CS 574 - Computer Vision Using Machine Learning

Project Report

Classification of digits in MNIST Dataset

Group 26
160101035 Inderpreet Singh Chera
160101039 Kapil Goyal
160101043 Mohit Singh
160101057 Sahib Khan
160101069 Shubham Kumar Koul

PROBLEM STATEMENT

Use the following methods to classify the given image sample of MNIST dataset into one of the 10 possible classes (0 to 9) and Write a report stating the detailed summary after tweaking different parameters.

1. Logistic Regression
2. Multi-Layer Perceptron
3. Deep Neural Network
4. Deep Convolutional Neural Network

Language / Library Used

Language Used: Python 3
Libraries Used:

- Keras is used for implementing **neural networks.**
- Scikit-learn is used for implementing **logistic regression.**
- Matplotlib is used for plotting various graphs.

About Dataset

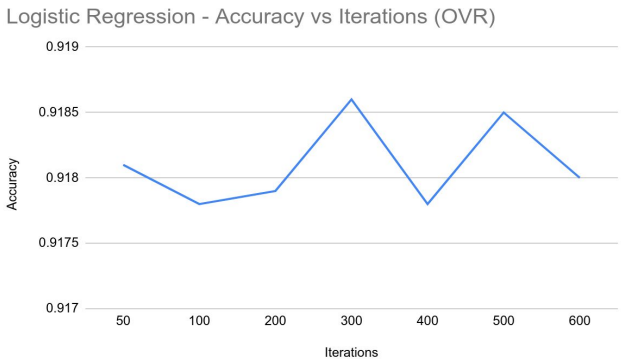
- MNIST dataset contains two parts. First part contains 60,000 labeled images of Numerical digits which is used for training and validation. While the second part contains 10,000 labeled images which is used as a test set. Training set and Test set are such that there are no common writers in the two sets. All images contain 28 x 28 pixels
- Training Set: 54,000 images of 28 x 28 pixels. (90 % of the MNIST train set)
- Validation Set: 6,000 images of 28 x 28 pixels. (10 % of the MNIST train set)
- Test Set: 10,000 images of 28 x 28 pixels. (100% of the MNIST test set)

Logistic Regression

Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable that is dependent on one or more nominal, ordinal, interval or ratio-level independent variables. We implemented our model using multinomial and one vs rest logistic regression and observed that multinomial class has better accuracy.
By default configuration Used:

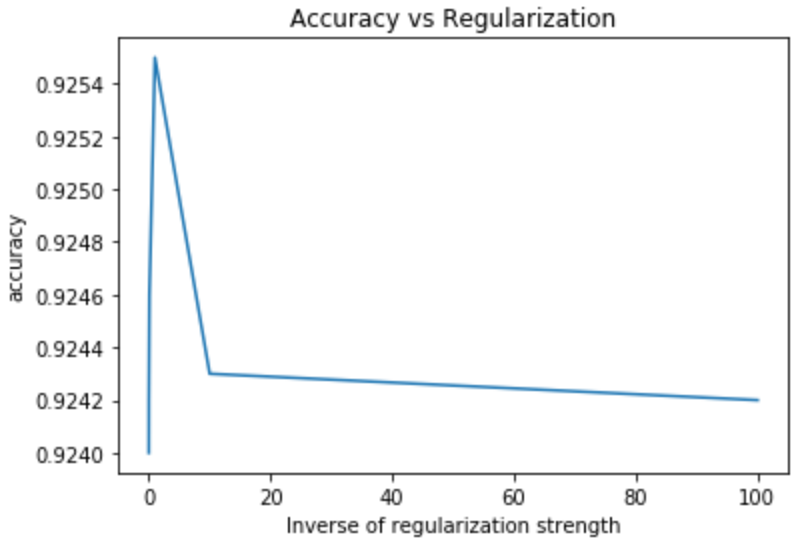
- Solver - lbfgs
- Iterations - 100
- Regularization Strength - 1
- multi-class - multinomial

Accuracy vs Iterations :



In one vs rest accuracy is best found at 300 iterations.

In multinomial accuracy is best found at 100 iterations.
Beyond that accuracy decreases because our model may overfit.



We iterated over C (inverse of regularization strength) in ([0.001, 0.01, 0.1, 1, 10, 100]) and found that accuracy is best at C = 1. Using C lower than that our model may underfit while higher value may overfit our model.

Multi-Layer Perceptron

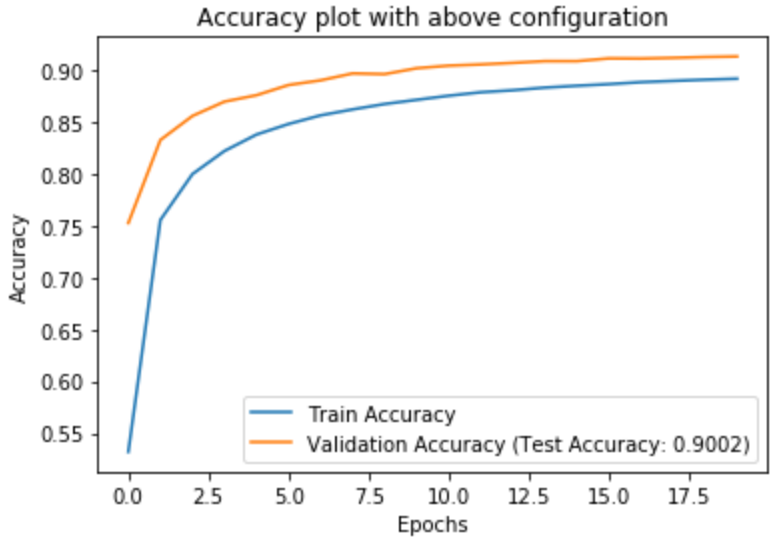
A multilayer perceptron (MLP) is a class of feedforward artificial neural network. A MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training

We will implement MLP model using the following configuration and will tweak some of these parameters to analyze their effect on the accuracy of the model.

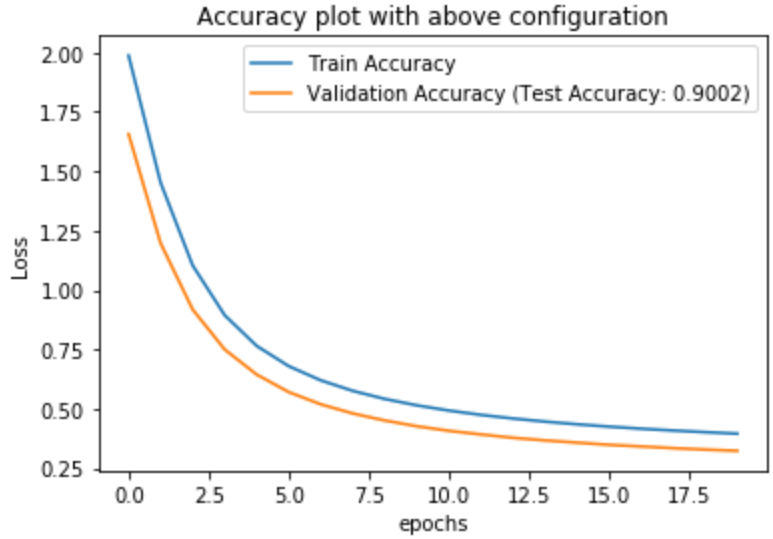
Configuration Used:

- Input Layer: 784 Neurons
- Hidden Layers: 1 layer of 512 Neurons
- Output Layer: 10 Neurons
- Optimizer: SGD
- Activation Function:

    - Hidden Layer: sigmoid function
    - Output Layer: softmax
- Batch Size: 128
- Epochs: 20



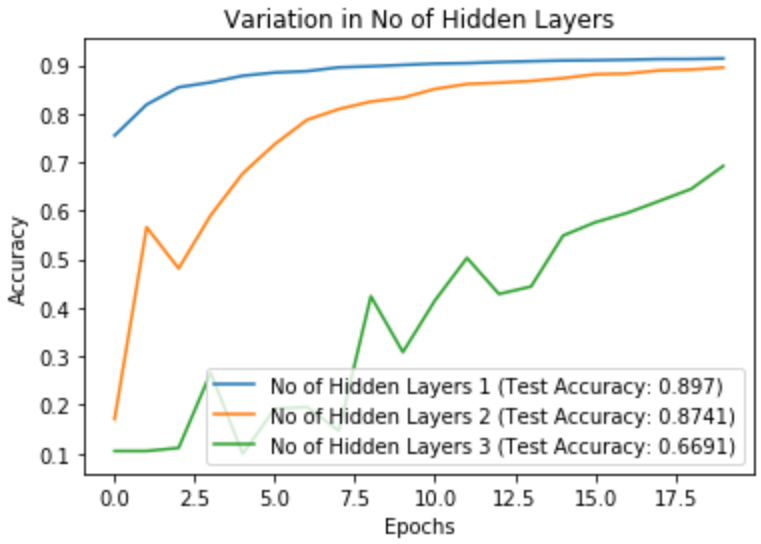
Training and Validation accuracy keeps increasing. That means there is no overfitting in this case.



