

Predicting Digits with MNIST

30.08.2018

Group 17

Chinmoy Jyoti Kathar 150101019

Nayanjyoti Kakati 150101041

Chinmoy Kachari 150101020

Pulak Kuli 150101050

Overview

In this assignment we have exercised on various parameters such as number of hidden layers, types of activation functions, number of convolution kernels, size of convolution kernels and have also exercised on over-fitting problem and have cited possible solutions.

We have written a report stating the summary about this assignment with required snapshots of training processes, testing processes and change in learning mechanism as we changed the parameters.

Experiments:

Default: Number of Hidden Layers: 2, Activation: reLU, Dropout: 0.2, regularisation :0

A. Without Conv2D

1. Type of Activation Function: reLU, Sigmoid, tanh
2. Dropout Probability: 0, 0.2, 0.5
3. Reduce Data : 5%, 30% , 100% to test overfitting
4. Number of Hidden layers: 1,2,3,4,5,10
5. Batch size effect on learning rate: 32,128,512
6. Varying the Learning Rate with SGD optimizer: 0.001, 0.01, 0.1

To test and Improve Overfitting: varying number of hidden layers , data 5%

7. Increase Data set size.
8. Use Early Stop method.
9. Increase Dropout.

Default: Number of conv2D Layers: 1, Number of hidden layers: 1, Number of Kernel: 8, Size of Kernel: 3, Activation: reLU, Dropout: 0.2, Regularisation :0

B. With Conv2D

1. Type of Activation Function in Conv2D layer: reLU, Sigmoid,tanh
2. Number of Kernels: 8, 16, 32
3. Size of kernels: 3, 5, 9
4. Number of Conv2D layers: 1, 2, 3, 5

Data Analysis and Preprocessing:

Number of Training Samples: 60000

Number of test Samples: 10000

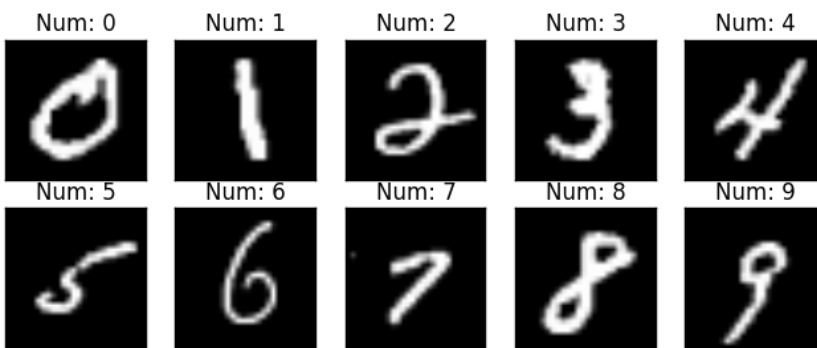
Number of Image Rows: 28

Number of Image Columns: 28

Dimensionality of image: $28 * 28 = 784$

Number of Classes: 10

A sample of each class is shown as follows:



Each image file is first loaded as a list of pixels where each pixel ranges from 0 to 255.

We divide each by 255.0. By dividing by 255.0, the 0-255 range can be described with a 0.0-1.0 range where 0.0 means 0 (0x00) and 1.0 means 255 (0xFF).

