Group 27
CVML Assignment 1

MNIST Digits Classification

Nitin Kedia 160101048
Ashish Ranjan 160123006
Mitanshu Mittal 160123020
Shivam Kumar 160101066
Sparsh Bansal 160101072

Language and Libraries used

Programming language: Python 3
LibrariesL

1. We are using Python's Scikit-learn library which provides an implementation of Logistic Regression classifier.
2. Used Keras for building all neural network models. Keras uses Tensorflow and Theano libraries in its backend.
3. Matplotlib for graph plotting. Jupyter notebook for hands-on graph use.

Dataset

The MNIST data consists of a total of 70000 images each with a label which are split into two different datasets for training and testing.The training data set contains 60,000 images and test data set contains 10,000 images.The images are grey scale and 28 x 28 pixels in size. Each image is transformed to one dimensional vector of size (28x28=)784 except CNN.

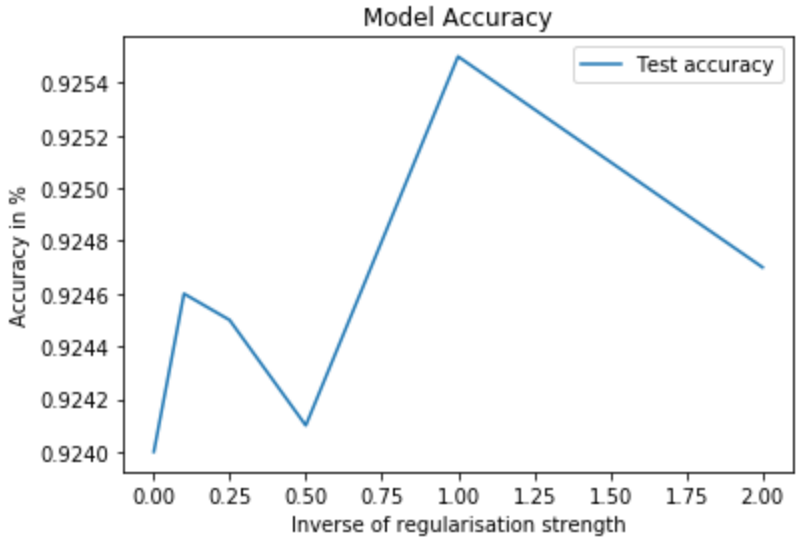
Logistic Regression

Our default parameters to LR model were the were: solver: *lgfgs*, Maximum iterations: 100, Inverse regularisation strength: 1.0

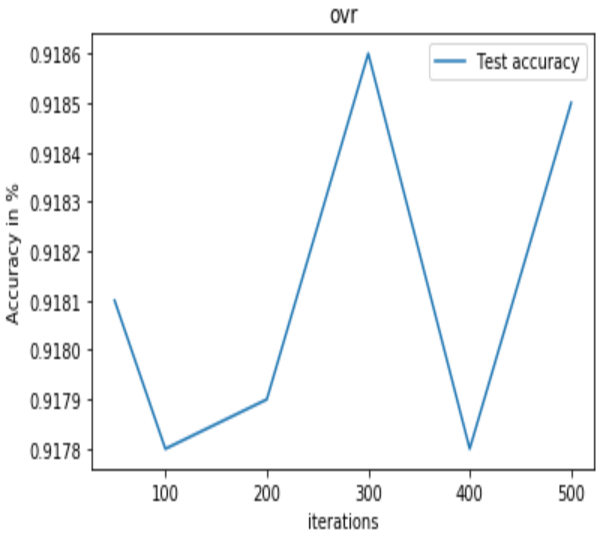
Variations

1. **L1 Regularisation Strength**

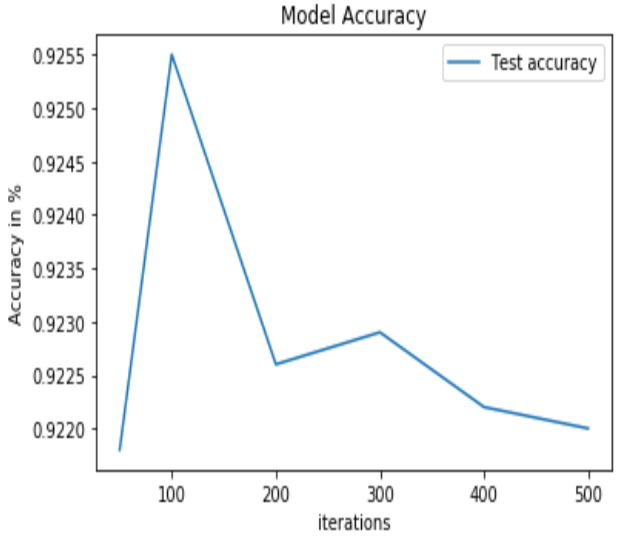
   L1 Regularisation penalises weights with high absolute values to prevent overfitting. It adds sum of absolute weights to the cost function to drive out higher weights. More Regularisation strength gives more weightage to regularisation part in cart function. We see that the model accuracy reaches its maximum when the regularisation parameter C is 1.0. On either side of 1.0 model accuracy is lesser.