Similarly training and validation loss keep decreasing. That means there is no overfitting in this case.

Different Number of Hidden Layers

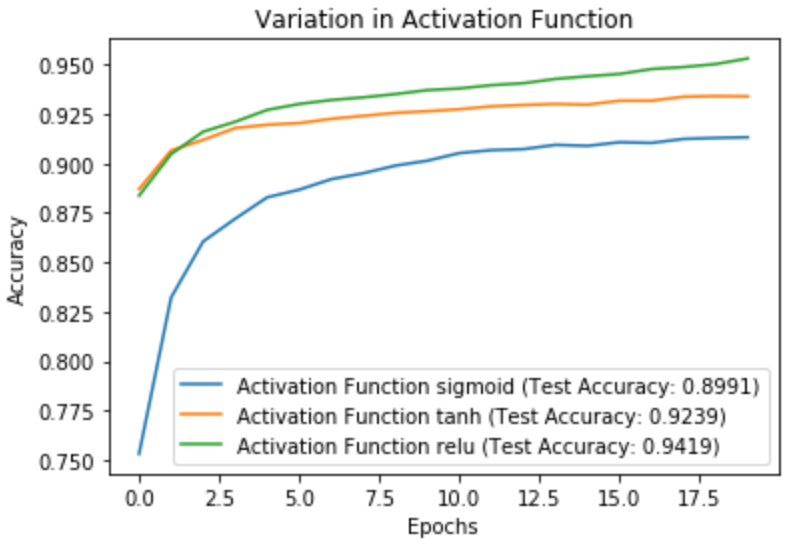
We will change the number of hidden layers from 1 to 3 and see its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



We can see best accuracy is observed when we use 1 layer instead of 2 or 3 layers. We can see that if we use 3 layers accuracy decrease drastically, since it suffers with vanishing gradient problem. This problem arises because we use stochastic gradient descent in MLP which suffers with the problem in case of more hidden layers.

Different Activation function

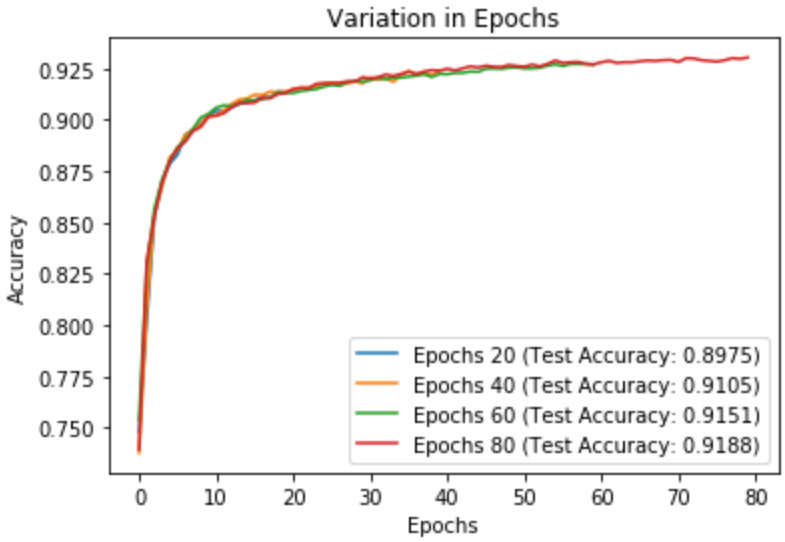
We will change the activation function of hidden layer (i.e. sigmoid, relu and tanh) and compare its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



We observe that relu has the best accuracy and sigmoid has lowest accuracy after 20 epochs. It is because relu has gradient equal to 1 and sigmoid has maximum gradient value equal to 0.25. This makes sigmoid to increase very slowly as compared to the other two. Similarly tanh function increases faster than sigmoid but slower that relu.

Different Number of Epochs

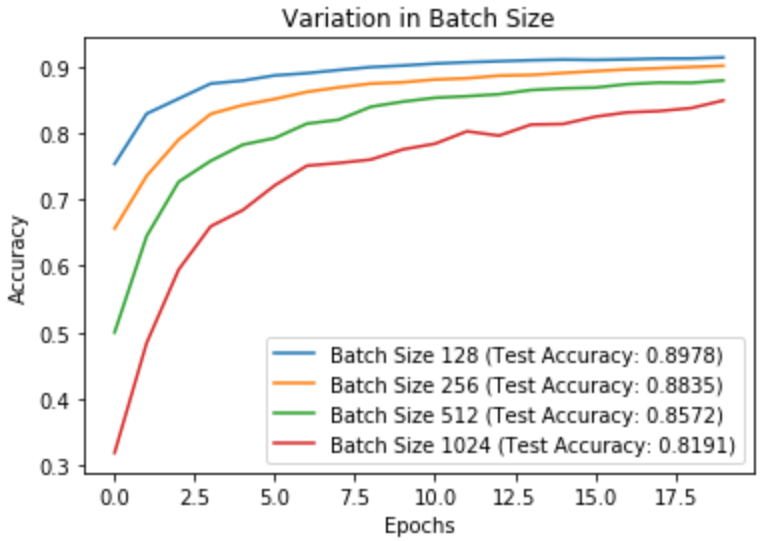
We will change the number of epochs (i.e 20, 40, 60, 80) and compare its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



We observe that accuracy increases with increasing numbers of epochs. That's also implies that there is no overfitting in this case.

Different Batch Size

We will change the batch size i.e.(128, 256, 512, 1024) and compare its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



We observe that using smaller batch size makes convergence faster as compared to the bigger batch size. This because using smaller batch size gives good generalization of data. We also observed that using higher batch size decreases computation time as compared to smaller batch size.

Deep Neural Networks

Deep Neural Networks is almost same as Multi-layer Perceptron except that deep neural networks are more complex (i.e. they have more neurons and layers as compared to multi-layer perceptron). Deep neural networks use more advanced optimization and activation functions so that they do not suffer from vanishing gradient problem.

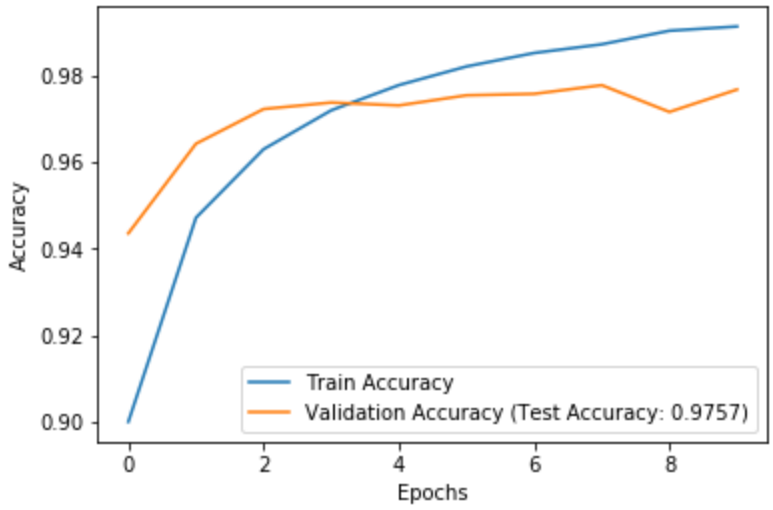
Vanishing Gradient Problem occurs when we try to train a Neural Network model using Gradient based optimization techniques. When we do Back-propagation i.e moving backward in the Network and calculating gradients of loss(Error) with

respect to the weights , the gradients tends to get smaller and smaller as we keep on moving backward in the Network. This means that the neurons in the Earlier layers learn very slowly as compared to the neurons in the later layers in the Hierarchy.

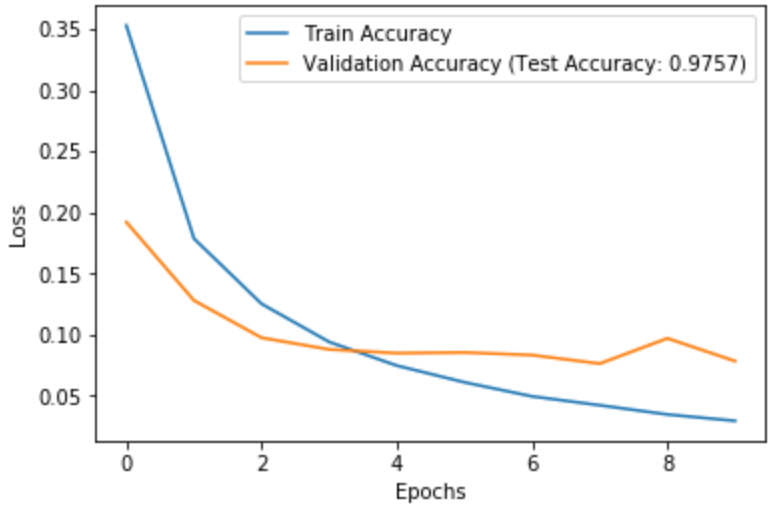
We will implement deep neural networks model using the following configuration and will tweak some of these parameters to analyze their effect on the accuracy of the model.

