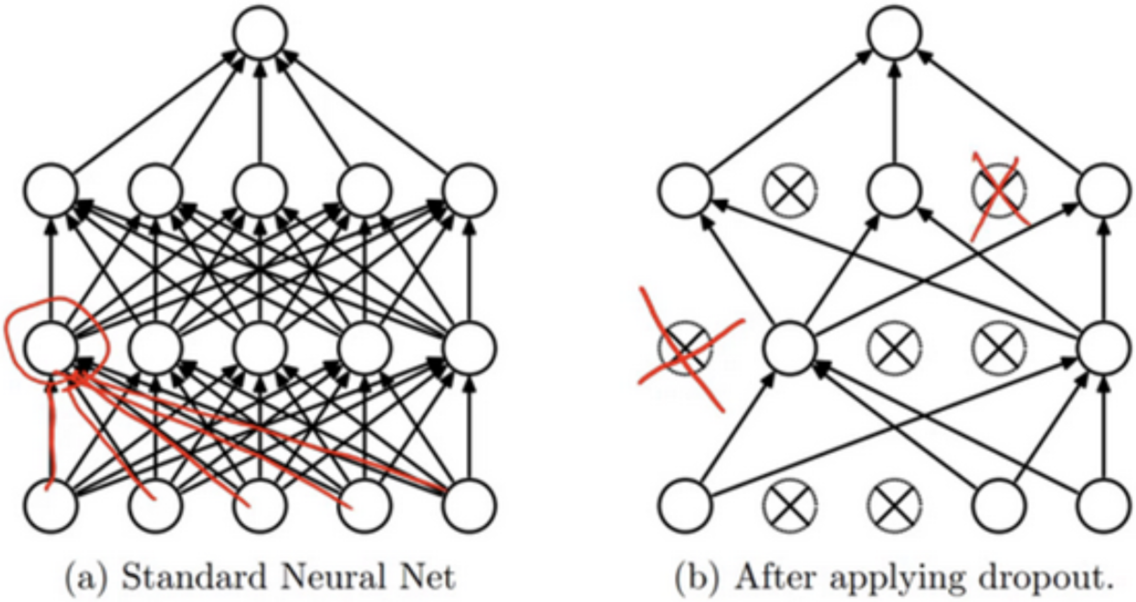
L2 encourages the model to use all of its inputs without leaning too heavily on any one. L1 is generally better if you expect the model to use certain inputs more heavily than others. L2 is more general-purpose and shows up more often as a result.

- **Dropout**

The procedure is simple: for each new input to the network in the training stage, a percentage of the neurons in each hidden layer is randomly deactivated, according to a previously defined **discard probability**. This could be the same for the entire network, or different in each layer.

By applying dropout we avoid that neurons could *memorize* part of the input; that's precisely what happens when the network is overfitting.

Once we have our model fitted, we must somehow **compensate for the** fact that part of the network was inactive in training time. When making predictions, every neuron will be active and therefore there
will be more activations (layer outputs) contributing to the network output, increasing its range. One easy solution is to multiply all parameters by its probability of not being discarded. Other solution to it is Inverted Dropout.



Inverted Dropout

Inverted dropout is a variant of the original dropout technique developed by Hinton et al.

Just like traditional dropout, inverted dropout randomly keeps some weights and sets others to zero. This is known as the “keep probability”

The one difference is that, during the training of a neural network, inverted dropout scales the activations by the inverse of the keep probability $q=1-p$

This prevents network’s activations from getting too large, and does not require any changes to the network during evaluation.

In contrast, traditional dropout requires scaling to be implemented during the test phase.

3.3 L1 - Methodology, Plots and Observations

<u>L1 Regularization :-</u>

Methodology

The loss item is updated to display the loss calculated using L1. Also the gradient from L1 is passed to the weight update step.

```
   def get_loss_item(self,log_p,labels,batch_size):
   grad_loss = -1*np.sum(np.multiply(labels
,np.log(log_p+self._eps)),axis=1)
   regularisation_loss = 0
   if self._regularization == 'l2':
     for layer in self._weights:
       regularisation_loss += np.square(layer).sum()
     regularisation_loss *= self._lambd/2
   elif self._regularization == 'l1':
     for layer in self._weights:
       regularisation_loss += np.abs(layer).sum()
     regularisation_loss *= self._lambd
   avg_loss = (np.sum(grad_loss)+ regularisation_loss)*
1/self._batch_size
   return avg_loss
```

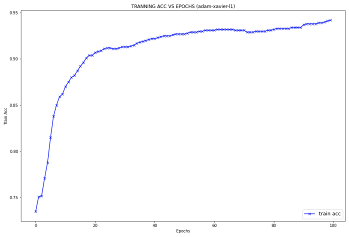
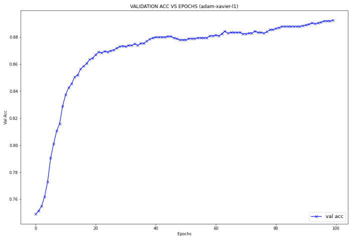
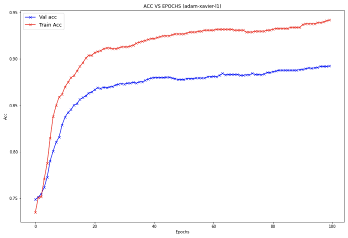
```
for del_,inp in zip(cur_delta, self._y[i]):
       if self._regularization == 'l2':
         self._dw[i] += np.matmul(np.expand_dims(del_,axis=1),
np.expand_dims(inp,axis=0)) +
self._lambd*self._weights[i]/self._batch_size
       elif self._regularization == 'l1':
         self._dw[i] += (np.matmul(np.expand_dims(del_,axis=1),
np.expand_dims(inp,axis=0)) + np.where(self._weights[i]>0,
self._lambd/self._batch_size, -self._lambd/self._batch_size))
       else:
         self._dw[i] += np.matmul(np.expand_dims(del_,axis=1),
np.expand_dims(inp,axis=0))
```

Plots

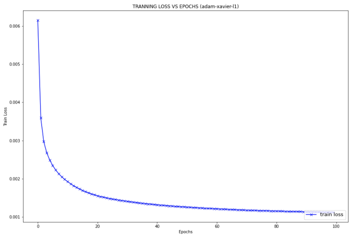
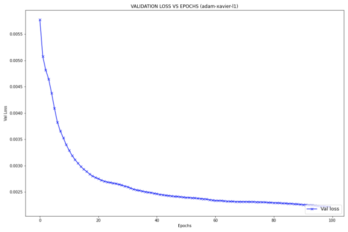
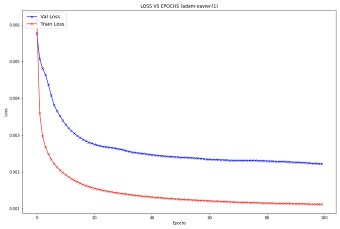
It can be clearly observed from the graph that the model is not overfitting as both training and testing loss are decreasing together and similarly the accuracies are increasing. So L1 regularization is definitely working here, where without it model was overfitting earlier at 20th epoch, Here we can clearly see that there is no overfitting happening. Both training and testing accuracies are close by in a range of 5% and this proves that model is able to generalise well.

A grid search was applied to find the optimal value of lambda among [0.1,0.01,0.001] and it was observed that this model required less regularization effect and so 0.001 works.