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, y_train, x_test, y_test = x_train[:max], y_train[:max], x_test[:max], y_test[:max]
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = np.reshape(x_train, (-1, x_train.shape[1], x_train.shape[2], 1))
x_test = np.reshape(x_test, (-1, x_test.shape[1], x_test.shape[2], 1))
```

We reshape the image data arrays to 2D arrays of width 28, height 28 and 1 channel (B&W image) and there are n number of such arrays (n is 60000 for x_train and 10000 for x_test).

Softmax Regression without Convolution Layers:

Base model We have Used:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=(28, 28, 1)),
    keras.layers.Dense(256, activation=tf.nn.relu),
    keras.layers.Dropout(d),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dropout(d),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

We have used a sequential model with **Flatten** as input layer and **2 Hidden Dense Layers**.

The first hidden **Dense** layer has **256** units and Second layer has **128** units, both have **Rectified Linear Unit** as activation function.


The **Output** layer is a dense layer with 10 units and **Softmax** as activation Function.

Experiment 1: To Test Model Performance by Changing Activation Function, Dropout Probability and Amount of Training Data

We used 3 different activation functions in our model and varied the amount of data used and varied the dropout probability

Activation Function	Dropout Probability	Amount of Training data Samples	Epoch 1 Training Accuracy, Validation Accuracy	Epoch 2 Training Accuracy, Validation Accuracy	Epoch 3 Training Accuracy, Validation Accuracy	Epoch 4 Training Accuracy, Validation Accuracy	Epoch 5 Training Accuracy, Validation Accuracy
ReLu	0	30,000	0.9181, 0.9579	0.9654, 0.9643	0.9778, 0.9670	0.9844, 0.9733	0.9871, 0.9694
ReLu	0	60,000	0.9392, 0.9682	0.9732, 0.9735	0.9813, 0.9760	0.9862, 0.9791	0.9892, 0.9787
ReLu	0.2	30,000	0.8969, 0.9561	0.9531, 0.9675	0.9646, 0.9698	0.9712, 0.9741	0.9777, 0.9738
ReLu	0.2	60,000	0.9222, 0.9593	0.9619, 0.9718	0.9709, 0.9752	0.9765, 0.9788	0.9786, 0.9785
ReLu	0.5	30,000	0.8295, 0.9429	0.9200, 0.9550	0.9342, 0.9608	0.9428, 0.9654	0.9506, 0.9676
ReLu	0.5	60,000	0.8721, 0.9552	0.9359, 0.9634	0.9468, 0.9701	0.9526, 0.9736	0.9567, 0.9748

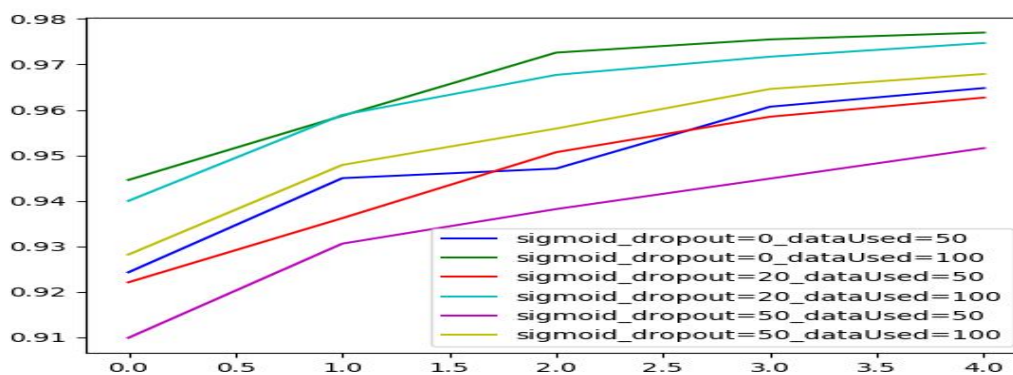
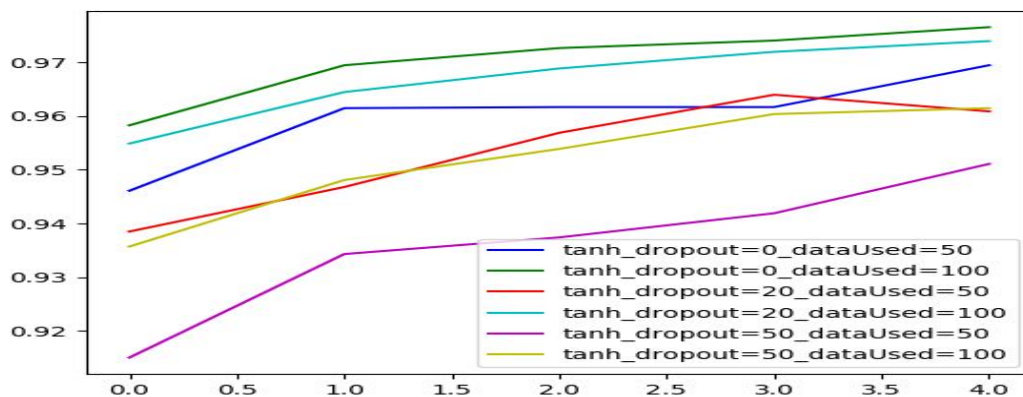
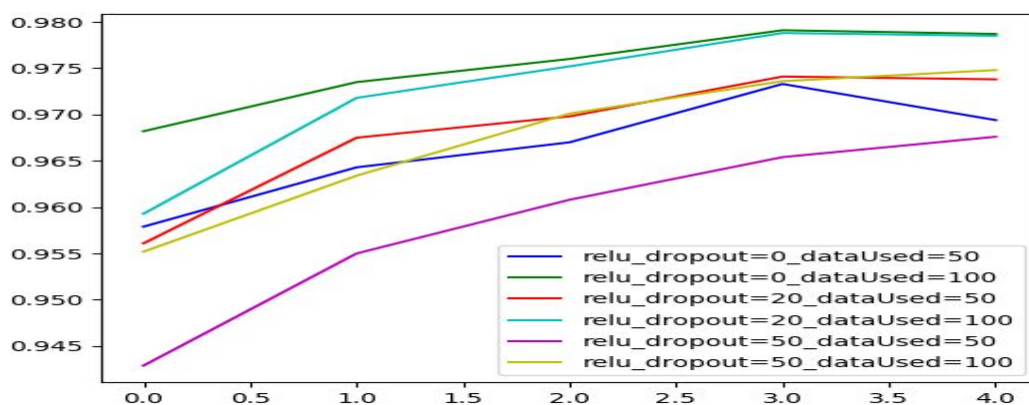
Activation Function	Dropout Probability	Amount of Training data Samples	Epoch 1 Training Accuracy, Validation Accuracy	Epoch 2 Training Accuracy, Validation Accuracy	Epoch 3 Training Accuracy, Validation Accuracy	Epoch 4 Training Accuracy, Validation Accuracy	Epoch 5 Training Accuracy, Validation Accuracy
Sigmoid	0	30,000	0.8593,	0.9357,	0.9523,	0.9638,	0.9740,



			0.9243	0.9450	0.9471	0.9607	0.9648
Sigmoid	0	60,000	0.8939, 0.9446	0.9534, 0.9587	0.9695, 0.9726	0.9778, 0.9755	0.9826, 0.9770
Sigmoid	0.2	30,000	0.8214, 0.9221	0.9203 , 0.9362	0.9386, 0.9507	0.9487 , 0.9585	0.9604, 0.9627