Configuration Used:

- Input Layer: 784 Neurons
- Hidden Layers: 2 layers of 100 Neurons
- Output Layer: 10 Neurons
- Optimizer: RMSprop
- Activation Function:
    - 1st Hidden Layer: tanh function
    - 2nd Hidden Layer: tanh function
    - Output Layer: softmax
- Batch Size: 128
- Epochs: 10



We observe that validation accuracy stops increasing after 3 epochs while training accuracy keeps increasing. This shows there is overfitting in this case.

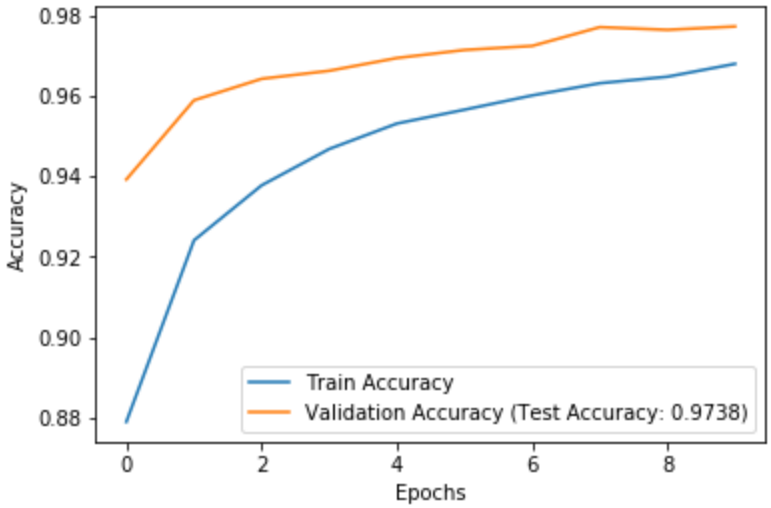


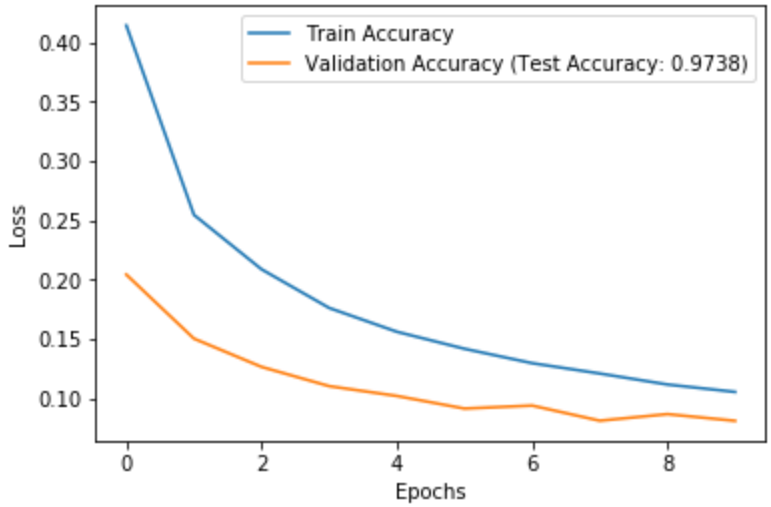
Similar pattern is observed in case of loss versus epochs. Loss for validation set stops decreasing while for training set it keeps decreasing after 3 epochs which shows overfitting.

Overfitting Improvements

Dropout

First we solved overfitting problem by using dropout. Dropout randomly shuts off some nodes and stop the gradients flowing through it. So, our forward and back propagation happen without those nodes. In that case the rest of the nodes need to pick up the slack and be more active in the training. We used dropout factor equal to 0.2. That means 2 out 5 neurons will be randomly discarded.

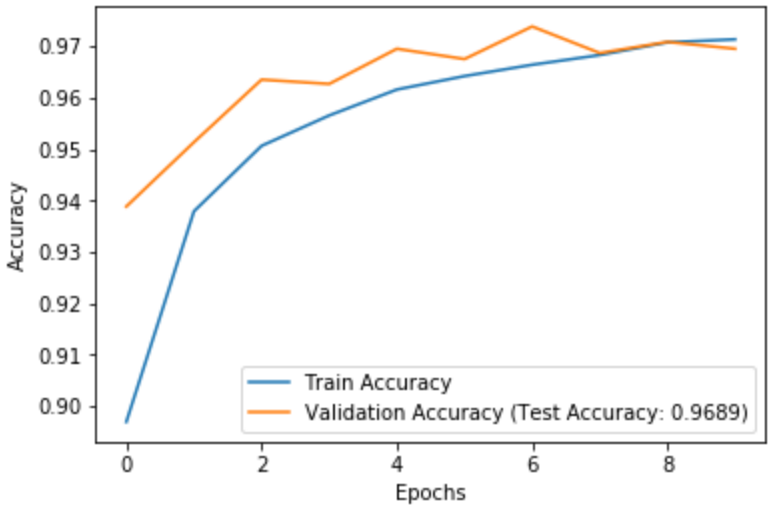


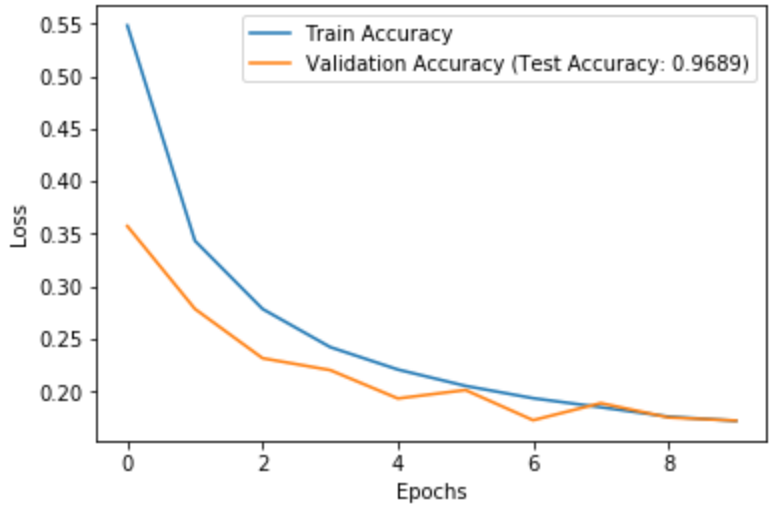


From the above graphs we can observe that after using dropout overfitting problem is solved (as validation accuracy is increasing and validation loss is decreasing)

L2 Regularization

L2 regularization adds “squared magnitude” of coefficient as penalty term to the loss function. We have used lambda = 0.001 that is regularization factor.

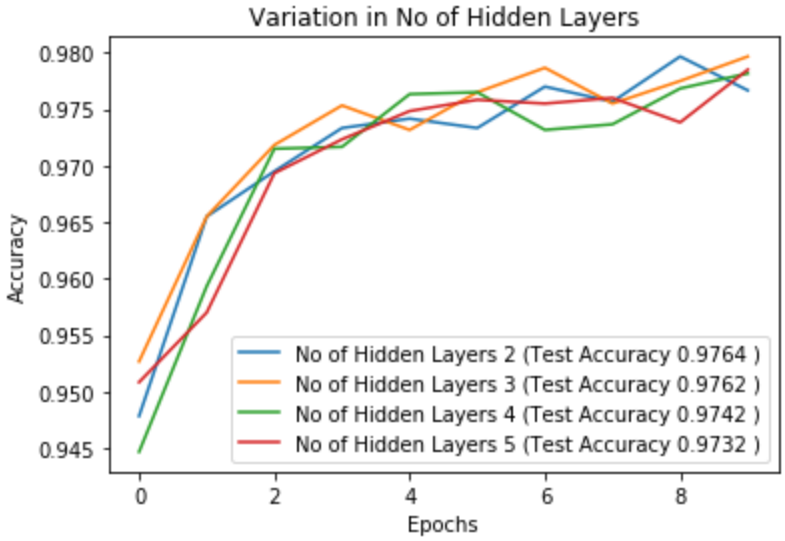




From the above graphs we observe that it solves overfitting problem to some extent but it is not as good as dropout because if we train on more epochs it may again overfit.