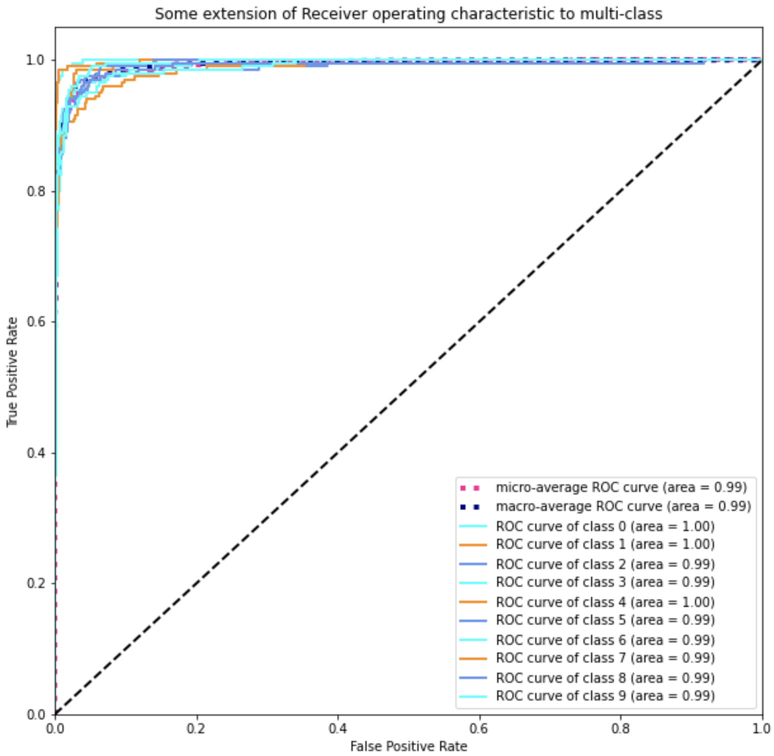
Accuracy vs Epoch

Train	Test	Combined
		

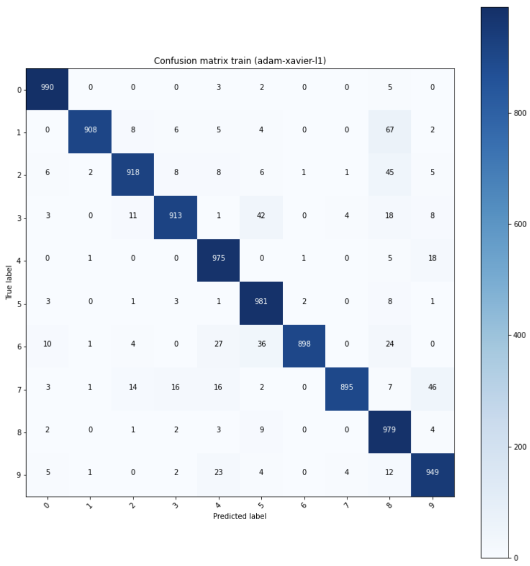
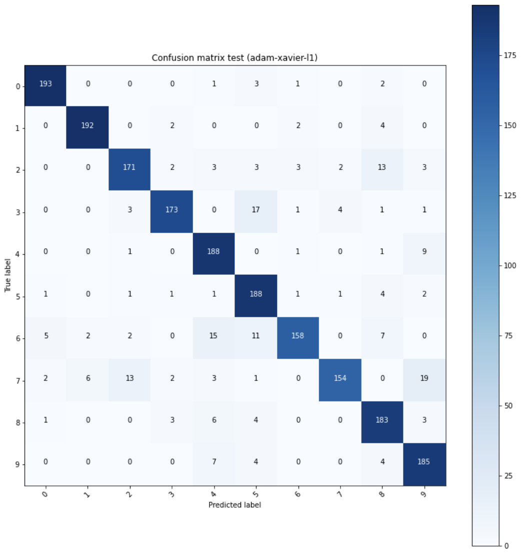
Loss vs Epoch

Train	Test	Combined
		

ROC curves (Val set)



Confusion Matrix of Validation set

Train	Test
	

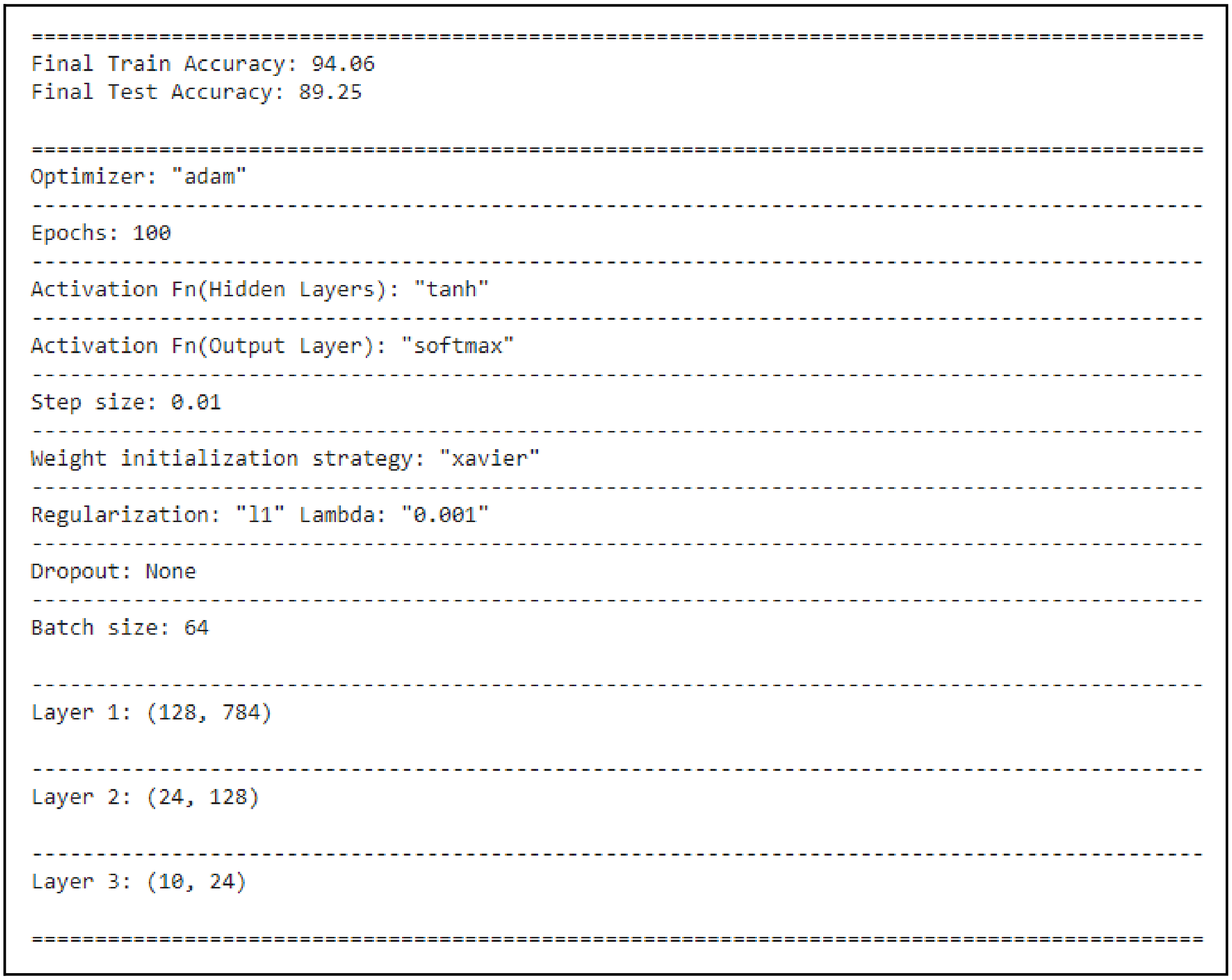
Confusion matrix train (adam-xavier-l1)

	0	1	2	3	4	5	6	7	8	9
0	990	0	0	0	3	2	0	0	5	0
1	0	908	8	6	5	4	0	0	67	2
2	6	2	918	8	8	6	1	1	45	5
3	3	0	11	913	1	42	0	4	18	8
4	0	1	0	0	975	0	1	0	5	18
5	3	0	1	3	1	981	2	0	8	1
6	10	1	4	0	27	36	898	0	24	0
7	3	1	14	16	16	2	0	895	7	46
8	2	0	1	2	3	9	0	0	979	4
9	5	1	0	2	23	4	0	4	12	949

Confusion matrix test (adam-xavier-l1)

	0	1	2	3	4	5	6	7	8	9
0	193	0	0	0	1	3	1	0	2	0
1	0	192	0	2	0	0	2	0	4	0
2	0	0	171	2	3	3	3	2	13	3
3	0	0	3	173	0	17	1	4	1	1
4	0	0	1	0	188	0	1	0	1	9
5	1	0	1	1	1	188	1	1	4	2
6	5	2	2	0	15	11	158	0	7	0
7	2	6	13	2	3	1	0	154	0	19
8	1	0	0	3	6	4	0	0	183	3
9	0	0	0	0	7	4	0	0	4	185

Observation



```
============================================================================================
Final Train Accuracy: 94.06
Final Test Accuracy: 89.25

============================================================================================
Optimizer: "adam"
--------------------------------------------------------------------------------------------
Epochs: 100
--------------------------------------------------------------------------------------------
Activation Fn(Hidden Layers): "tanh"
--------------------------------------------------------------------------------------------
Activation Fn(Output Layer): "softmax"
--------------------------------------------------------------------------------------------
Step size: 0.01
--------------------------------------------------------------------------------------------
Weight initialization strategy: "xavier"
--------------------------------------------------------------------------------------------
Regularization: "l1" Lambda: "0.001"
--------------------------------------------------------------------------------------------
Dropout: None
--------------------------------------------------------------------------------------------
Batch size: 64

--------------------------------------------------------------------------------------------
Layer 1: (128, 784)

--------------------------------------------------------------------------------------------
Layer 2: (24, 128)

--------------------------------------------------------------------------------------------
Layer 3: (10, 24)

============================================================================================
```