Sigmoid	0.2	60,000	0.8680, 0.9400	0.9401, 0.9589	0.9565, 0.9677	0.9672 , 0.9717	0.9723, 0.9747
Sigmoid	0.5	30,000	0.7197, 0.9099	0.8863, 0.9306	0.9084, 0.9382	0.9243, 0.9449	0.9334 , 0.9516
Sigmoid	0.5	60,000	0.8038, 0.9282	0.9141, 0.9479	0.9323, 0.9559	0.9442, 0.9646	0.9500, 0.9679

Activation Function	Dropout Probability	Amount of Training data Samples	Epoch 1 Training Accuracy, Validation Accuracy	Epoch 2 Training Accuracy, Validation Accuracy	Epoch 3 Training Accuracy, Validation Accuracy	Epoch 4 Training Accuracy, Validation Accuracy	Epoch 5 Training Accuracy, Validation Accuracy
tanh	0	30,000	0.9064, 0.9461	0.955, 0.9615	0.9704, 0.9617	0.9786, 0.9617	0.9858, 0.9695
tanh	0	60,000	0.9272, 0.9583	0.9664, 0.9695	0.9765, 0.9727	0.9832, 0.9741	0.9874, 0.9766
tanh	0.2	30,000	0.8900, 0.9385	0.9369, 0.9468	0.9514, 0.9569	0.9595, 0.9640	0.9679, 0.9609
tanh	0.2	60,000	0.9101, 0.9549	0.9500, 0.9645	0.9604, 0.9689	0.9664, 0.9720	0.9704, 0.9740
tanh	0.5	30,000	0.8389, 0.9150	0.8951, 0.9343	0.9106, 0.9374	0.9207, 0.9419	0.9260, 0.9511
tanh	0.5	60,000	0.9181, 0.9579	0.9654, 0.9643	0.9778, 0.9670	0.9844, 0.9733	0.9871, 0.9694

We can visualize the above table data using the graphs below. In the graphs we have **plotted the Validation Accuracy along the y-axis and Number of epochs along the x-axis.**



Observations From the Above Tables:

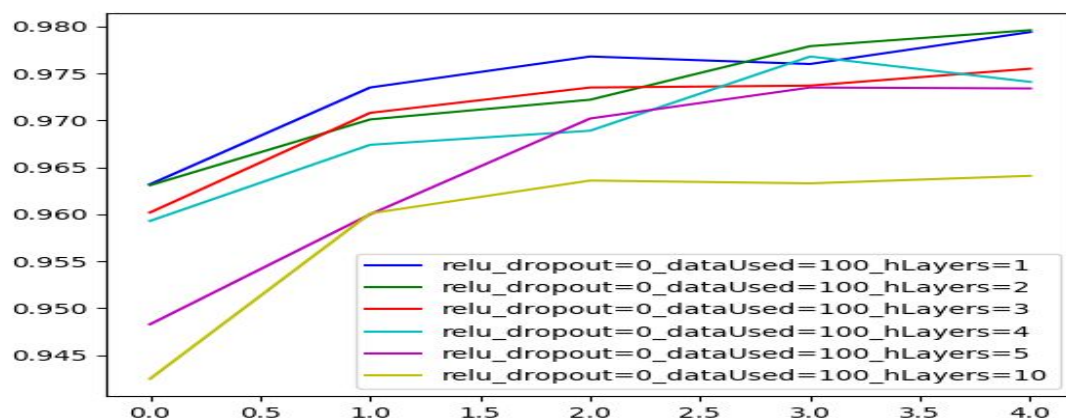
- We got more validation accuracy when more data was used. The reason is that increasing the amount of data improves every algorithm's generalization error.
- The model performed with a dropout probability of about 0 - 0.2. Dropout usually helps remove overfitting. In this experiment the parameters were such that a low dropout probability gave optimal results.
- We got best results for ReLU, followed by sigmoid and tanh activation functions. The differences are very minute.

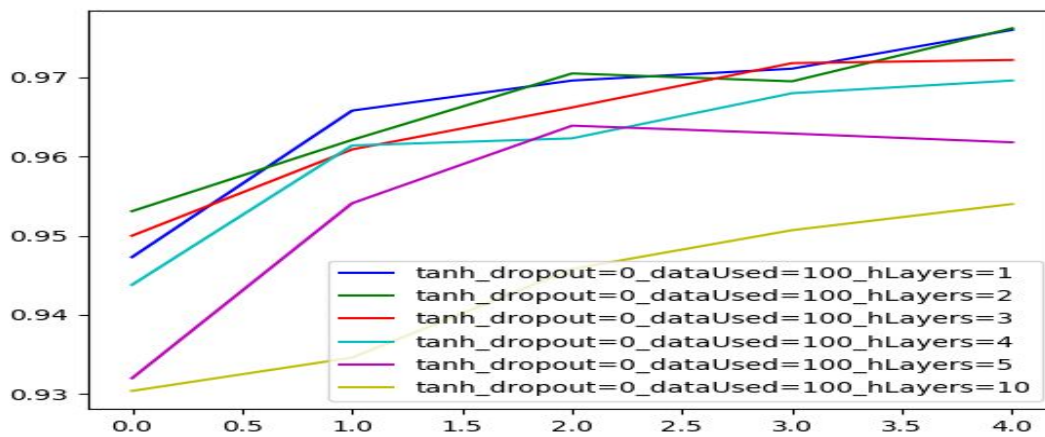
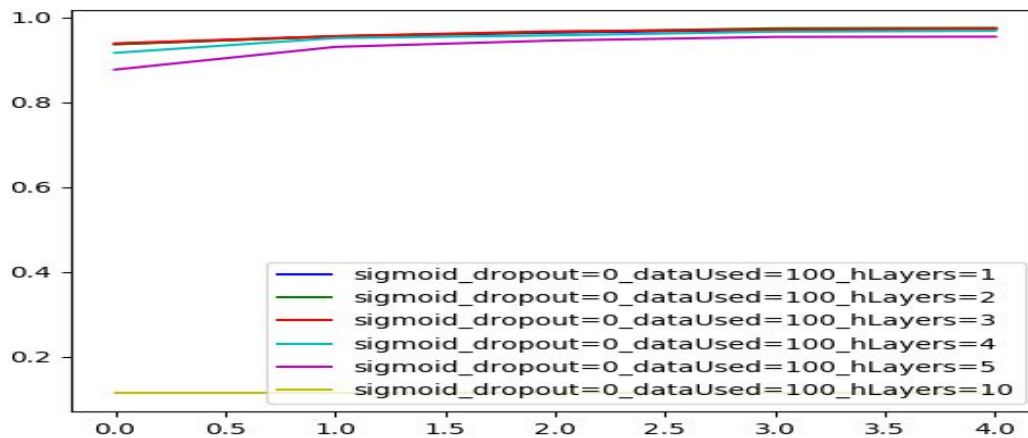
Experiment 2: To Test Model Performance by Changing Number of Hidden Layers

We used 3 different activation functions in our next model and varied the number of hidden layers used, keeping the amount of data and dropout probability constant.

We used 100% of the data (60,000 samples) and dropout probability was set to 0

The following plots help visualise the results we obtained. The labels in the plots denote what they describe.





Observations:

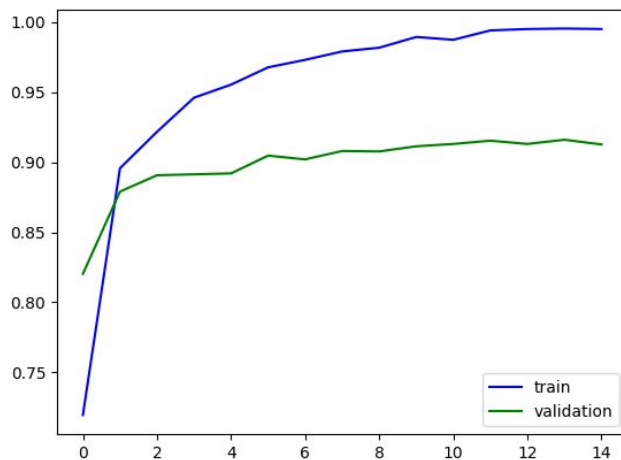
- The model performed very bad with larger number of hidden layers i.e complex models or too simple model. The reason is that if the neural network is too large (too small) the neural network could potentially overfit (underfit) the data meaning that the network would not generalize well out of sample.
- The model performed similarly for smaller number of hidden layers and model with 2 layers performed the best for all three activation functions.