Different Number of Hidden Layers

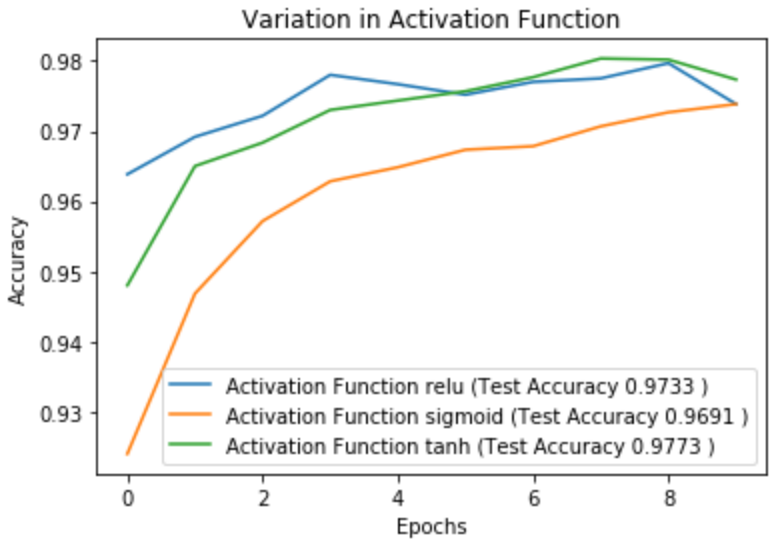
We will change the number of hidden layers from 2 to 5 and see its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



We observed that increasing the number of hidden layers decreases test accuracy because using more hidden layers may increase the complexity of the model and hence our model may overfit.

Different Activation function

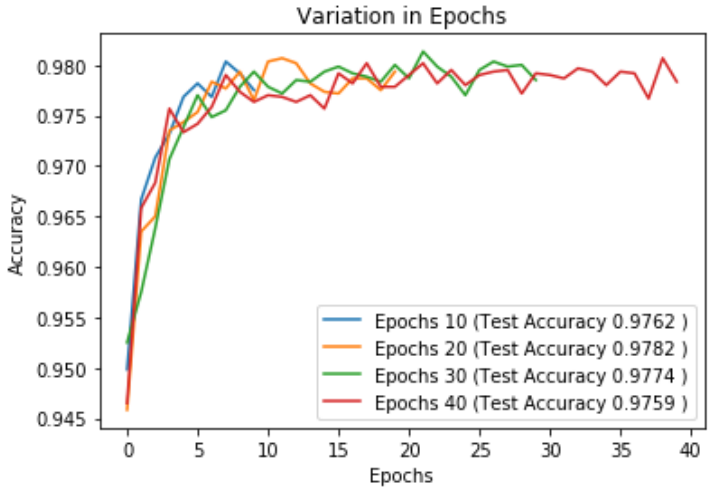
We will change the activation function of hidden layer (i.e. sigmoid, relu and tanh) and compare its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



We observed that sigmoid function increases slowly while relu and tanh increases faster than sigmoid. Tanh gives best accuracy while sigmoid gives lowest accuracy among the three.

Different Number of Epochs

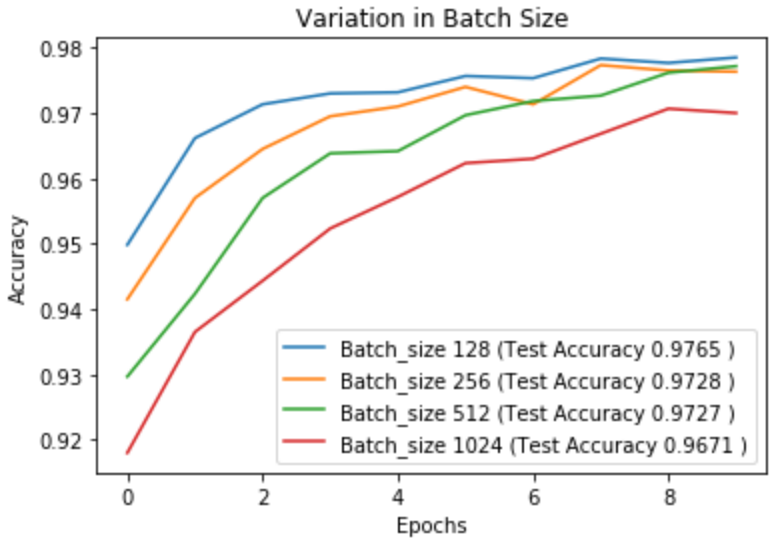
We will change the number of epochs (i.e 10, 20, 30, 40) and compare its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



We observe that accuracy first increases from 10 to 20 epochs. After that it starts decreasing which shows that our model is overfitting.

Different Batch Size

We will change the batch size i.e.(128, 256, 512,1024) and compare its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



We observe that using smaller batch sizes convergence is reached faster than using bigger batch size. This is because smaller batch sizes give better generalization of data than bigger batch size. We also observed that it takes less time to train using larger batch size than smaller ones.

Deep Convolutional Neural Network

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery.

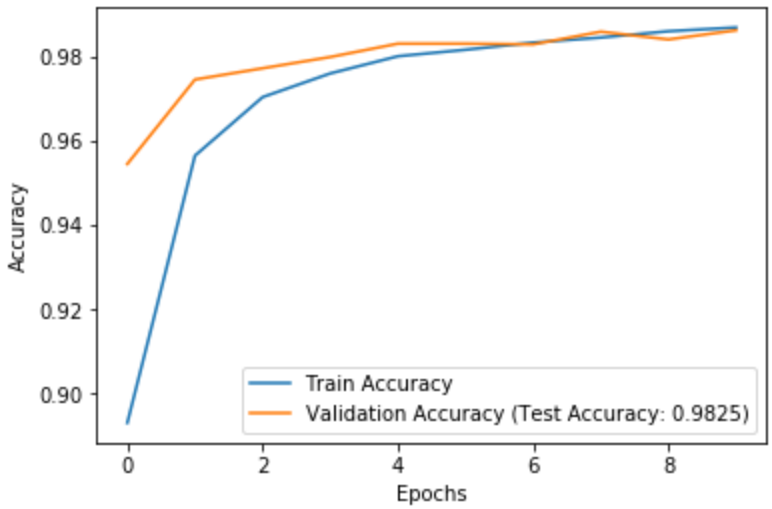
CNNs are regularized versions of multilayer perceptrons. Multilayer perceptrons usually refer to fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to overfitting data. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. However, CNNs take a different

approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.

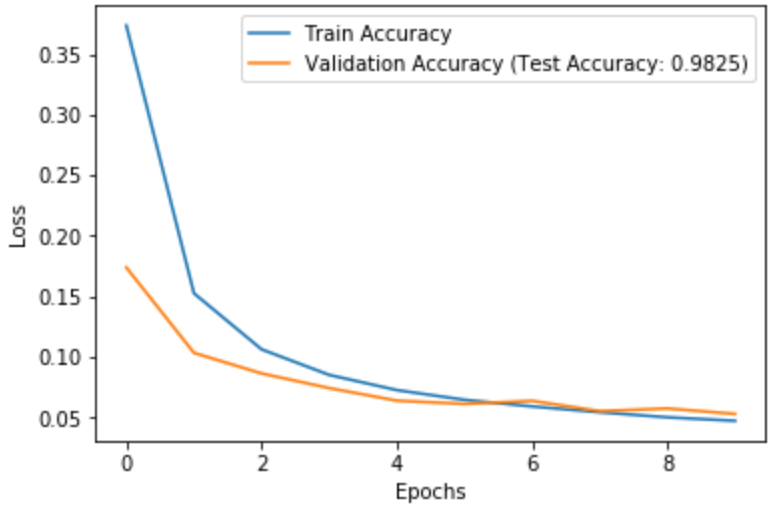
We will implement convolutional neural networks model using the following configuration and will tweak some of these parameters to analyze their effect on the accuracy of the model.

Configuration Used:

- Input Layer: 784 Neurons
- Hidden Layers: 1 2D Convolutional layer with 32 kernels of size 3 x 3
- Output Layer: 10 Neurons
- Optimizer = Adadelta
- Activation Function:
    - Hidden Layer: relu function
    - Output Layer: softmax
- Batch Size: 128
- Epochs: 10



We can observe here that our Validation Accuracy decreases after 6th epoch and again decreases after 8th epoch but our train accuracy keeps on increasing, hence it shows the case of overfitting.



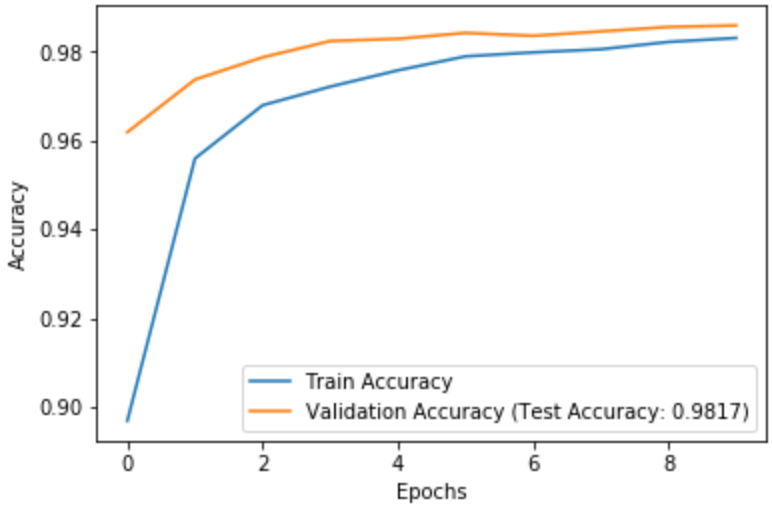
Similarly we can see here that our loss increases after 6th epoch and 8th epoch for validation set but loss keeps on decreasing for training set thus it is a case of overfitting.