3.2 L2 - Methodology, Plots and Observations

L2 Regularization :-

Methodology

The loss item is updated to display the loss calculated using L1. Also the gradient from L1 is passed to the weight update step.

```
  def get_loss_item(self,log_p,labels,batch_size):
  grad_loss = -1*np.sum(np.multiply(labels
,np.log(log_p+self._eps)),axis=1)
  regularisation_loss = 0
  if self._regularization == 'l2':
   for layer in self._weights:
    regularisation_loss += np.square(layer).sum()
   regularisation_loss *= self._lambd/2
  elif self._regularization == 'l1':
   for layer in self._weights:
    regularisation_loss += np.abs(layer).sum()
   regularisation_loss *= self._lambd
  avg_loss = (np.sum(grad_loss)+ regularisation_loss)*
1/self._batch_size
  return avg_loss
```

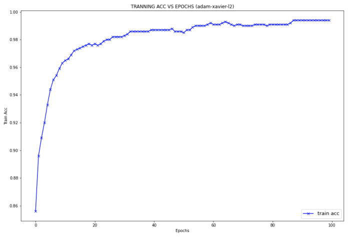
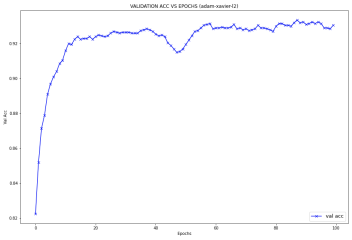
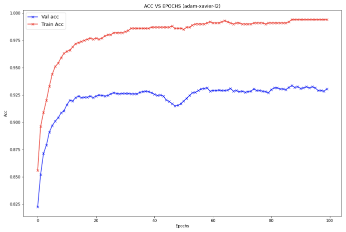
```
for del_,inp in zip(cur_delta, self._y[i]):
       if self._regularization == 'l2':
         self._dw[i] += np.matmul(np.expand_dims(del_,axis=1),
np.expand_dims(inp,axis=0)) +
self._lambd*self._weights[i]/self._batch_size
       elif self._regularization == 'l1':
         self._dw[i] += (np.matmul(np.expand_dims(del_,axis=1),
np.expand_dims(inp,axis=0)) + np.where(self._weights[i]>0,
self._lambd/self._batch_size, -self._lambd/self._batch_size))
       else:
         self._dw[i] += np.matmul(np.expand_dims(del_,axis=1),
np.expand_dims(inp,axis=0))
```

Plots

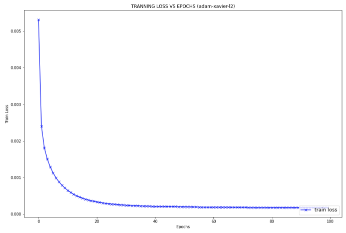
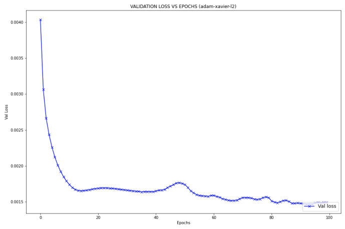
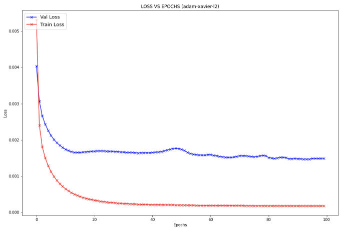
It can be clearly observed from the graph that the model is not overfitting as both training and testing loss are decreasing together and similarly the accuracies are increasing. So L2 regularization is definitely working here, where without it model was overfitting earlier at 20th epoch, Here we can clearly see that there is no overfitting happening. Both training and testing accuracies are close by in a range of 5% and this proves that model is able to generalise well.

A grid search was applied to find the optimal value of lambda among [0.1,0.01,0.001] and it was observed that this model required less regularization effect and so 0.001 works.

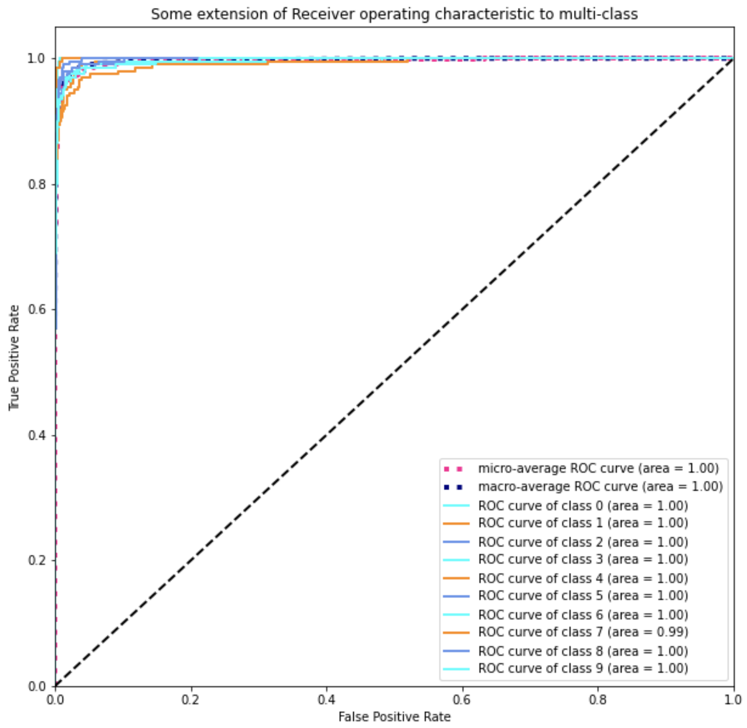
Accuracy vs Epoch

Train	Test	Combined
		

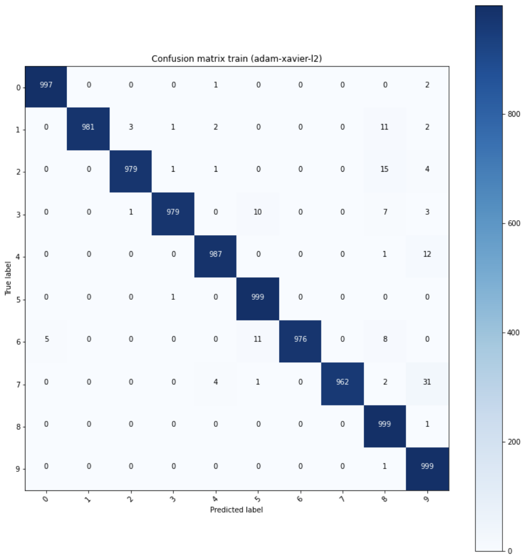
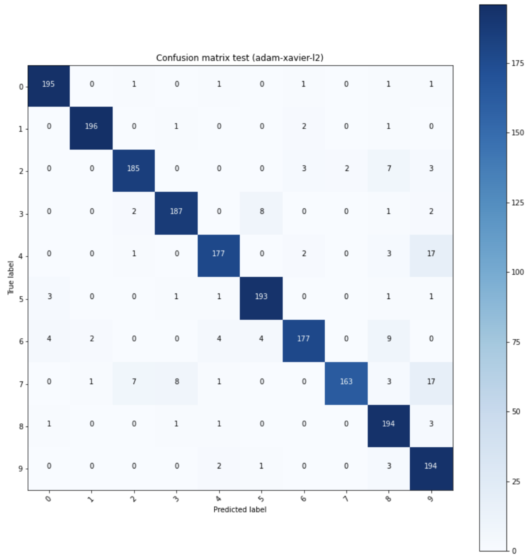
Loss vs Epoch