- By adding more hidden layers, training time as well as information learned in each epoch increases. It helps to improve the performance for complex tasks but may not help significantly for relatively simple datasets such as MNIST.

Using the observations from the above experiments we use 100% data , 0 dropout probability , reLU activation function and 2 hidden layers for further experiments, since they were giving the best results.

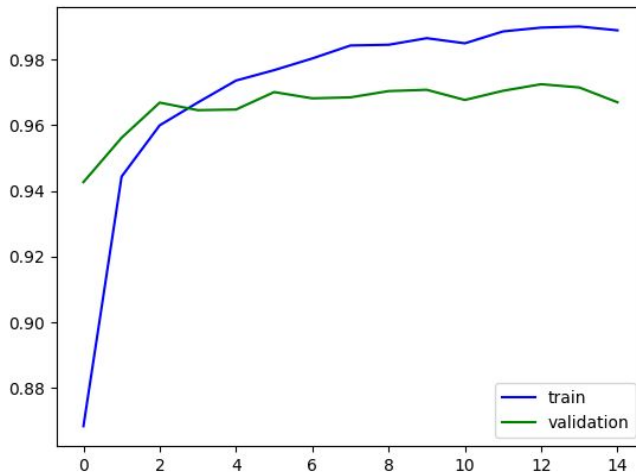
Experiment 3: Changing Data Set Size to Test overfitting because of data scarcity.

In this experiment we change the size of the dataset, starting from 5% to 30% to 100%. The plots below show the results for each dataset size mentioned above serially.

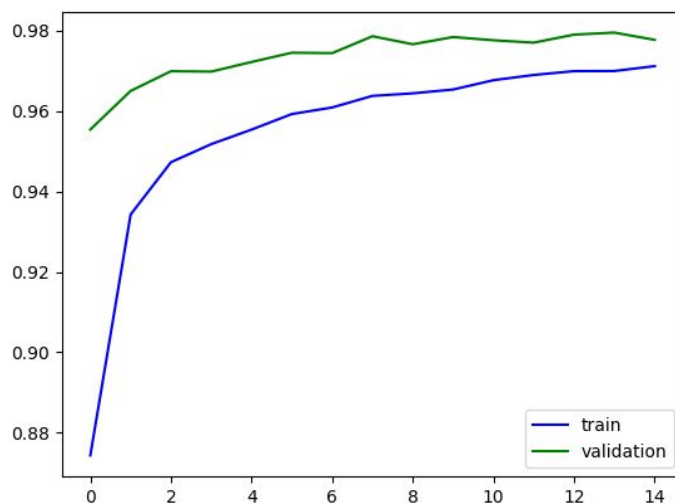
Data Set Size: 5% (3000/60000 data samples) :



Data Set Size: 30% (18000/60000 data samples) :



Data Set Size: 100% (60000/60000 data samples) :



Observations:

- For small datasets, overfitting may occur. Neural networks need large (very large) data sets for training purposes, if the data set is too small overfitting may very well occur - deep learning, with many layers, is particularly problematic.

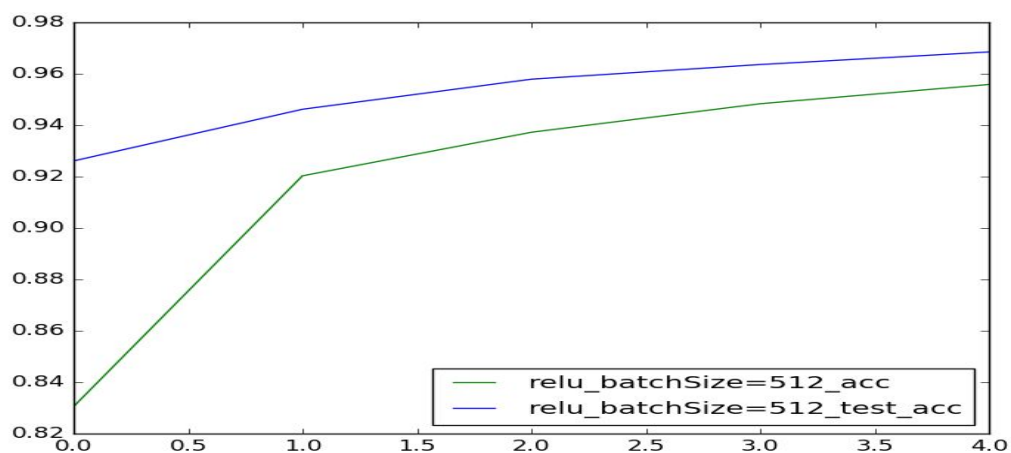
Experiment 4: Changing Batch Size to observe learning rate for different Input Batch Sizes.

We Used 3 different training batch sizes to train and validate the model. The following tables show the observation data:

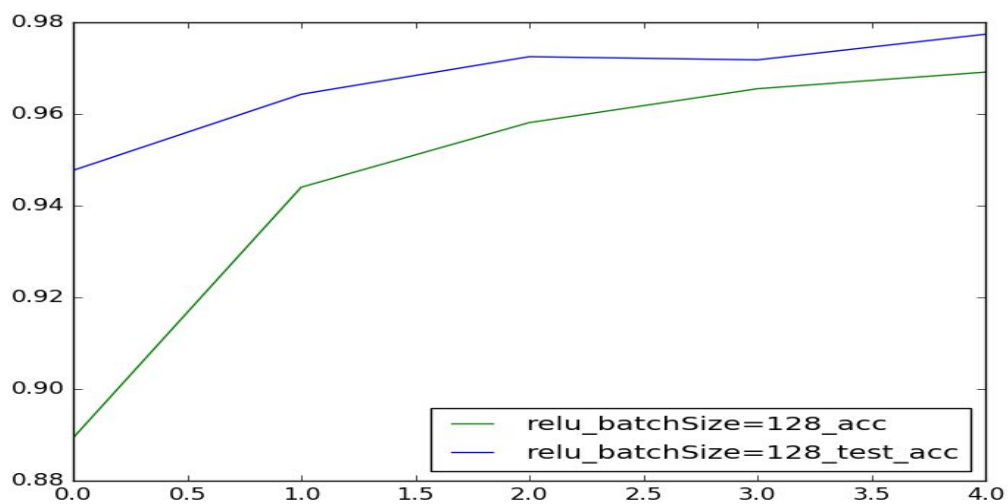
Activation Function	Time taken	Batch size	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5
ReLU	7.40sec	512	0.8305, 0.9261	0.9202, 0.9461	0.9302, 0.9552	0.9372, 0.9579	0.94835, 0.9635
	13.48sec	128	0.8893, 0.9477	0.9440, 0.9643	0.9540, 0.9588	0.9581, 0.9725	0.9655, 0.9718
	34.11sec	32	0.9130, 0.9622	0.9574, 0.9710	0.9596, 0.9739	0.9651, 0.9773	0.9715, 0.9783

The data in the above table can be visualised using the plots below.

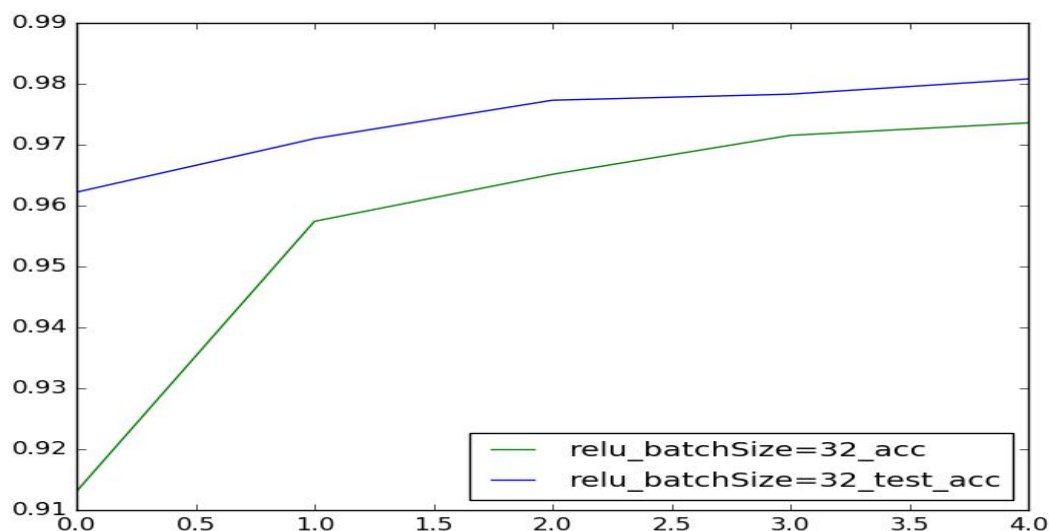
For Batch Size = 512:



For Batch Size = 128:



For Batch Size = 32:



Clearly, the learning rate is lower with input larger batch sizes.

Observations:

- Increasing the batch size decreases the training time. The reason is that Computational speed is simply the speed of performing numerical calculations in hardware which is higher for bigger batch size.
- Increasing the batch size reduces the rate of learning. The reason being the inverse of the above.

Experiment 5: To remove overfitting by using Early Stopping method.

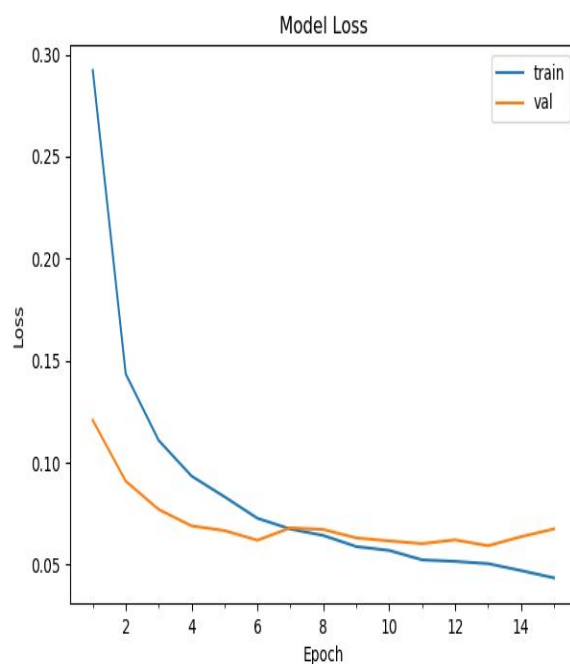
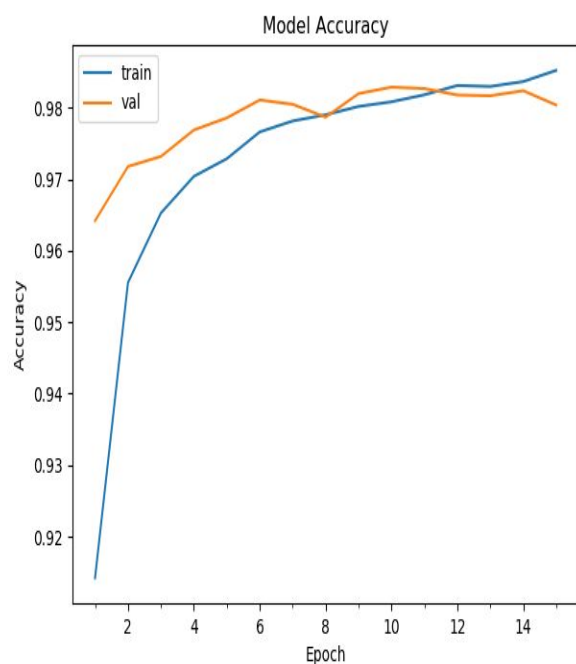
In this experiment we study the training process over a large number of epochs by using Early Stop method. Early Stop method is used to overcome overfitting. One can note that a model overfits if Validation Accuracy is lower than Training accuracy, which interprets to the model failing in real life data.

In our experiment we found that the accuracy saturates after 15 epochs. The following screenshot shows the same:

```

00/00 [=====] 0s 270s/step
chinmoy@labpc:~/Acads/CVML/2nd$ python new_mnist.py
/home/chinmoy/anaconda3/lib/python3.6/site-packages/h5py/_init_.py:36: FutureWarning: Conversion of the second argument of issubdtype from `float` to `np.float64` is deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/100
2018-09-02 20:42:00.797540: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow binary was not built to support: SSE4.2
Epoch 2/100
60000/60000 [=====] - 16s 262us/step - loss: 0.2923 - acc: 0.9142 - val_loss: 0.1208 - val_acc: 0.9642
Epoch 3/100
60000/60000 [=====] - 16s 260us/step - loss: 0.1435 - acc: 0.9555 - val_loss: 0.0910 - val_acc: 0.9718
Epoch 4/100
60000/60000 [=====] - 16s 265us/step - loss: 0.1110 - acc: 0.9653 - val_loss: 0.0771 - val_acc: 0.9732
Epoch 5/100
60000/60000 [=====] - 15s 251us/step - loss: 0.0935 - acc: 0.9704 - val_loss: 0.0690 - val_acc: 0.9769
Epoch 6/100
60000/60000 [=====] - 15s 256us/step - loss: 0.0833 - acc: 0.9729 - val_loss: 0.0667 - val_acc: 0.9786
Epoch 7/100
60000/60000 [=====] - 16s 259us/step - loss: 0.0728 - acc: 0.9766 - val_loss: 0.0620 - val_acc: 0.9811
Epoch 8/100
60000/60000 [=====] - 17s 279us/step - loss: 0.0675 - acc: 0.9782 - val_loss: 0.0681 - val_acc: 0.9805
Epoch 9/100
60000/60000 [=====] - 16s 267us/step - loss: 0.0643 - acc: 0.9790 - val_loss: 0.0673 - val_acc: 0.9787
Epoch 10/100
60000/60000 [=====] - 16s 259us/step - loss: 0.0589 - acc: 0.9802 - val_loss: 0.0631 - val_acc: 0.9820
Epoch 11/100
60000/60000 [=====] - 16s 263us/step - loss: 0.0570 - acc: 0.9808 - val_loss: 0.0617 - val_acc: 0.9829
Epoch 12/100
60000/60000 [=====] - 12s 192us/step - loss: 0.0524 - acc: 0.9818 - val_loss: 0.0603 - val_acc: 0.9827
Epoch 13/100
60000/60000 [=====] - 16s 260us/step - loss: 0.0516 - acc: 0.9831 - val_loss: 0.0622 - val_acc: 0.9818
Epoch 14/100
60000/60000 [=====] - 15s 256us/step - loss: 0.0505 - acc: 0.9830 - val_loss: 0.0593 - val_acc: 0.9817
Epoch 15/100
60000/60000 [=====] - 15s 255us/step - loss: 0.0471 - acc: 0.9837 - val_loss: 0.0637 - val_acc: 0.9824
Epoch 16/100
60000/60000 [=====] - 15s 256us/step - loss: 0.0436 - acc: 0.9852 - val_loss: 0.0675 - val_acc: 0.9804
Epoch 17/100
60000/60000 [=====] - 1s 66us/step
chinmoy@labpc:~/Acads/CVML/2nd$

```



Observations:

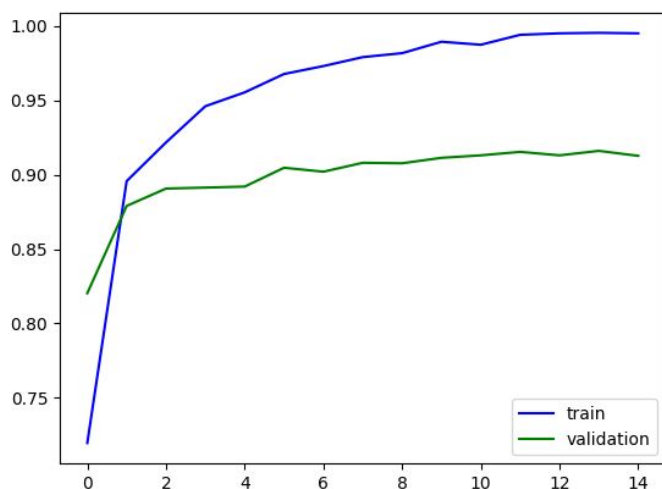
- As the number of epochs increases , the training and validation accuracy increases upto a certain point.
- After a certain number of epochs , the accuracy becomes stable.
- In-order to stop overfitting, the learning process is shutdown by observing/monitoring the validation accuracy.

Experiment 6: To show that increasing dropout probability removes overfitting

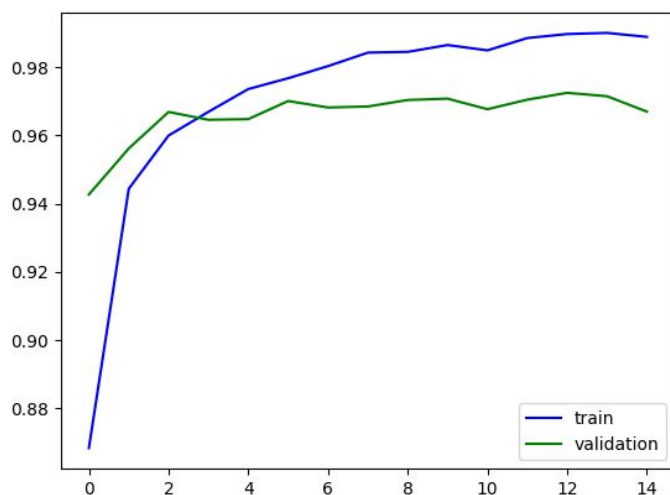