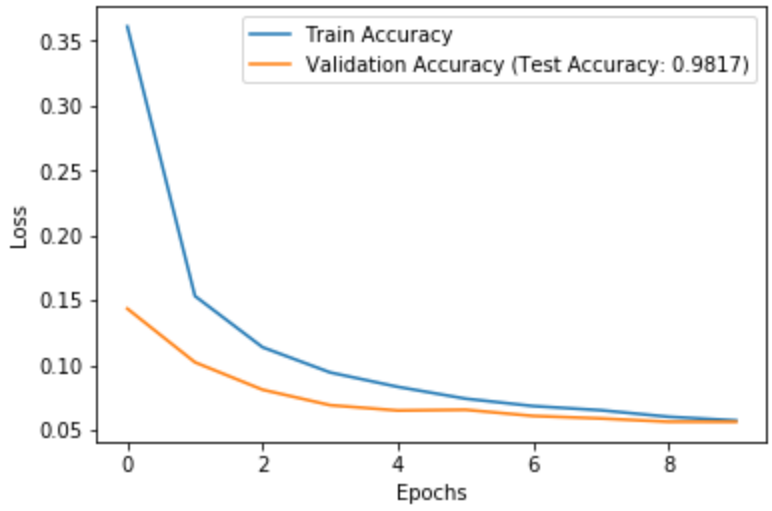
Overfitting Improvements

To handle the case of overfitting we have used two techniques :-

1. **Dropout**

   Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel. In dropout, some percentage of layer outputs are randomly ignored or “dropped out” knows. We used dropout factor equal to 0.2. That means 2 out 5 neurons will be randomly discarded.

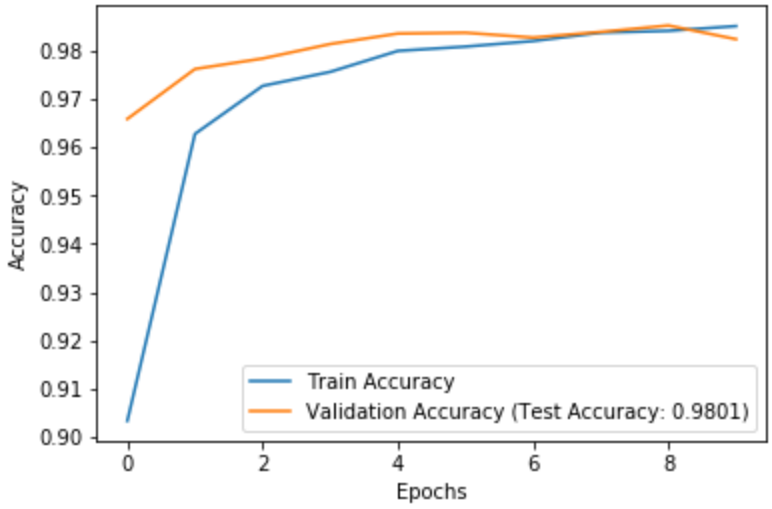


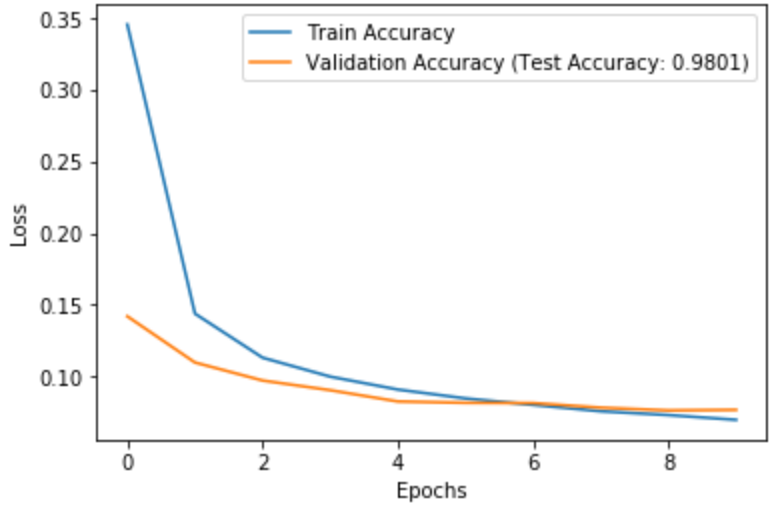


In the above plots we can see that overfitting problem is solved as our validation accuracy also increases along with training accuracy.

2. L2 regularisation

L2 regularisation adds “squared magnitude” of coefficient as penalty term to the loss function. Here we have used *lamda* = 0.001.



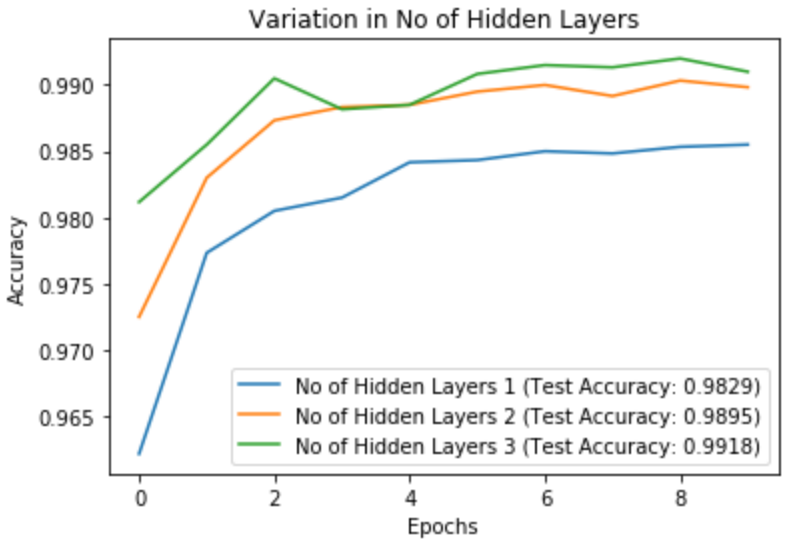


From the above plots we can see that L2 regularisation tries to remove the overfitting problem to some extent but overfitting still remains.

Different Number of Hidden Layers

We will use the following configuration of hidden layers:

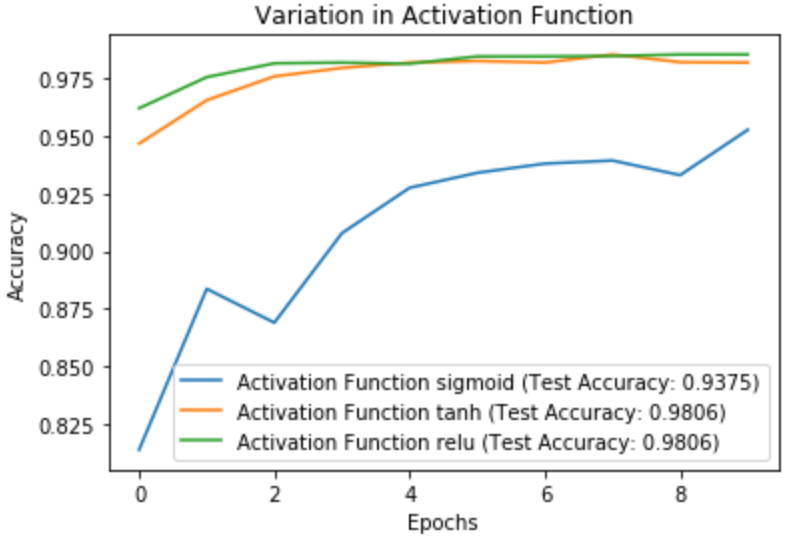
- 1 Hidden Layer
  - 1 2D Convolutional layer with 32 kernels of size 3 x 3
- 2 Hidden Layers
  - 1 2D Convolutional layer with 32 kernels of size 3 x 3
  - 1 2D Convolutional layer with 64 kernels of size 3 x 3
- 3 Hidden Layers
  - 1 2D Convolutional layer with 32 kernels of size 3 x 3
  - 1 2D Convolutional layer with 64 kernels of size 3 x 3
  - 1 Dense Layer with 128 neurons



We can see from the above plots that on increasing the number of hidden layers our test accuracy increases. This is because on increasing the number of hidden layers our model can capture more complex features of the input hence it will work more efficiently.