Train	Test	Combined
		

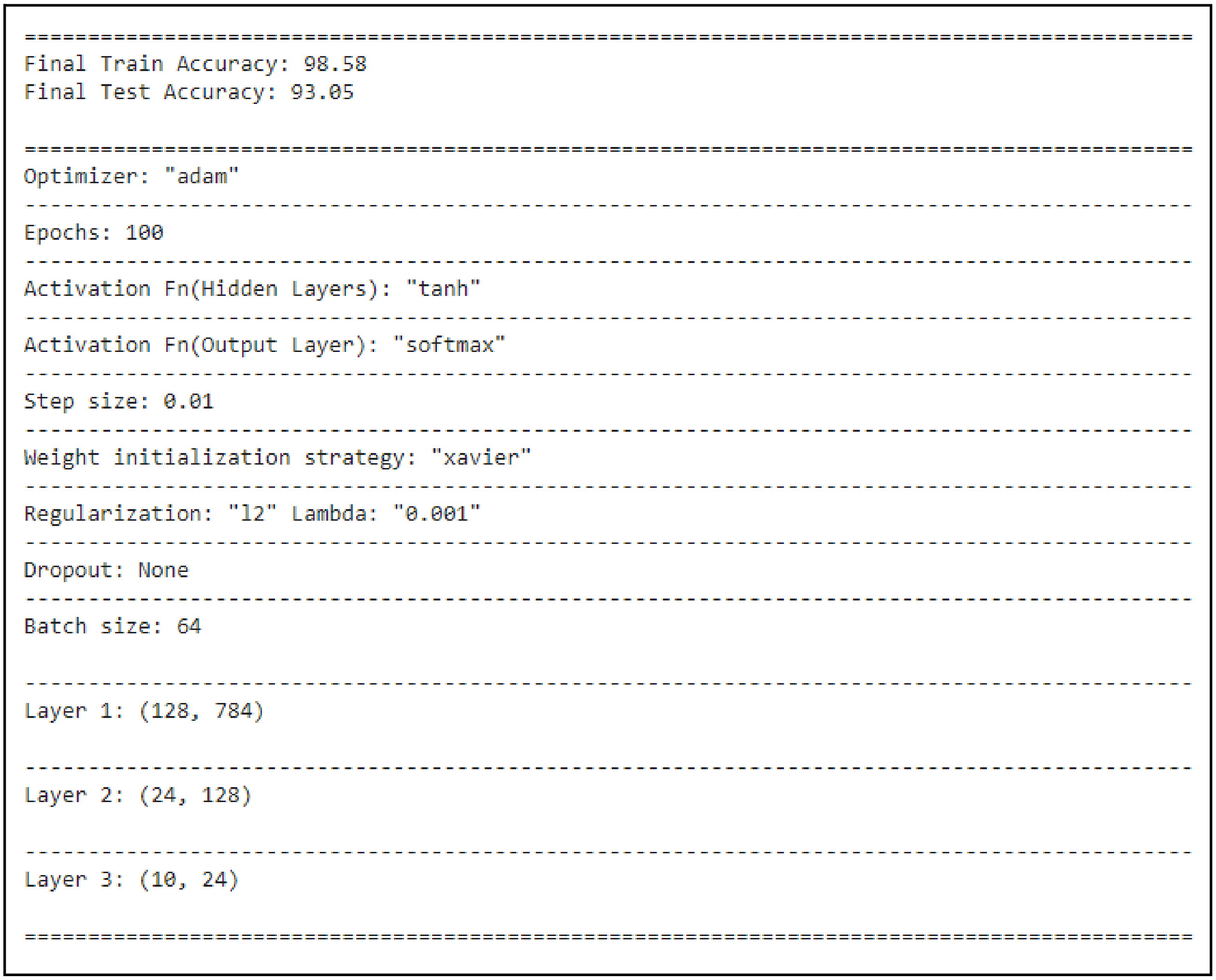
ROC curves (Val set)



Confusion Matrix of Validation set

<table>
<tr><th>Train</th><th>Test</th></tr>
<tr><td>
</td>
<td>
</td></tr>
</table>

Observations



```
============================================================================================
Final Train Accuracy: 98.58
Final Test Accuracy: 93.05

============================================================================================
Optimizer: "adam"
--------------------------------------------------------------------------------------------
Epochs: 100
--------------------------------------------------------------------------------------------
Activation Fn(Hidden Layers): "tanh"
--------------------------------------------------------------------------------------------
Activation Fn(Output Layer): "softmax"
--------------------------------------------------------------------------------------------
Step size: 0.01
--------------------------------------------------------------------------------------------
Weight initialization strategy: "xavier"
--------------------------------------------------------------------------------------------
Regularization: "l2" Lambda: "0.001"
--------------------------------------------------------------------------------------------
Dropout: None
--------------------------------------------------------------------------------------------
Batch size: 64

--------------------------------------------------------------------------------------------
Layer 1: (128, 784)

--------------------------------------------------------------------------------------------
Layer 2: (24, 128)

--------------------------------------------------------------------------------------------
Layer 3: (10, 24)

============================================================================================
```

3.1 Dropout- Methodology, Plots and Observations

Dropout :-

Methodology

```
 def drop_out_matrices(self,layers_dims, m):
   np.random.seed(1)
   self._D = {}
   L = len(layers_dims)

   for l in range(L):
       # initialize the random values for the dropout matrix
       self._D[str(l)] = np.random.rand(1,layers_dims[l])
       # Convert it to 0/1 to shut down neurons corresponding to
each element
       self._D[str(l)] = self._D[str(l)] < self._keep_prob_arr[l]
```

We used Inverted Dropout for applying dropout. From the drop out array sent during initialization a keep probability array is created for each layer. During each batch processing first a marking array D is computed using the above function for each and every layer. It actually is used to track which nodes must be kept and which nodes must be dropped.

During forward propagation each activation output is multiplied by the same mask with the respective layer and scaled using 1/keep_prob[i] i.e keep probability of the respective layer. During Back propagation the respective deltas are multiplied by the same mask respective to each layer to make sure for turned off neurons there is no update happening.

Using this Inverted Dropout technique there are changes only done during the training phase and testing phase is left untouched.
The following changes are made in forward and backward propagation functions with generating marks in fit function.

Forward Propagation

```
def forward_propagate(self):

    temp = self._y[0]
    temp = np.multiply(temp,self._D[str(0)])
    temp = temp * (1/self._keep_prob_arr[0])
    self._y[0] = temp

    for idx, (w_i, b_i) in enumerate(zip(self._weights,self._bias)):
      # with dropout
      z_i = np.dot(temp,w_i.T) + b_i.T
      self._z[idx] = z_i
      if (idx == len(self._weights)-1):
        y_i = self.Mysoftmax(z_i,axis=1)
        y_i = np.multiply(y_i,self._D[str(idx+1)])
        y_i = y_i * (1/self._keep_prob_arr[idx+1])
      else:
        y_i = self._activation(z_i)
        y_i = np.multiply(y_i,self._D[str(idx+1)])
        y_i = y_i * (1/self._keep_prob_arr[idx+1])

      self._y[idx+1] = y_i
      temp = y_i
```

Backward Propagation

```
def back_propagate(self, label):
   for i in reversed(range(len(self._layers)-1)):
     if i == len(self._layers) - 2:
       # Dropout for output layer but internal keep prob = 1
       self._delta[-1] = self._y[-1] - label
       self._delta[-1] = np.multiply(self._delta[-1],
self._D[str(len(self._layers)-1)])
       # self._delta[-1] = self._delta[-1] * (1/
self._keep_prob_arr[len(self._layers)-1])
     else:
       if self._optimizer == 'nesterov':
         self._optimizer_weight[i+1] = self._beta *
self._optimizer_weight[i+1]
         self._optimizer_bias[i+1] = self._beta *
self._optimizer_bias[i+1]
         self._weights[i+1] += self._optimizer_weight[i+1]
       a1 = np.dot(self._delta[i+1], self._weights[i+1])
       b1 = self._activation_derivative(self._y[i+1])
       self._delta[i] = np.multiply(a1,b1)
       # To add the dropout term
       self._delta[i] = np.multiply(self._delta[i], self._D[str(i+1)])

     cur_delta = self._delta[i]/self._batch_size
     self._db[i] = np.expand_dims(np.sum(cur_delta,axis=0),axis=1)
     for del_,inp in zip(cur_delta, self._y[i]):
       if self._regularization == 'l2':
         self._dw[i] += np.matmul(np.expand_dims(del_,axis=1),
np.expand_dims(inp,axis=0)) +
self._lambd*self._weights[i]/self._batch_size
       elif self._regularization == 'l1':
         self._dw[i] += (np.matmul(np.expand_dims(del_,axis=1),
np.expand_dims(inp,axis=0)) + np.where(self._weights[i]>0,
self._lambd/self._batch_size, -self._lambd/self._batch_size))
       else:
         self._dw[i] += np.matmul(np.expand_dims(del_,axis=1),
np.expand_dims(inp,axis=0))
```

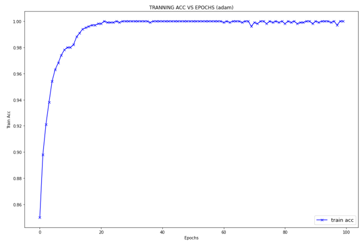
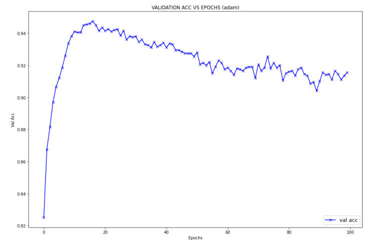
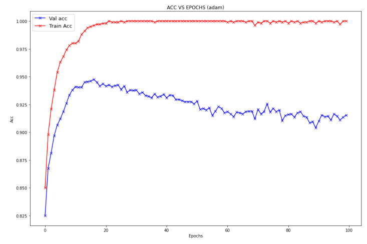
Plots

By changing the dropout probability of different hidden layers for the same configuration as Part 2 , Q1 we found the following observations. A thorough analysis of the accuracy by different dropout probabilities is shown in the later part.