We had noticed that with low amount of data, overfitting may occur. So to mitigate overfitting, we use a Dropout Layer with a certain dropout probability to stop the model from overfitting.

In this experiment we used dataset sizes: 5%, 50% and 100% and dropout probabilities 0, 0.5, 1. One can note that a model overfits if Validation Accuracy is lower than Training accuracy, which interprets to the model failing in real life data.

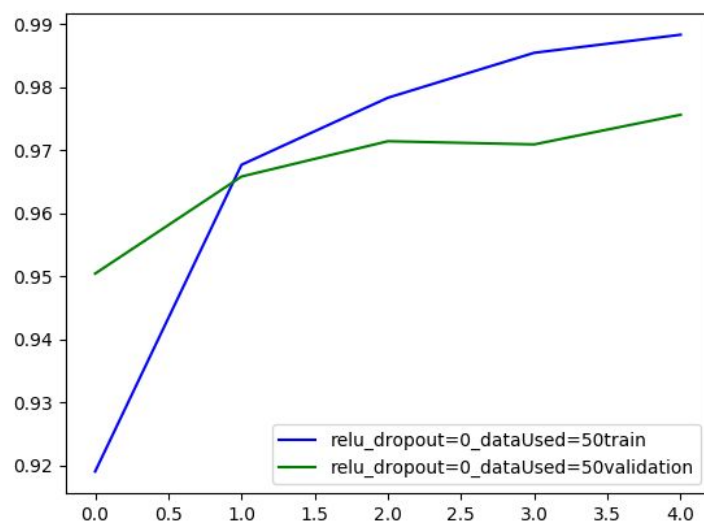
Data = 5% and dropout probability 0:



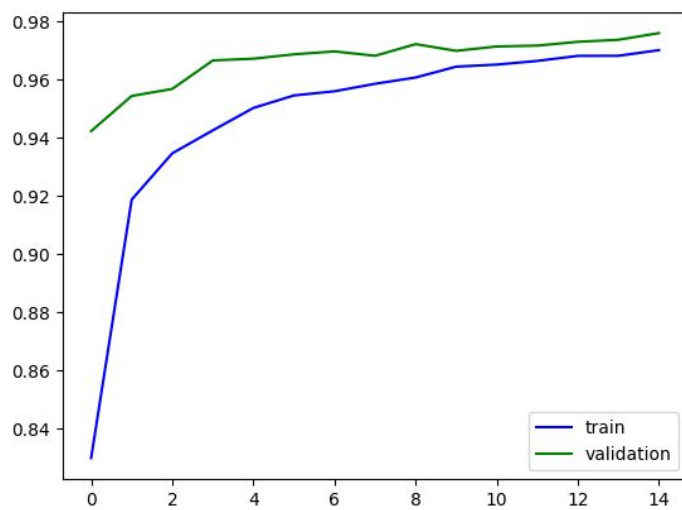
Data = 5% and dropout probability 0.5:



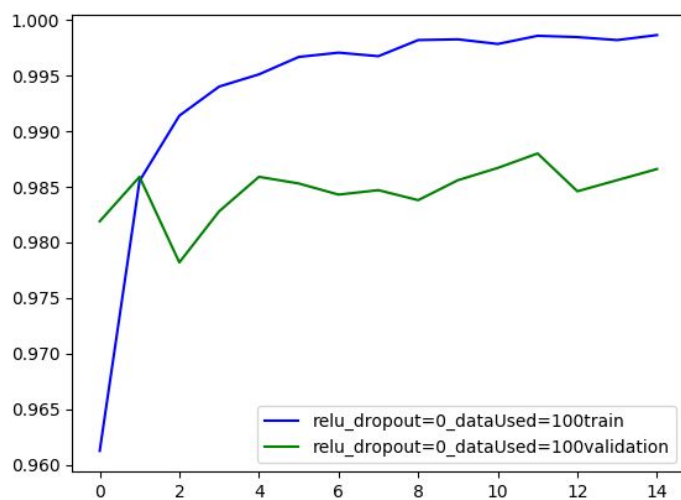
Data = 50% and dropout probability 0:



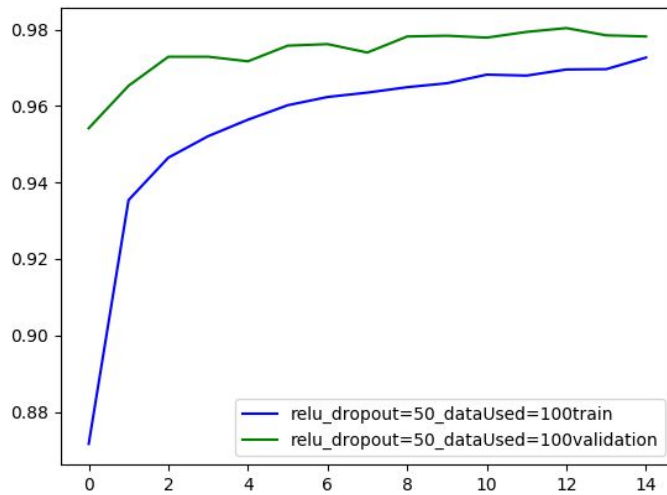
Data = 50% and dropout probability 0.5:



Data = 100% and dropout probability 0:



Data = 100% and dropout probability 0.5:

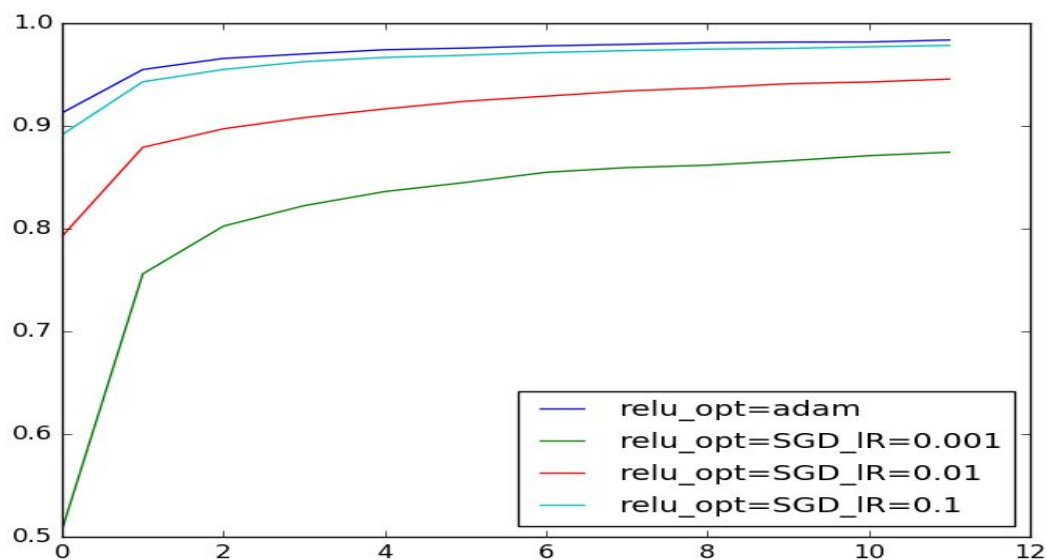


Observations:

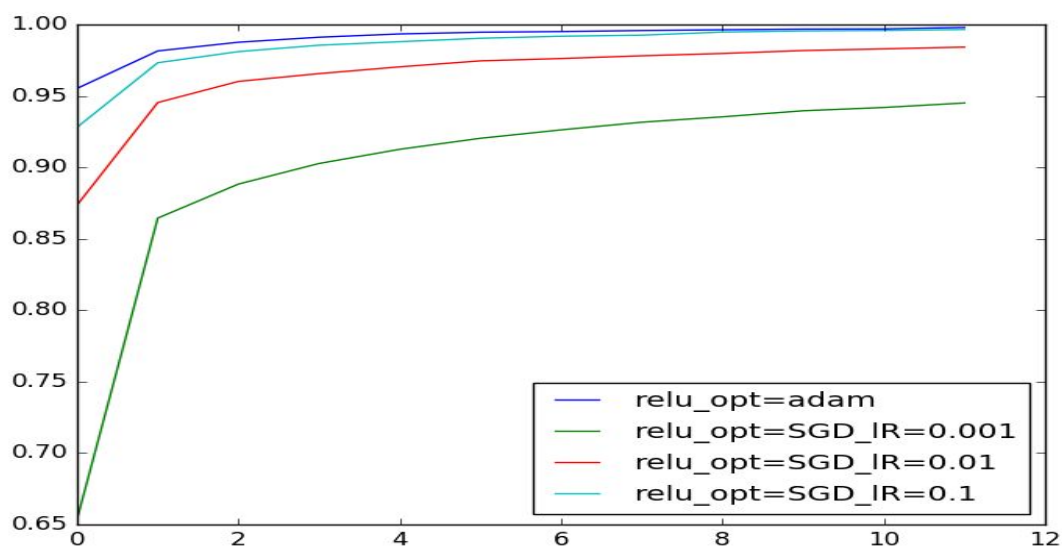
- We found that a certain combination of dataset size and dropout probability can stop a model from overfitting.

Experiment 7: To Test Model Performance by changing the learning rate

For this model, we have varied the learning rate in the SGD optimizer, and compared it with the Adam optimizer. We used ReLU as the activation function. We used learning rates 0.001, 0.01 and 0.1 for the SGD optimizer.



Graph for the experiment with no convolution layers



Graph for the experiment with no convolution layers

Observations:

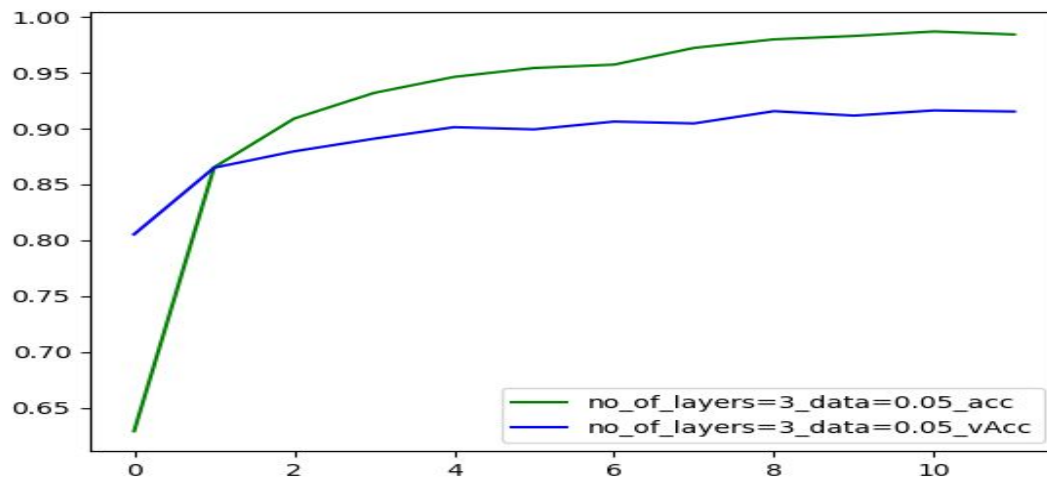
- We found that if the learning rate was decreased less information was learned in each epoch and more epochs were required to give better performance of the model
- If the learning rate was increased more information was learned in each epoch and better results of the model were obtained even with less number of epochs
- We found that the Adam optimizer performed better than the SGD optimizer. For Adam optimizer we don't need to specify a learning rate.

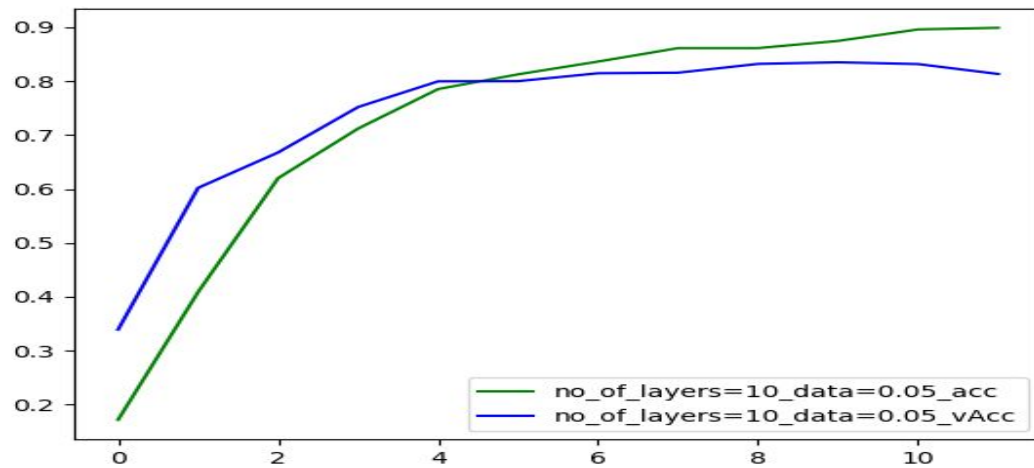
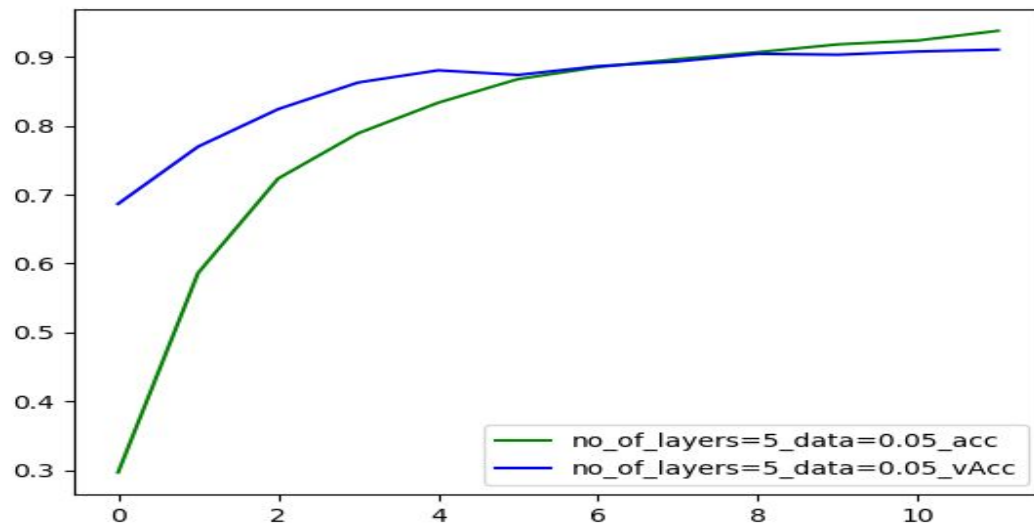
- When SGD optimizer was used the best performance was obtained when learning rate was 0.1 followed by 0.01 and 0.001

Number of epochs for the experiment was 12

Experiment 8: To Test when overfitting occurs on models with more hidden layers

For this model, we have used 5% of the data and used 3, 5 and 10 dense layers in our model.





Observations:

- We found that on a small dataset, lower the number of hidden layers, lower is the number of epochs from which overfitting occurs. The reason is that the learning rate of complex models is lower, so they tend to overfit after a larger number of epochs.

Number of epochs for the experiment was 12

Softmax Regression with Convolution Layers:

```
model = keras.models.Sequential([
    keras.layers.Conv2D(filters = 8, kernel_size = 3, padding='valid', input_shape = (28,28, 1), activation=tf.nn.relu),
    keras.layers.Flatten(),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(256, activation=tf.nn.relu),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

We have used a sequential model with a 2D **Convolution** layer as input layer and **4 Hidden Layers**.

Out of the 4 hidden layers, First is **Flatten** which flattens the 2D array of dimensions 28 * 28 into a 1D array of size 784.

Second hidden Layer is a **Dropout layer** with dropout probability **0.2**

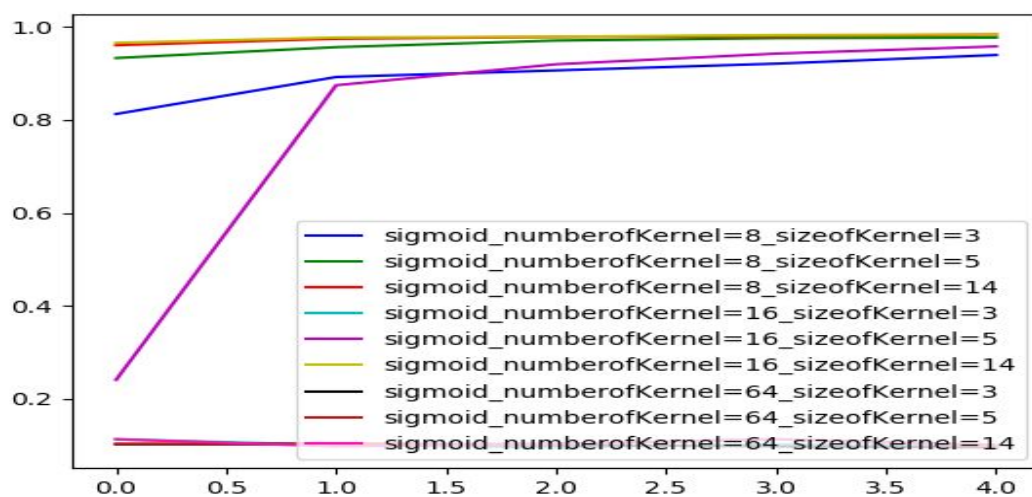
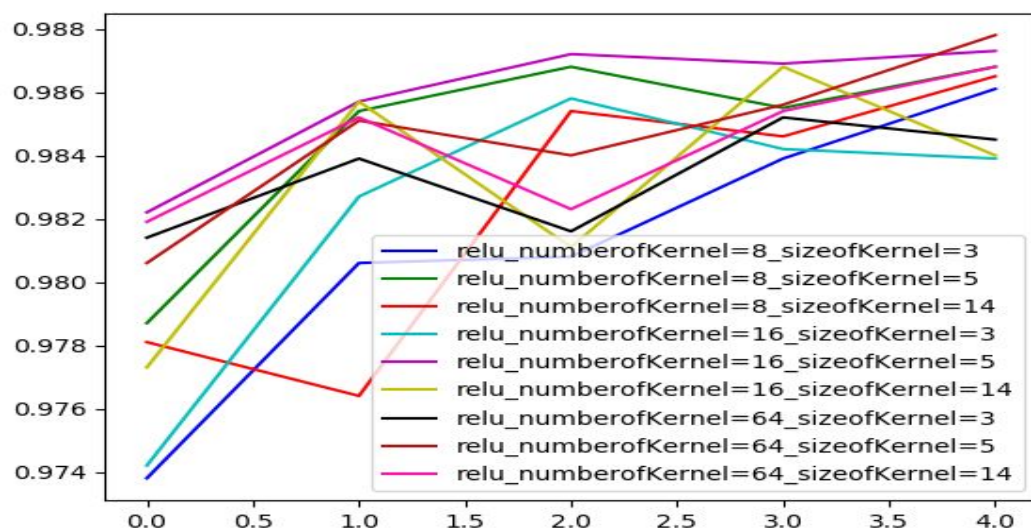
The third hidden **Dense** layer has **256** units and has **Rectified Linear Unit** as activation function.

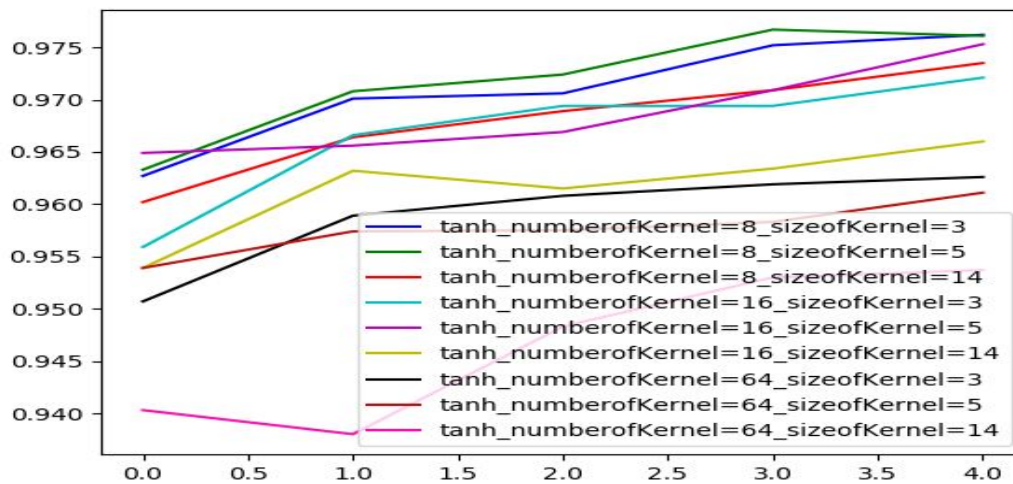
Fourth hidden Layer is a **Dropout layer** with dropout probability **0.2**

The **Output** layer is a dense layer with **10** units and **Softmax** as activation Function.

Experiment 9: To Test Model Performance by Changing Activation Function, Number of convolution Kernels and size of convolution Kernels

For the next model, We have used 1 convolution layer, 1 Dense Layer and varied the number of kernels and the kernel size. Throughout the experiment we used reLU activation function, 0 dropout probability and used 100% of the data.