Different Activation function

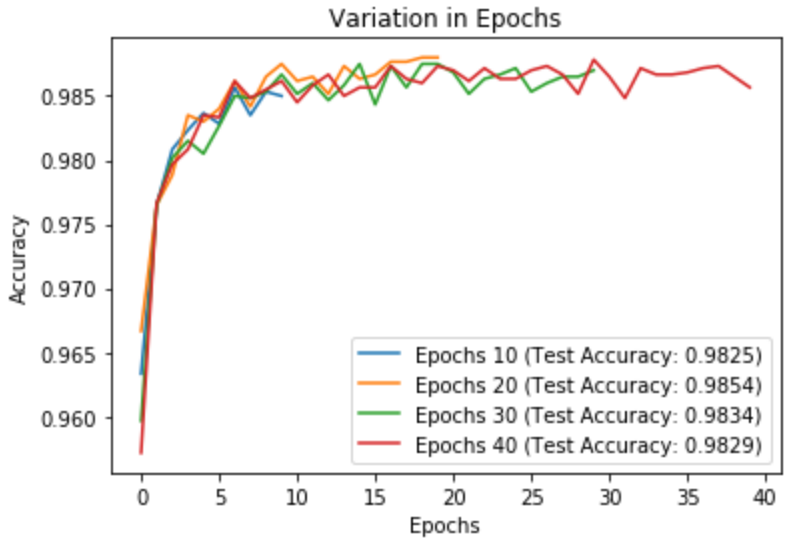
We will change the activation function of hidden layer (i.e. sigmoid, relu and tanh) and compare its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



From the above graph we can see that model having activation function as ReLu and tanh have better accuracy than model with sigmoid.This is because partial derivatives of sigmoid tends to zero at extremes which result in slow learning rate.

Different Number of Epochs

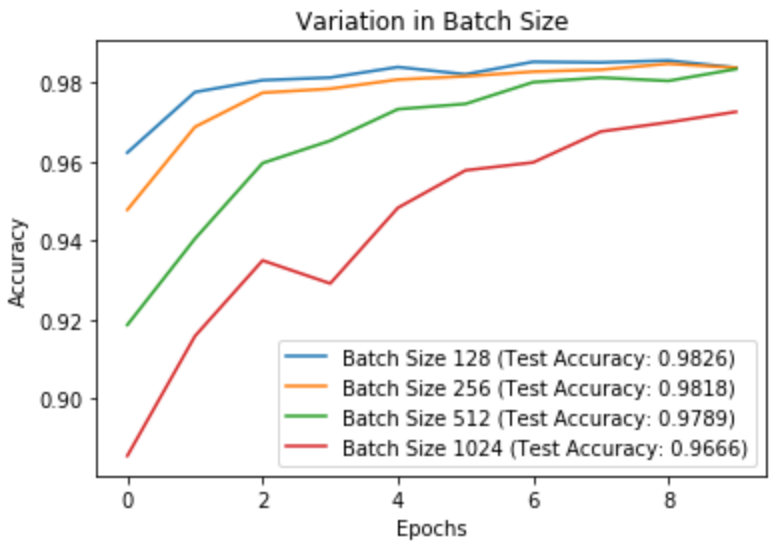
We will change the number of epochs (i.e 10, 20, 30, 40) and compare its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



We can see from above that our accuracy first increases upto 20 epochs then it started decreasing because on increasing epochs our model starts to overfit.

Different Batch Size

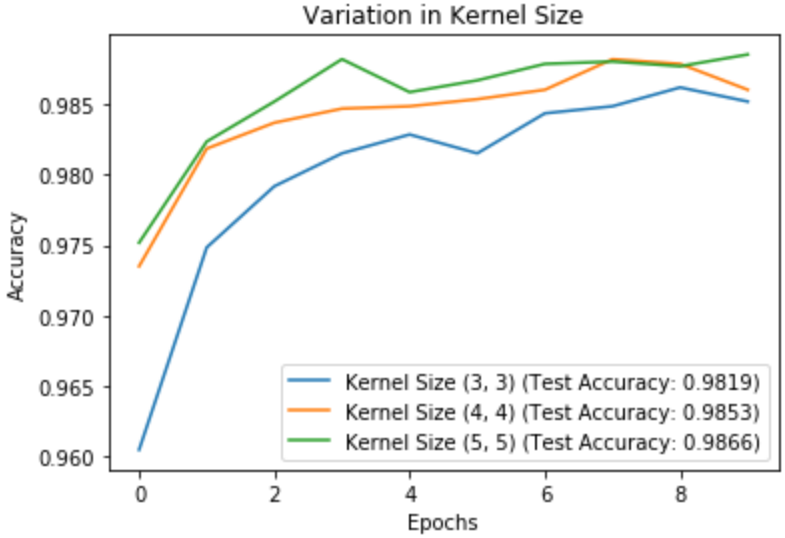
We will change the batch size i.e.(128, 256, 512, 1024) and compare its effect on the accuracy of the model. We will keep all other parameters same as the main configuration.



We observe that using smaller batch sizes convergence is reached faster than using bigger batch size. This is because smaller batch sizes give better generalization of data than bigger batch size. We also observed that it takes less time to train using larger batch size than smaller ones.

Different Kernel Size

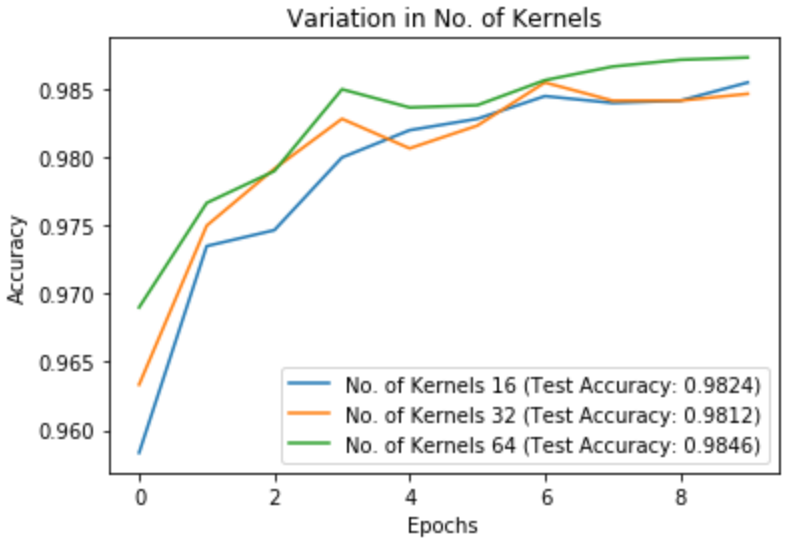
We changed the size of kernel of 2D convolution layer i.e. ((3 x 3), (4 x 4) and (5 X 5)) and observed the results keeping all other parameters same as initial configuration.



If kernel size is too low it may result in overfitting because of very high complexity of model. On the other hand if kernel size too high it may result in underfitting because overlooking of features. We observe that accuracy increases by increasing the kernel size from (3 x 3) to (5 x 5). We also observed that using larger kernel size model converges faster than using smaller kernel size.

Different No. of Kernels

We changed the no. of kernels of 2D convolution layer i.e. (16, 32 and 64) and observed the results keeping all other parameters same as initial configuration.



We observed that using more no. of kernels we can capture more features as the no. of possible combination grow and hence our model converges more accurately and faster. For more complex dataset, it is generally observed that using more kernels will perform better.

Conclusion

- We observe that accuracy increases as the number of epochs increases. But it also takes more time to train the model.
- As our Dataset is image based, CNN outperform other models because it makes use of pattern found in data.
- To avoid overfitting a dataset with a large number of test examples should be chosen.

Codes

Logistic Regression

```
import keras
from keras.datasets import mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

X_train.shape

X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)

"""Logistic Regression"""

from sklearn.linear_model import LogisticRegression

# all parameters not specified are set to their defaults
itr=[0.001,0.01,0.1,1,10,100]
yl=[]
for i in itr:
  logisticRegr = LogisticRegression(solver='lbfgs',
multi_class='multinomial', C=i)
  logisticRegr.fit(X_train, Y_train)
  score = logisticRegr.score(X_test, Y_test.tolist())
  print(score)
  yl.append(score)

import matplotlib.pyplot as plt
plt.plot(itr, yl)
plt.xlabel('Inverse of regularization strength')
plt.ylabel('accuracy')
plt.title('Accuracy vs Regularization')
plt.show()

print(yl)

score = logisticRegr.score(X_test, Y_test.tolist())

print(score)

logisticRegr.predict(X_test)
```

```
Y_test
```