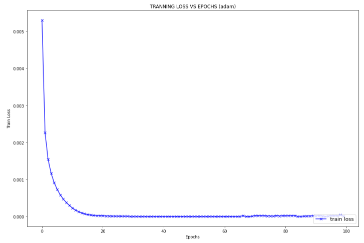
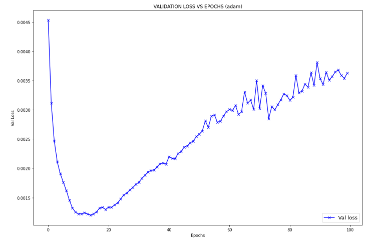
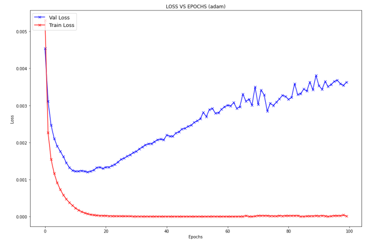
It was noticed that keeping the drop probability high model was underfitting so We preferred to keep drop probability low. (Explained Later part)

```
Network = [784,128,24,10]
Learning Rate = 0.01
Epochs = 100
Activation = Tanh
Initialization = Xavier
Dropout (layers) = [0,0.1,0.1,0]
Optimizer = Adam
```

Accuracy vs Epoch

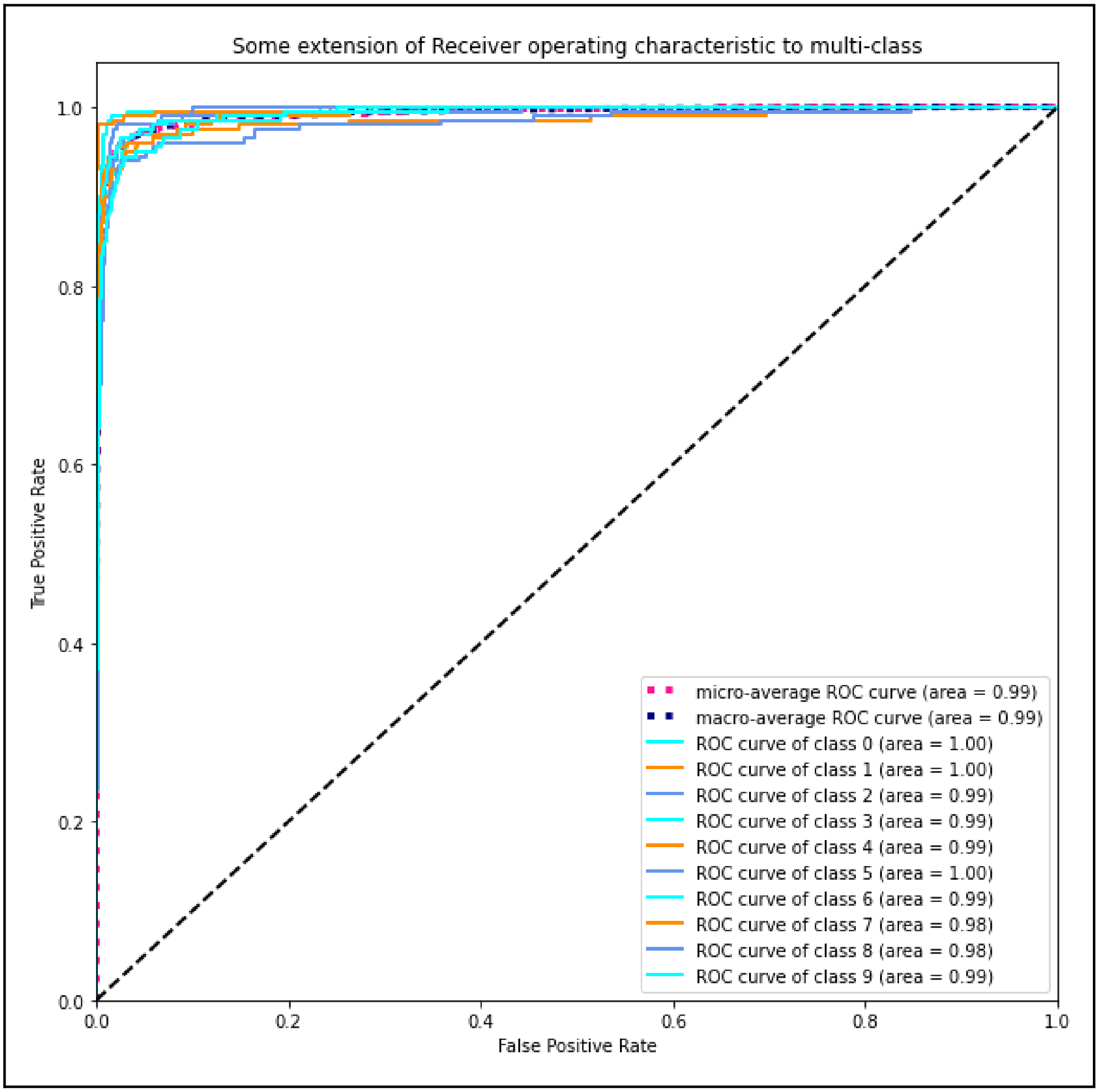
Train	Test	Combined
		

Loss vs Epoch

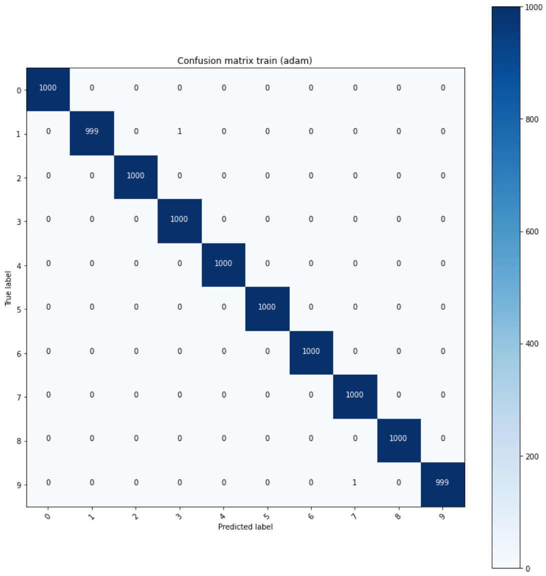
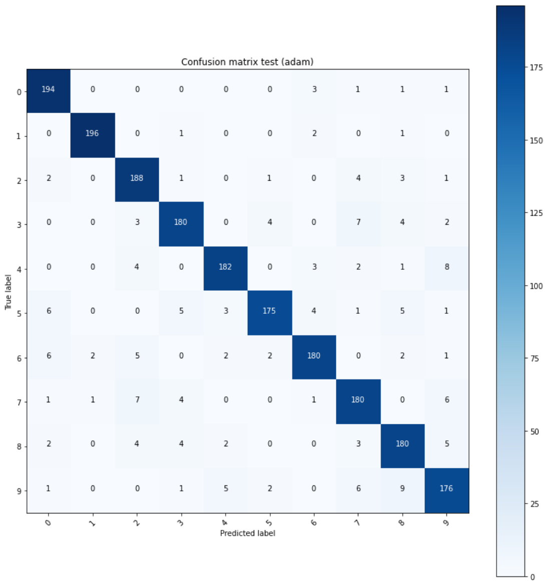
Train	Test	Combined
		

From these plots it can be seen that with more number of epochs along with dropout model starts getting simpler and accuracies start decreasing. So the right time to stop will be epoch-10 where maximum accuracy reached is 94%

ROC curves (Val set)