2. **Accuracy vs Iteration**



Ovr



Multinomial

We varied number of iterations over [50, 100, 200, 300, 400, 500] over 2 types of classifiers OVR and multi_class. The accuracy values in both cases was varies very slightly about 0.1%.
Multinomial classifier had better average accuracy than OVR.
In one-vs-rest (OvR) the maximum accuracy is obtained at 300 iterations while in multinomial it is at 100 iterations.

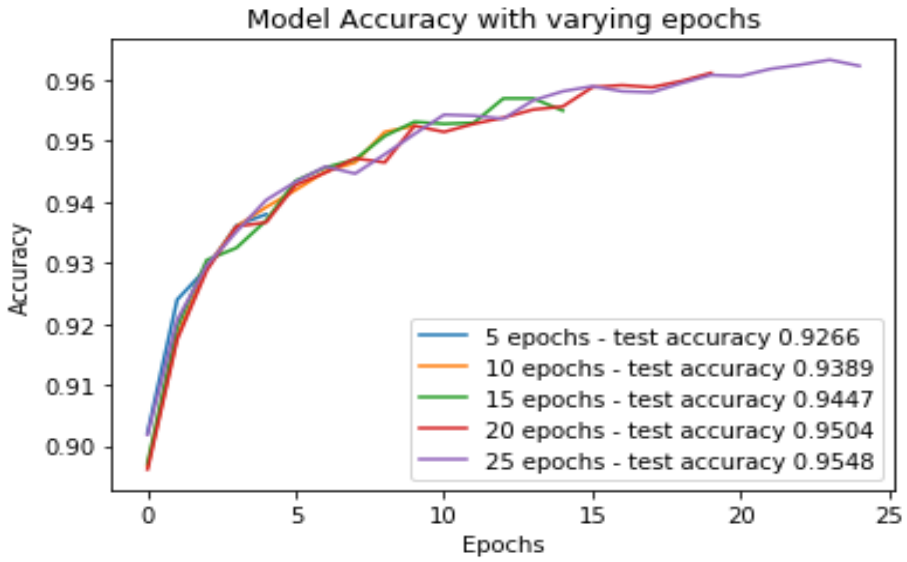
Multi Layer Perceptron

The architecture for perceptron is as follows:

1. We have 1 hidden layer, the input and the output layer.
2. Hidden layer has 256 nodes. Input layer has 784 nodes corresponding to each of the pixels in the 28 * 28 image from MNIST. Output layer has 10 nodes which will have one-hot encoding since we have 10 digits 0-9.
3. In the hidden layer *sigmoid* activation function is used. The output layer has *softmax* activation which modifies each activation (node values) of the layer to lie between 0 and 1 and total sum 1, enabling us to interpret it as a probability.
4. Default epochs is 15 with each epoch being a full training pass over the entire dataset such that each training example has been seen once.