Multi Layer Perceptron

```
from __future__ import print_function

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.regularizers import l2
from keras.optimizers import RMSprop
import matplotlib.pyplot as plt
from google.colab import files

#Data loaded from mnist dataset available in Keras
num_classes=10
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# convert Each class to binary catogery
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

def train_model(layers = 1, hidden_layer_width =512,
                activation_fn = 'sigmoid', epochs = 10,
                dropout = False, dropout_factor = 0.2, regularisation = 0.0,
                batch_size = 128, l2_reg = 0.0, optimizer ='sgd' ):

  model = Sequential()
  model.add(Dense(hidden_layer_width, kernel_regularizer = l2(l2_reg),
                  activation = activation_fn, input_shape = (784,)))
  if dropout:
    model.add(Dropout(dropout_factor))
```

```
  for i in range(1,layers):
    model.add(Dense(hidden_layer_width, kernel_regularizer = l2(l2_reg),
                    activation = activation_fn))
    if dropout:
      model.add(Dropout(dropout_factor))


  model.add(Dense(num_classes, activation = 'softmax'))

  model.summary()

  model.compile(loss = 'categorical_crossentropy',
                optimizer = optimizer,
                metrics = ['accuracy'])

  history = model.fit(x_train, y_train,
                      batch_size = batch_size,
                      epochs = epochs,
                      verbose = 1,
                      validation_split = .1)
  score = model.evaluate(x_test, y_test, verbose=0)
  return history,score

def print_plot(histories, title, xlabel, ylabel, legend,
               print_train = False, print_loss = False):

  for history in histories:
    if print_train:
      plt.plot(history.history['acc'])
    plt.plot(history.history['val_acc'])
  plt.title(title)
  plt.ylabel("Accuracy")
  plt.xlabel("Epochs")
  plt.legend(legend, loc='best')
  plt.show()

  if print_loss:
    for history in histories:
      if print_train:
        plt.plot(history.history['loss'])
      plt.plot(history.history['val_loss'])
    plt.title(title)
    plt.ylabel("Loss")
```

```
    plt.xlabel("Epochs")
    plt.legend(legend, loc='best')
    plt.show()

  pass

# Base Configuration
number_of_hidden_layers = 1
hidden_layer_width = 512
optimizer = 'sgd'
activation_fn = "sigmoid"
batch_size = 128
epoch = 20

history, score = train_model(layers=number_of_hidden_layers,
                             hidden_layer_width=hidden_layer_width,
                             optimizer=optimizer,
                             activation_fn=activation_fn,
                             batch_size=batch_size,
                             epochs=epoch)

print_plot([history], "",
           "Epochs", "Accuracy",
            ["Train Accuracy",
             "Validation Accuracy (Test Accuracy: {})".format(score[1])],
             True, True)

#variation in layers
history=[]
label=[]

for i in range(1,4):
  [h,s]=train_model(layers=i,
                    hidden_layer_width=hidden_layer_width,
                    optimizer=optimizer,
                    activation_fn=activation_fn,
                    batch_size=batch_size,
                    epochs=epoch)
  history.append(h)
  label.append("No of Hidden Layers {} (Test Accuracy: {})".format(i,s[1]))

print_plot(history,"Variation in No of Hidden Layers"," epochs","acc",label)
```

```
#variation in Activation Func
history=[]
label=[]

for i in ['sigmoid','tanh','relu']:
  [h,s]=train_model(layers=number_of_hidden_layers,
                    hidden_layer_width=hidden_layer_width,
                    optimizer=optimizer,
                    activation_fn=i,
                    batch_size=batch_size,
                    epochs=epoch)
  history.append(h)
  label.append("Activation Function {} (Test Accuracy: {})".format(i,s[1]))

print_plot(history,"Variation in Activation Function"," epochs","acc",label)

#Variation in epochs
history=[]
label=[]

for i in range(20,81,20):
  [h,s]=train_model(layers=number_of_hidden_layers,
                    hidden_layer_width=hidden_layer_width,
                    optimizer=optimizer,
                    activation_fn=activation_fn,
                    batch_size=batch_size,
                    epochs=i)
  history.append(h)
  label.append("Epochs {} (Test Accuracy: {})".format(i,s[1]))

print_plot(history,"Variation in Epochs"," epochs","acc",label)

#Variation in Batch Size
  history=[]
  label=[]

  i=128
  while i<1025:
    [h,s]=train_model(layers=number_of_hidden_layers,
                      hidden_layer_width=hidden_layer_width,
                      optimizer=optimizer,
                      activation_fn=activation_fn,
                      batch_size=i,
```

```
                    epochs=epoch)
    history.append(h)
    label.append("Batch Size {} (Test Accuracy: {})".format(i,s[1]))
    i*=2

print_plot(history,"Variation in Batch Size"," epochs","acc",label)
```

Deep Neural Network

```
from __future__ import print_function

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.regularizers import l2
from keras.optimizers import RMSprop, SGD, Adadelta, Adam
import matplotlib.pyplot as plt
from google.colab import files

#no of hidden layes
#activation fnc of each layer
# epochs
#overfitting
#regularisation


#Data loaded from mnist dataset available in Keras
num_classes=10
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# convert Each class to binary catogery
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

def train_model(layers = 2, hidden_layer_width =512,
```

```
                activation_fn = 'relu', epochs = 20,
                dropout = False, dropout_factor = 0.2, regularisation = 0.0,
                batch_size = 128, l2_reg = 0.0, optimizer = RMSprop()):

  model = Sequential()
  model.add(Dense(hidden_layer_width, kernel_regularizer = l2(l2_reg),
                  activation = activation_fn, input_shape = (784,)))
  if dropout:
    model.add(Dropout(dropout_factor))

  for i in range(1,layers):
    model.add(Dense(hidden_layer_width, kernel_regularizer = l2(l2_reg),
                    activation = activation_fn))
    if dropout:
      model.add(Dropout(dropout_factor))


  model.add(Dense(num_classes, activation = 'softmax'))

  model.summary()

  model.compile(loss = 'categorical_crossentropy',
                optimizer = optimizer,
                metrics = ['accuracy'])

  history = model.fit(x_train, y_train,
                      batch_size = batch_size,
                      epochs = epochs,
                      verbose = 1,
                      validation_split = .1)
  score = model.evaluate(x_test, y_test, verbose=0)
  return history,score

def print_plot(histories, title, xlabel, ylabel, legend,
               print_train = False, print_loss = False):

  for history in histories:
    if print_train:
      plt.plot(history.history['acc'])
    plt.plot(history.history['val_acc'])
  plt.title(title)
  plt.ylabel("Accuracy")
  plt.xlabel("Epochs")
```

```
  plt.legend(legend, loc='best')
  plt.show()

  if print_loss:
    for history in histories:
      if print_train:
        plt.plot(history.history['loss'])
      plt.plot(history.history['val_loss'])
    plt.title(title)
    plt.ylabel("Loss")
    plt.xlabel("Epochs")
    plt.legend(legend, loc='best')
    plt.show()

  pass

# Base Configuration
number_of_hidden_layers = 2
hidden_layer_width = 100
optimizer = RMSprop()
activation_fn = "tanh"
batch_size = 128
epoch = 10

history, score = train_model(layers=number_of_hidden_layers,
                             hidden_layer_width=hidden_layer_width,
                             optimizer=optimizer,
                             activation_fn=activation_fn,
                             batch_size=batch_size,
                             epochs=epoch)

print_plot([history], "",
           "Epochs", "Accuracy",
            ["Train Accuracy",
             "Validation Accuracy (Test Accuracy: {})".format(score[1])],
             True, True)

history, score = train_model(layers=number_of_hidden_layers,
                             hidden_layer_width=hidden_layer_width,
                             optimizer=optimizer,
                             activation_fn=activation_fn,
                             batch_size=batch_size,
                             epochs=epoch, dropout=True)
```

```
print_plot([history], "",
           "Epochs", "Accuracy",
            ["Train Accuracy",
             "Validation Accuracy (Test Accuracy: {})".format(score[1])],
             True, True)

history, score = train_model(layers=number_of_hidden_layers,
                             hidden_layer_width=hidden_layer_width,
                             optimizer=optimizer,
                             activation_fn=activation_fn,
                             batch_size=batch_size,
                             epochs=epoch, l2_reg=0.001)

print_plot([history], "",
           "Epochs", "Accuracy",
            ["Train Accuracy",
             "Validation Accuracy (Test Accuracy: {})".format(score[1])],
             True, True)

#Layers
history=[]
label=[]
for i in range(2,6):
  [h,s] = train_model(layers=i,
                      hidden_layer_width=hidden_layer_width,
                      optimizer=optimizer,
                      activation_fn=activation_fn,
                      batch_size=batch_size,
                      epochs=epoch)
  history.append(h)

  label.append("No of Hidden Layers "+str(i)+" (Test Accuracy "+str(s[1])+"
)")

print_plot(history, "Variation in No of Hidden Layers",
            " Epochs", "Accuracy", label)

#Activation Function
history=[]
label=[]
for i in ['relu','sigmoid','tanh']:
  [h,s]=train_model(layers=number_of_hidden_layers,
```

```
                    hidden_layer_width=hidden_layer_width,
                    optimizer=optimizer,
                    activation_fn=i,
                    batch_size=batch_size,
                    epochs=epoch)
  history.append(h)
  label.append("Activation Function "+i+" (Test Accuracy " + str(s[1]) + "
)")

print_plot(history, "Variation in Activation Function",
           "Epochs", "Accuracy", label)

#Variation in Epochs
history=[]
label=[]
for i in range(10,41,10):
  [h,s]=train_model(layers=number_of_hidden_layers,
                    hidden_layer_width=hidden_layer_width,
                    optimizer=optimizer,
                    activation_fn=activation_fn,
                    batch_size=batch_size,
                    epochs=i)
  history.append(h)
  label.append("Epochs "+str(i)+" (Test Accuracy "+str(s[1])+" )")

print_plot(history, "Variation in Epochs",
           "Epochs", "Accuracy", label)

#Variation in Batch Size
history=[]
label=[]
i=128
while i < 1025:
  [h,s]=train_model(layers=number_of_hidden_layers,
                    hidden_layer_width=hidden_layer_width,
                    optimizer=optimizer,
                    activation_fn=activation_fn,
                    batch_size=i,
                    epochs=epoch)
  history.append(h)
  label.append("Batch_size "+str(i)+" (Test Accuracy "+str(s[1])+" )")
  i *= 2
```

```
print_plot(history,"Variation in Batch Size"," epochs","acc",label)
```