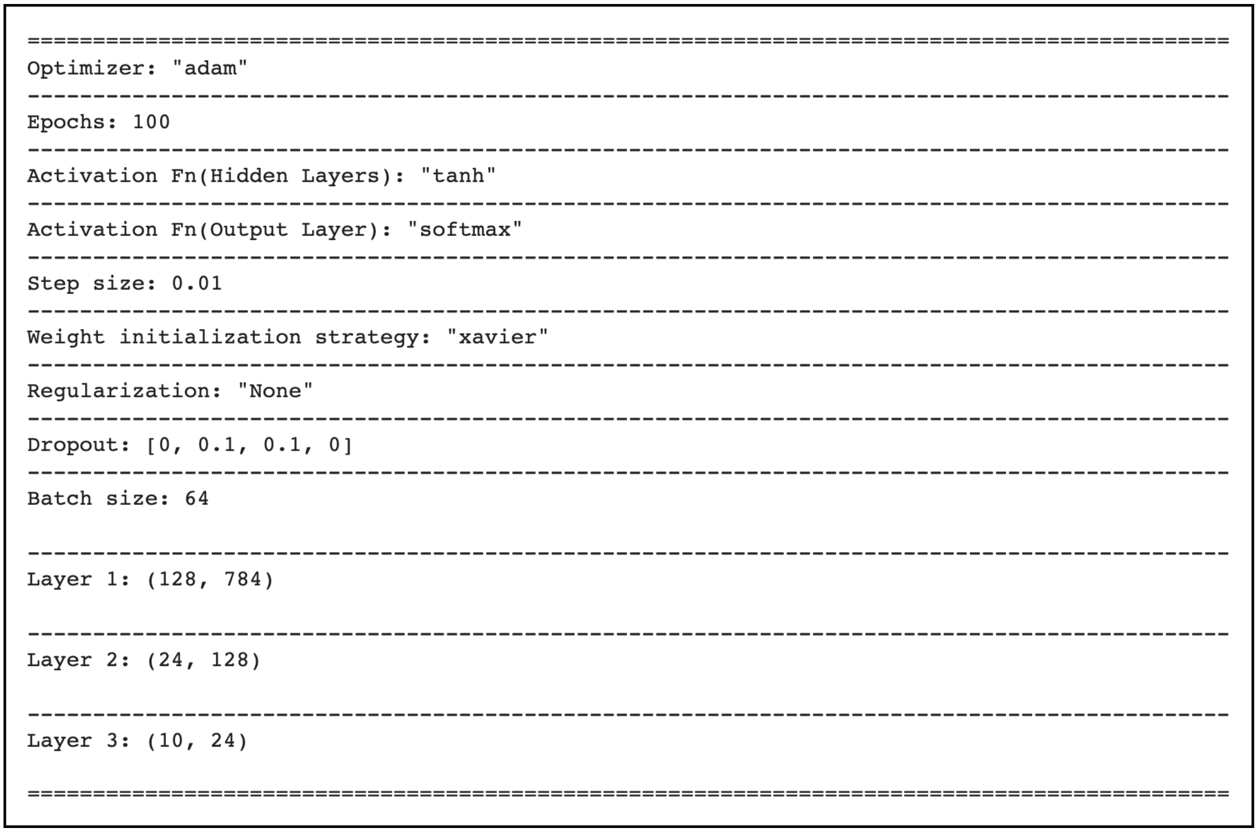


Confusion Matrix of Validation set

Train	Test
	

Observations

TESTING ACCURACY
91.55
TRAINING ACCURACY
99.98

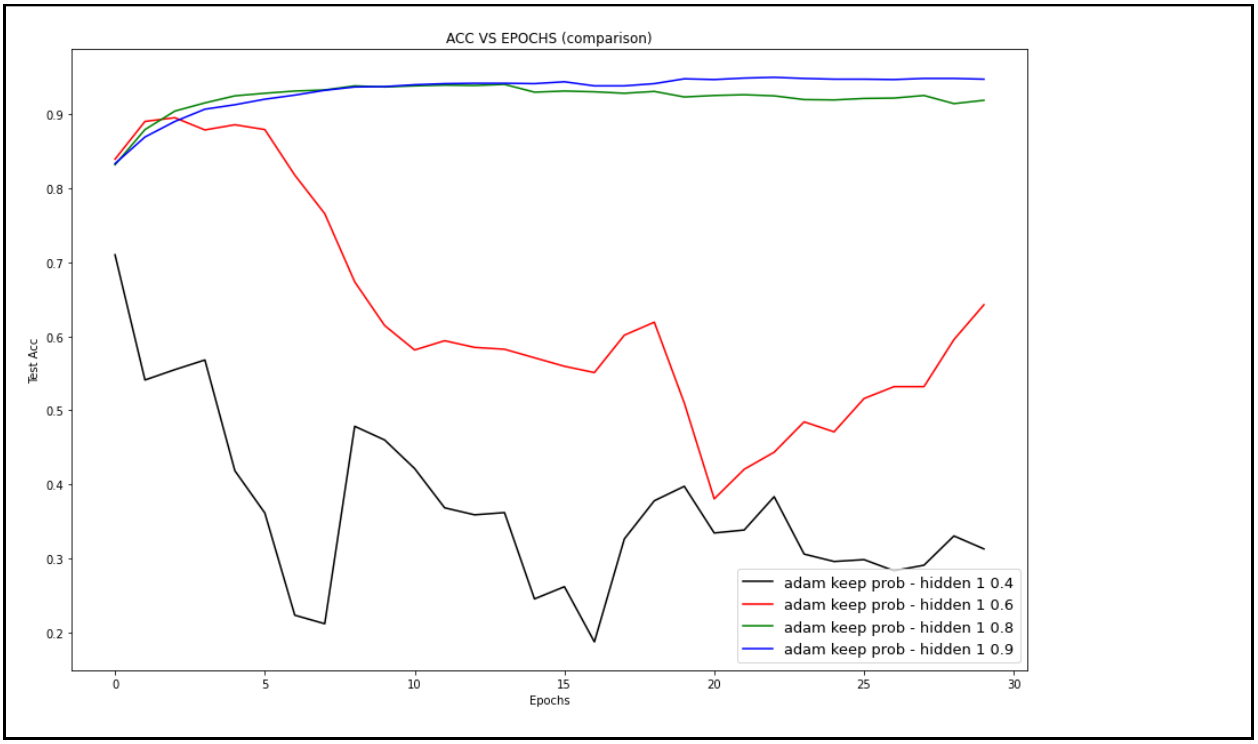


```
============================================================================================
Optimizer: "adam"
--------------------------------------------------------------------------------------------
Epochs: 100
--------------------------------------------------------------------------------------------
Activation Fn(Hidden Layers): "tanh"
--------------------------------------------------------------------------------------------
Activation Fn(Output Layer): "softmax"
--------------------------------------------------------------------------------------------
Step size: 0.01
--------------------------------------------------------------------------------------------
Weight initialization strategy: "xavier"
--------------------------------------------------------------------------------------------
Regularization: "None"
--------------------------------------------------------------------------------------------
Dropout: [0, 0.1, 0.1, 0]
--------------------------------------------------------------------------------------------
Batch size: 64

--------------------------------------------------------------------------------------------
Layer 1: (128, 784)

--------------------------------------------------------------------------------------------
Layer 2: (24, 128)

--------------------------------------------------------------------------------------------
Layer 3: (10, 24)

============================================================================================
```

Analysis of Effect of Dropout over changing Dropout Probability

```
Network = [784,128,24,10]
Learning Rate = 0.001
Epochs = 30
Activation = Tanh
Initialization = Xavier
Dropout (layers) = [0,variable,0,0]
Optimizer = Adam
```

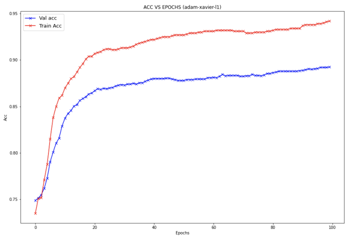
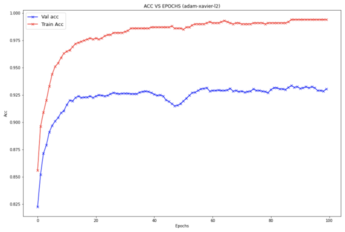
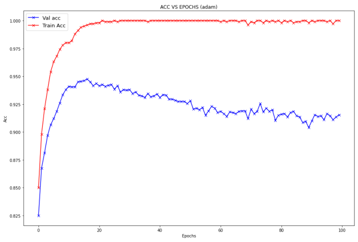
Here for analysis we kept all hyper parameters same and change only the dropout probabilities of first hidden layer containing 128 nodes respectively in this fashion [0.6,0.4,0.2,0.1]



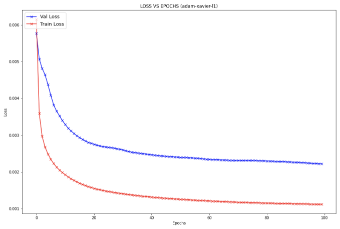
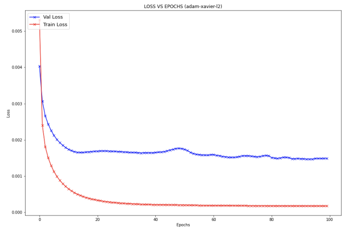
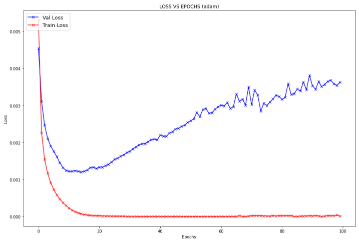
- It can be easily observed from the graph that dropout is actually making the model simpler i.e the more nodes dropped the simpler the model and it may also lead to underfitting if a lot of nodes are dropped.
- From the above graph it can be visualised when only 0.2 percent of nodes are dropped model is able to perform good but as soon as this value goes up to 0.4 the model actually starts underfitting and is not able to give good accuracy.
- From this analysis we chose to keep the dropout probability low so that model does not underfit. Also as hidden layer 2 has only 24 nodes we decided not to drop any nodes from that layer or drop a few only if from that layer.