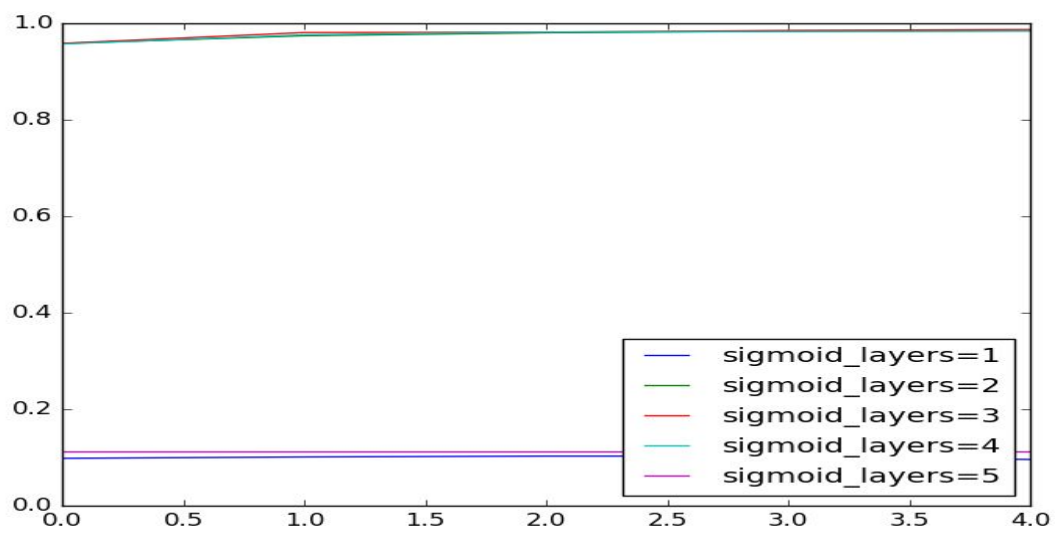
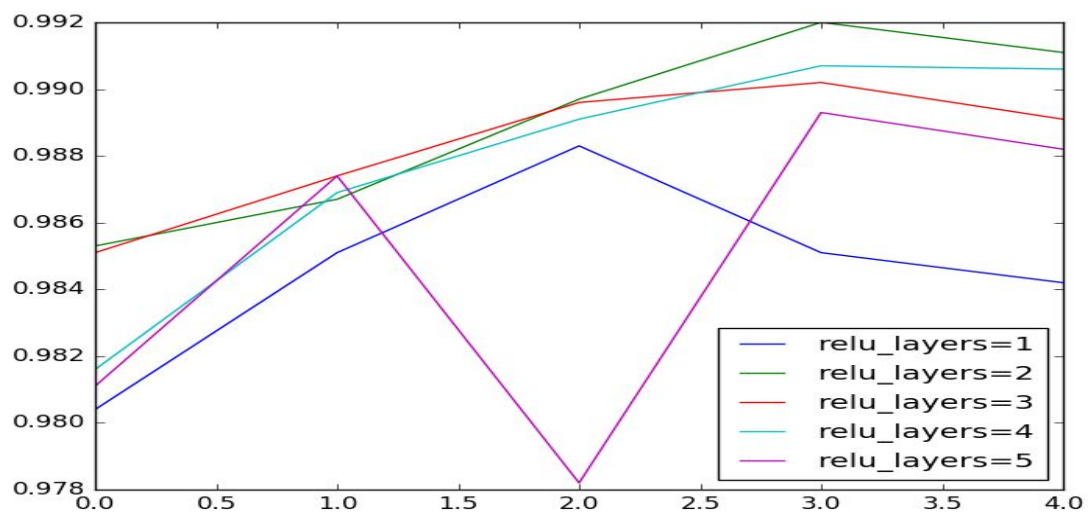


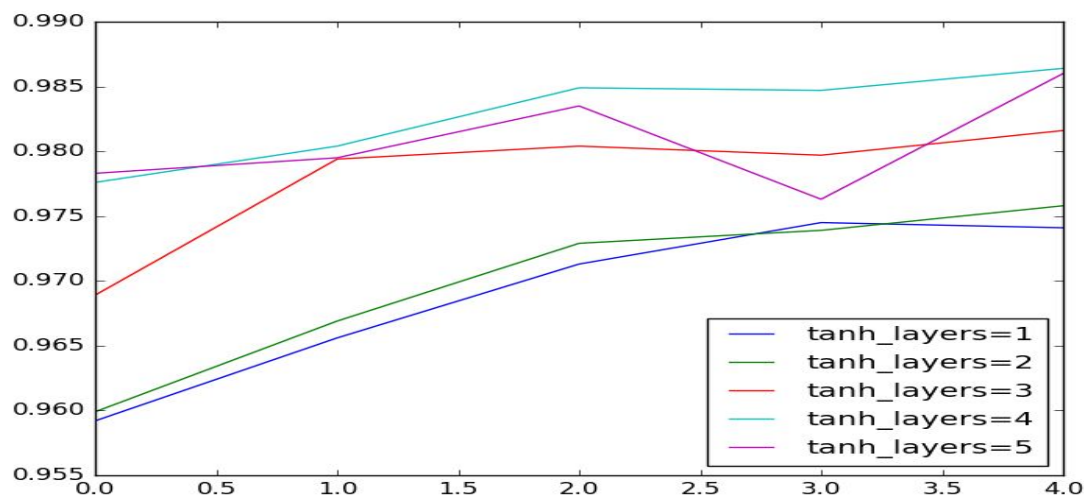
Observations:

- We found that our model worked best when the number of kernels was 16 and the kernel size was 5. Too many Kernels caused the model to perform bad in our experiment.
- For all three activation functions we found that 16 kernels and kernel size 5 was giving consistently good results for the above reasons.

Experiment 10: To Test Model Performance by Changing number of Convolution Layers keeping the activation function as ReLU

For the next model, We have varied the number of convolution layers from 1 to 5. Throughout the experiment we used ReLU activation function, 0.2 dropout probability and used 100% of the data.





Observations:

- Very less number of Convolutional layers can cause the model to perform poorly.
- Too many convolution layers can cause the model to overfit.
- For sigmoid, we find the best results for 2 convolution layers while for tanh as the activation function, it is 4. Sigmoid gives equal result for 2 and 3 convolution layers while performs very poorly for 1,4 and 5.

Basic structure of the codes used for experiment:

```
import tensorflow as tf
import keras
import numpy as np
import matplotlib.pyplot as plt
```

```
def plot(arr,name):
    plt.plot(arr)
    plt.savefig('images/'+name)
    plt.close()
```

```
mnist = keras.datasets.mnist
```

```
max = 60000 #setting the number of sample images to use
```

```

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, y_train, x_test, y_test = x_train[:max],y_train[:max],x_test[:max],y_test[:max]
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = np.reshape(x_train, (-1, x_train.shape[1], x_train.shape[2], 1))
x_test = np.reshape(x_test, (-1, x_test.shape[1], x_test.shape[2], 1))
def MNIST(activation_fn,numberOf_kernel,kernel_s,d,i):
    model = keras.models.Sequential([
        keras.layers.Conv2D(filters = numberOf_kernel, kernel_size = kernel_s, padding='valid', input_shape = (28,28,
1), activation=tf.nn.relu),
        keras.layers.Flatten(),
        keras.layers.Dropout(d),
        keras.layers.Dense(256, activation=activation_fn),
        keras.layers.Dropout(d),
        keras.layers.Dense(10, activation=tf.nn.softmax) #output layer consisting of 10 neurons
    ])
    optimizer = 'adam'
    model.compile(optimizer=optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

    h = model.fit(x_train, y_train, validation_data = (x_test, y_test),epochs=1000)
    model.evaluate(x_test, y_test)[1]
    plot(h.history['acc'],str(i)+"_"+str(numberOf_kernel)+"_"+str(kernel_s)+"_"+str(int(d*10))+ "acc_")
    plot(h.history['loss'],str(i)+"_"+str(numberOf_kernel)+"_"+str(kernel_s)+"_"+str(int(d*10))+ "loss")
    plot(h.history['val_acc'],str(i)+"_"+str(numberOf_kernel)+"_"+str(kernel_s)+"_"+str(int(d*10))+ "v_acc")
    plot(h.history['val_loss'],str(i)+"_"+str(numberOf_kernel)+"_"+str(kernel_s)+"_"+str(int(d*10))+ "v_loss")
    with open('logs.txt','a') as wf:
        for k in range(4):
            wf.write(str(i)+"_"+str(numberOf_kernel)+"_"+str(kernel_s)+"_"+str(int(d*10)) + str(h.history["acc"][k])+" "+
str(h.history["loss"][k])+" "+ str(h.history["val_acc"][k])+" "+str(h.history["val_loss"][k]) + "\n")

activation_functions = [tf.nn.relu]

MNIST(tf.nn.relu,8,3,0.2,1) #we can vary the parameters here

```