Convolutional Neural Network

```
from __future__ import print_function

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.regularizers import l2
from keras import backend as K
import matplotlib.pyplot as plt

#Data loaded from mnist dataset available in Keras
num_classes=10
(x_train, y_train), (x_test, y_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, 28, 28)
    x_test = x_test.reshape(x_test.shape[0], 1, 28, 28)
    input_shape = (1, 28, 28)
else:
    x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
    x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
    input_shape = (28, 28, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

def train_model(layers = 1, hidden_layer_width =512,
                activation_fn = 'sigmoid', epochs = 10,
                dropout = False, dropout_factor = 0.2, regularisation = 0.0,
                batch_size = 128, l2_reg = 0.0,
                no_of_kernels = 32, kernel_size = (3,3), dense_layer =
False):
```

```
  model = Sequential()
  model.add(Conv2D(no_of_kernels, kernel_size=kernel_size,
                   activation = activation_fn,
                   input_shape = input_shape, kernel_regularizer =
l2(l2_reg)))
  model.add(MaxPooling2D(pool_size=(2, 2)))

  if layers >= 2:
    model.add(Conv2D(64, kernel_size, activation = activation_fn,
                     kernel_regularizer=l2(l2_reg)))
    model.add(MaxPooling2D(pool_size=(2, 2)))

  if dropout:
    model.add(Dropout(dropout_factor))
  model.add(Flatten())

  if dense_layer:
    model.add(Dense(hidden_layer_width, activation = activation_fn))
    if dropout:
      model.add(Dropout(dropout_factor))

  model.add(Dense(num_classes, activation='softmax'))

  model.compile(loss=keras.losses.categorical_crossentropy,
                optimizer=keras.optimizers.Adadelta(),
                metrics=['accuracy'])

  history = model.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=epochs,
            verbose=1,
            validation_split = 0.1)

  score = model.evaluate(x_test, y_test, verbose=0)
  return history,score

def print_plot(histories, title, xlabel, ylabel, legend,
               print_train = False, print_loss = False):

  for history in histories:
    if print_train:
      plt.plot(history.history['acc'])
    plt.plot(history.history['val_acc'])
```

```
  plt.title(title)
  plt.ylabel("Accuracy")
  plt.xlabel("Epochs")
  plt.legend(legend, loc='best')
  plt.show()

  if print_loss:
    for history in histories:
      if print_train:
        plt.plot(history.history['loss'])
      plt.plot(history.history['val_loss'])
    plt.title(title)
    plt.ylabel("Loss")
    plt.xlabel("Epochs")
    plt.legend(legend, loc='best')
    plt.show()

  pass

# Base Configuration
number_of_hidden_layers = 1
hidden_layer_width =128
activation_fn = 'relu'
epochs =10
batch_size = 128
no_of_kernels = 32
kernel_size = (3,3)

history, score = train_model(layers=number_of_hidden_layers,
                             hidden_layer_width=hidden_layer_width,
                             activation_fn=activation_fn,
                             batch_size=batch_size,
                             epochs=epochs,
                             kernel_size=kernel_size,
                             no_of_kernels=no_of_kernels,)

print_plot([history], "",
           "Epochs", "Accuracy",
            ["Train Accuracy",
             "Validation Accuracy (Test Accuracy: {})".format(score[1])],
             True, True)

history, score = train_model(layers=number_of_hidden_layers,
```

```
                              hidden_layer_width=hidden_layer_width,
                              activation_fn=activation_fn,
                              batch_size=batch_size,
                              epochs=epochs,
                              kernel_size=kernel_size,
                              no_of_kernels=no_of_kernels, dropout=True)

print_plot([history], "",
           "Epochs", "Accuracy",
            ["Train Accuracy",
             "Validation Accuracy (Test Accuracy: {})".format(score[1])],
             True, True)

history, score = train_model(layers=number_of_hidden_layers,
                              hidden_layer_width=hidden_layer_width,
                              activation_fn=activation_fn,
                              batch_size=batch_size,
                              epochs=epochs,
                              kernel_size=kernel_size,
                              no_of_kernels=no_of_kernels, l2_reg=0.001)

print_plot([history], "",
           "Epochs", "Accuracy",
            ["Train Accuracy",
             "Validation Accuracy (Test Accuracy: {})".format(score[1])],
             True, True)

#variation in layers
history=[]
label=[]

[h,s]=train_model(layers=1,
                  hidden_layer_width=hidden_layer_width,
                  activation_fn=activation_fn,
                  batch_size=batch_size,
                  epochs=epochs,
                  kernel_size=kernel_size,
                  no_of_kernels=no_of_kernels,)
history.append(h)
label.append("No of Hidden Layers {} (Test Accuracy: {})".format(1,s[1]))

[h,s]=train_model(layers=2,
                  hidden_layer_width=hidden_layer_width,
```

```
                  activation_fn=activation_fn,
                  batch_size=batch_size,
                  epochs=epochs,
                  kernel_size=kernel_size,
                  no_of_kernels=no_of_kernels,)
history.append(h)
label.append("No of Hidden Layers {} (Test Accuracy: {})".format(2,s[1]))

[h,s]=train_model(layers=3,
                  hidden_layer_width=hidden_layer_width,
                  activation_fn=activation_fn,
                  batch_size=batch_size,
                  epochs=epochs,
                  kernel_size=kernel_size,
                  no_of_kernels=no_of_kernels, dense_layer=True)

history.append(h)
label.append("No of Hidden Layers {} (Test Accuracy: {})".format(3,s[1]))

print_plot(history,"Variation in No of Hidden
Layers","Epochs","Accuracy",label)

#variation in Activation Func
history=[]
label=[]

for i in ['sigmoid','tanh','relu']:
  [h,s]=train_model(layers=number_of_hidden_layers,
                    hidden_layer_width=hidden_layer_width,
                    activation_fn=i,
                    batch_size=batch_size,
                    epochs=epochs,
                    kernel_size=kernel_size,
                    no_of_kernels=no_of_kernels,)
  history.append(h)
  label.append("Activation Function {} (Test Accuracy: {})".format(i,s[1]))

print_plot(history,"Variation in Activation
Function","Epochs","Accuracy",label)

#Variation in epochs
history=[]
label=[]
```

```
for i in range(10,41,10):
  [h,s]=train_model(layers=number_of_hidden_layers,
                    hidden_layer_width=hidden_layer_width,
                    activation_fn=activation_fn,
                    batch_size=batch_size,
                    epochs=i,
                    kernel_size=kernel_size,
                    no_of_kernels=no_of_kernels,)
  history.append(h)
  label.append("Epochs {} (Test Accuracy: {})".format(i,s[1]))

print_plot(history,"Variation in Epochs","Epochs","Accuracy",label)

#Variation in Batch Size
  history=[]
  label=[]

  i=128
  while i<1025:
    [h,s]=train_model(layers=number_of_hidden_layers,
                      hidden_layer_width=hidden_layer_width,
                      activation_fn=activation_fn,
                      batch_size=i,
                      epochs=epochs,
                      kernel_size=kernel_size,
                      no_of_kernels=no_of_kernels,)
    history.append(h)
    label.append("Batch Size {} (Test Accuracy: {})".format(i,s[1]))
    i*=2

print_plot(history,"Variation in Batch Size","Epochs","Accuracy",label)

#Variation in kernel size
history=[]
label=[]

for i in [(3,3), (4,4), (5,5)]:
  [h,s]=train_model(layers=number_of_hidden_layers,
                    hidden_layer_width=hidden_layer_width,
                    activation_fn=activation_fn,
                    batch_size=batch_size,
                    epochs=epochs,
```

```
                    kernel_size=i,
                    no_of_kernels=no_of_kernels)
  history.append(h)
  label.append("Kernel Size {} (Test Accuracy: {})".format(i,s[1]))

print_plot(history,"Variation in Kernel Size","Epochs","Accuracy",label)

#Variation in no. of kernels
history=[]
label=[]

for i in [16, 32, 64]:
  [h,s]=train_model(layers=number_of_hidden_layers,
                    hidden_layer_width=hidden_layer_width,
                    activation_fn=activation_fn,
                    batch_size=batch_size,
                    epochs=epochs,
                    kernel_size=kernel_size,
                    no_of_kernels=i)
  history.append(h)
  label.append("No. of Kernels {} (Test Accuracy: {})".format(i,s[1]))

#@title
print_plot(history,"Variation in No. of Kernels","Epochs","Accuracy",label)
```