Analysis of Various Regularization Plots and Observation

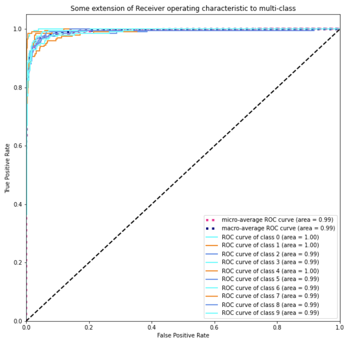
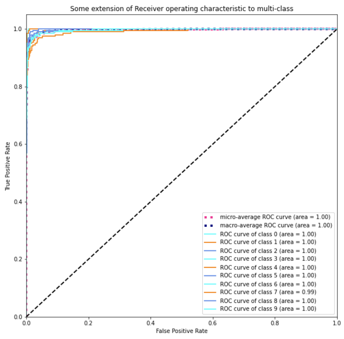
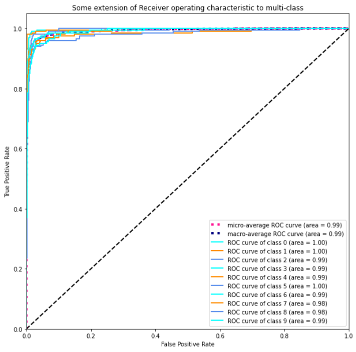
Accuracy vs Epoch

L1	L2	Dropout
		

Loss vs Epoch

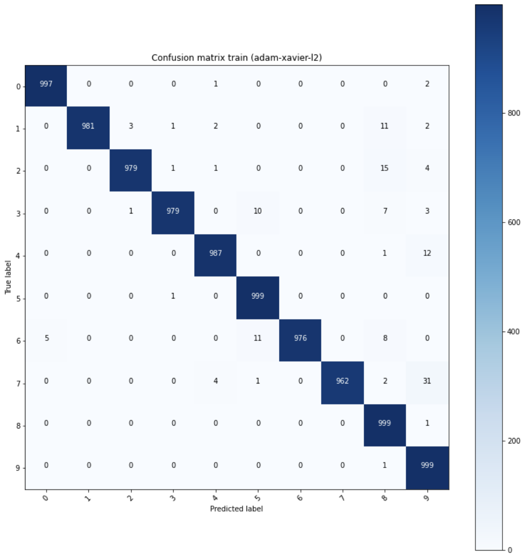
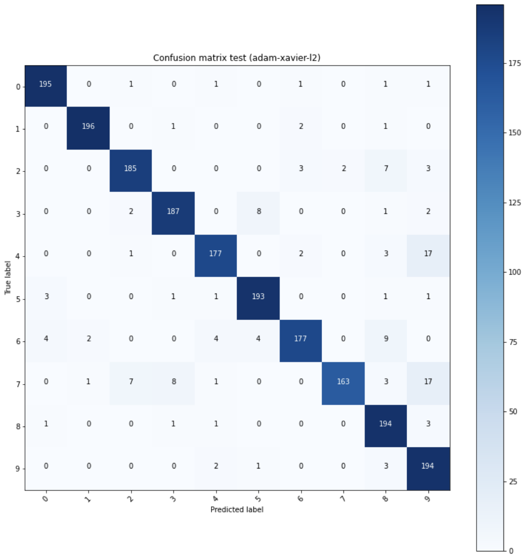
L1	L2	Dropout
		

ROC curves (Val set)

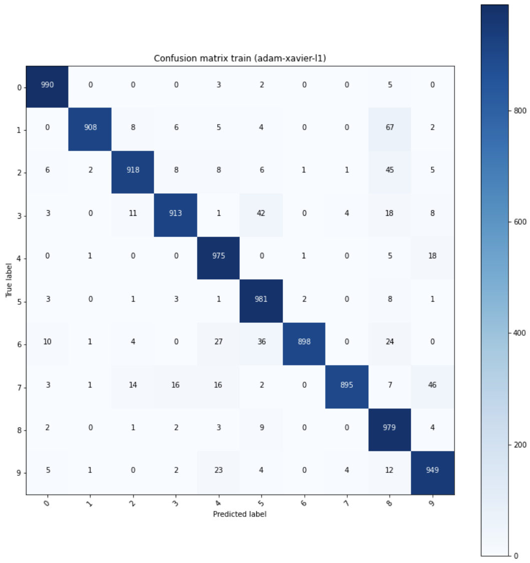
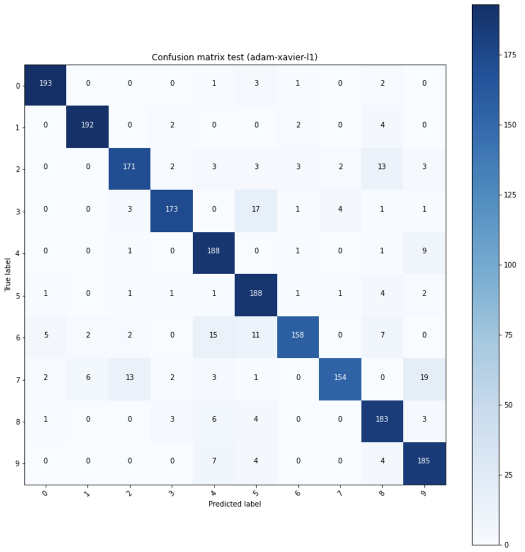
L1	L2	Dropout
		

Confusion Matrix of Validation set

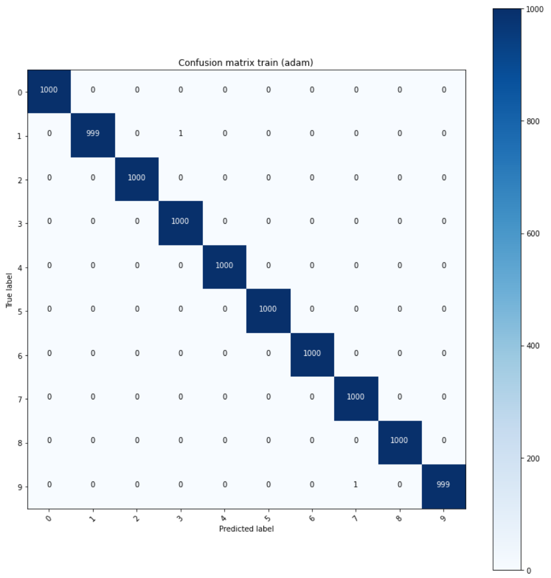
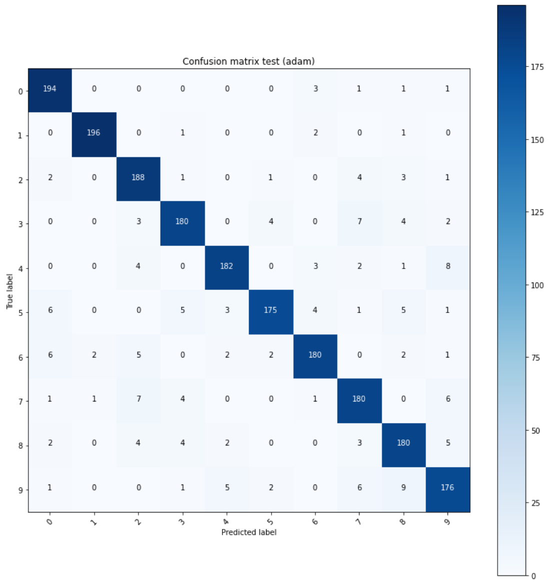
L1

Train	Test
	

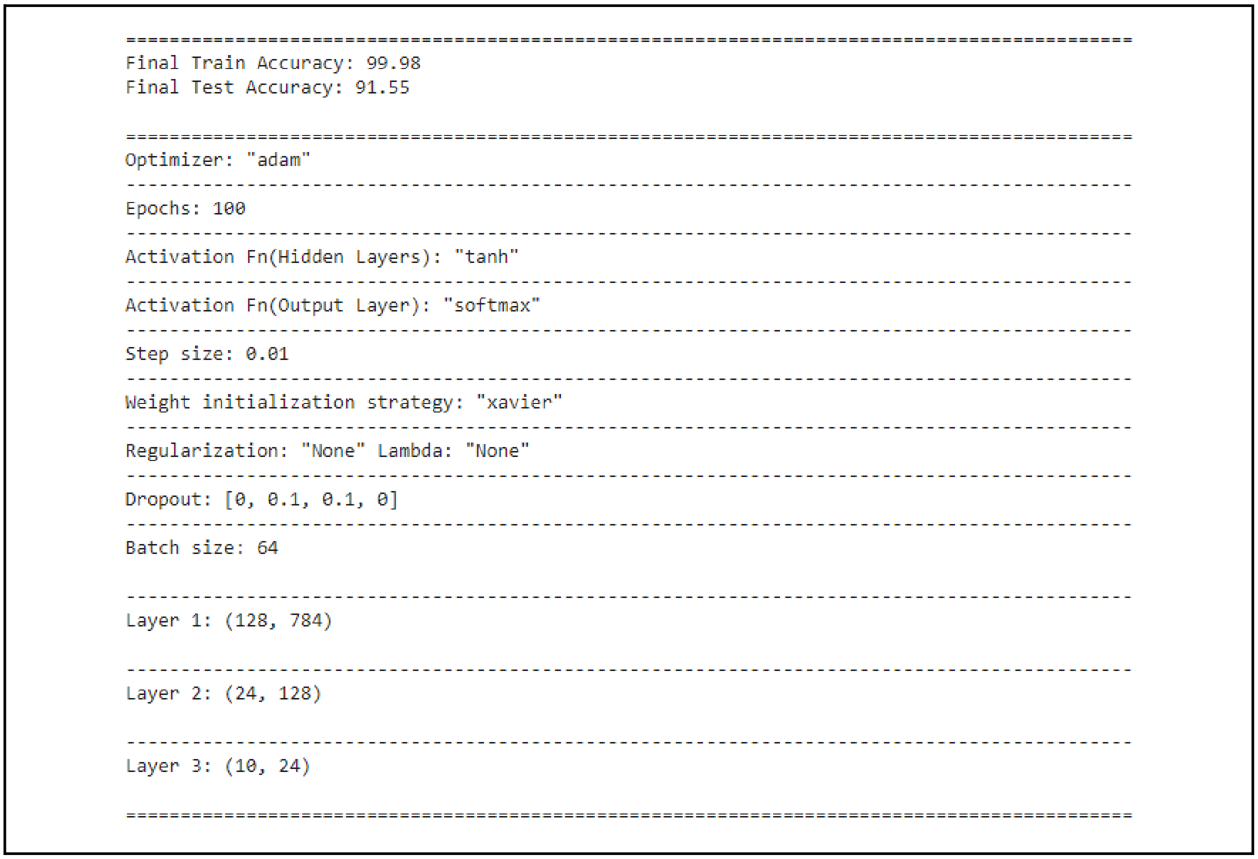
L2

Train	Test
	

Dropout

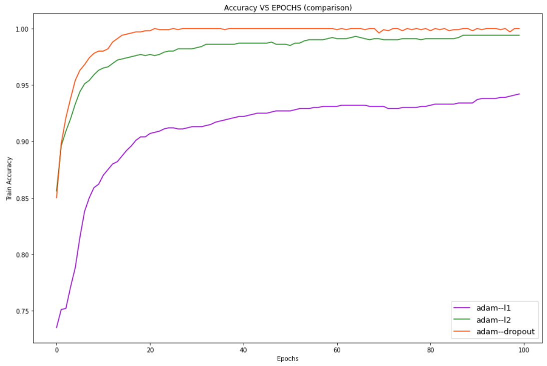
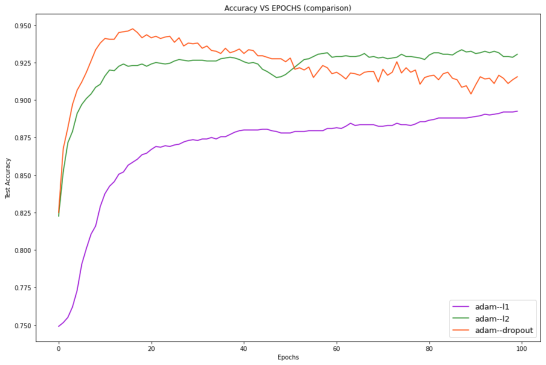
Train	Test
	

Observations

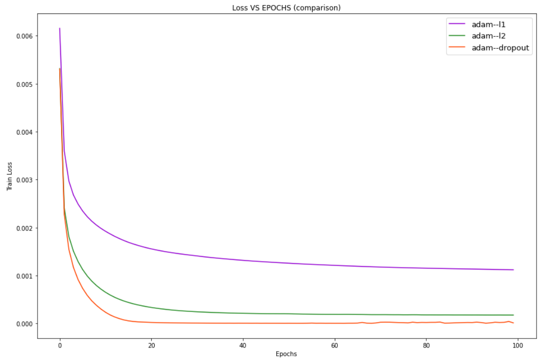
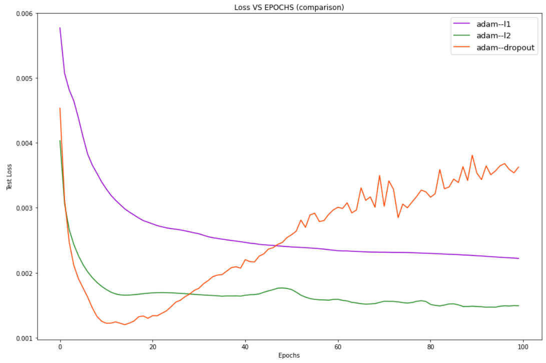


```
==========================================================================================
Final Train Accuracy: 99.98
Final Test Accuracy: 91.55

==========================================================================================
Optimizer: "adam"
------------------------------------------------------------------------------------------
Epochs: 100
------------------------------------------------------------------------------------------
Activation Fn(Hidden Layers): "tanh"
------------------------------------------------------------------------------------------
Activation Fn(Output Layer): "softmax"
------------------------------------------------------------------------------------------
Step size: 0.01
------------------------------------------------------------------------------------------
Weight initialization strategy: "xavier"
------------------------------------------------------------------------------------------
Regularization: "None" Lambda: "None"
------------------------------------------------------------------------------------------
Dropout: [0, 0.1, 0.1, 0]
------------------------------------------------------------------------------------------
Batch size: 64

------------------------------------------------------------------------------------------
Layer 1: (128, 784)

------------------------------------------------------------------------------------------
Layer 2: (24, 128)

------------------------------------------------------------------------------------------
Layer 3: (10, 24)

==========================================================================================
```

Accuracy vs Epoch (L1, L2, Dropout)

Train	Test
	

Loss

Train	Test
	

Observation :-

- From the above plots it is observed that regularization is working to reduce the overfitting problem and L2,Dropout perform equally good while L1 regularization lags behind.
- The reason why L1 is not able to perform as good as other techniques is that L1 tries to reduce the loss and in this process makes weights zero i.e weight matrix become sparse. This affects a lot as many activations become zero and moving across layers some neurons contribute less due to sparse weights.
- In comparison to L1, L2 and Dropout perform better.
- In Dropout it can be observed that as we increase the number of epochs more loss starts increasing for the validation set, this is because we are regularizing too much. But if we keep epocs to 20, dropout outperforms both L1 and L2 in the case of regularization.

2.2 Best Model

BEST OPTIMIZER AND MODEL :-

```
Network = [784,128,24,10]
Learning Rate = 0.01
Epochs = 30
Activation = Tanh
Initialization = Xavier
Dropout (layers) = [0,0.1,0.1,0]
Optimizer = Adam
```

From the above analysis we can observe that dropout has performed the best with an accuracy around 95% and we can obtain this model using early stopping near epoch 20. Because the model architecture is quite simple we are able to obtain a generalized model with a low dropout factor of just 0.1 in the hidden layers.

Contribution of Each Member

1. **Shivam Sharma**

   Implemented Regularization and plots for each along with regularization part of readme

2. **Akanksha Shrimal**

   Implemented Dropout and report

3. **Vaibhav Goswami**

   Implemented Initialization and report.