Variations

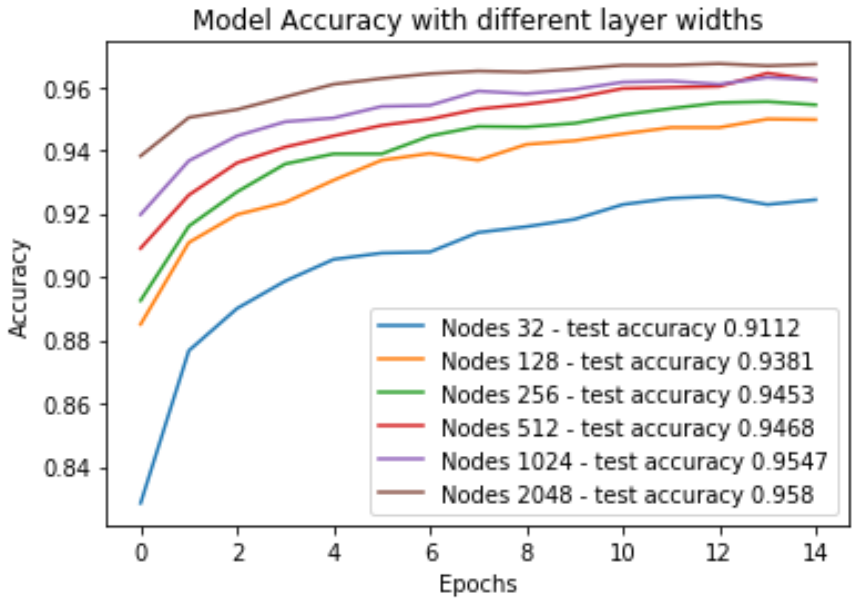
1. **Epochs**
   We vary the epochs after which the model is tested.



Our base model has 2 layers and needs 5 epochs to be reach 92.66% test accuracy (the highest being 95.48% at 25 epochs) and then we get diminishing returns, ~3% increase in accuracy after

20 more epochs.

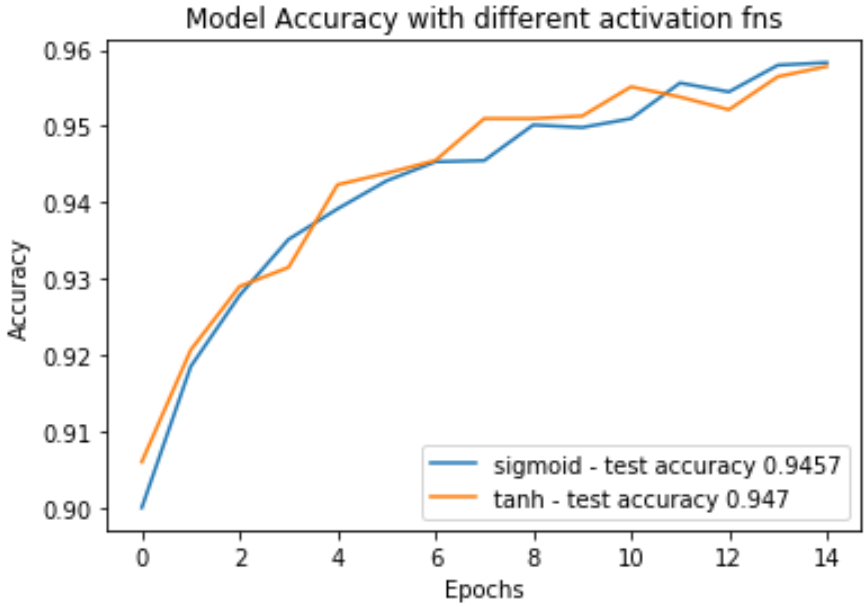
2. **Layer Width**



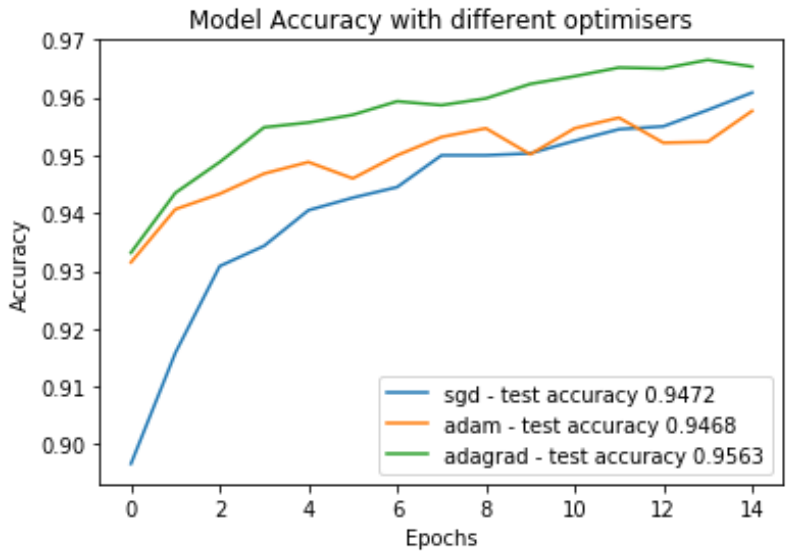
Accuracy increases with increase in number of nodes ranging from 91.12% at 32 nodes to 95.8% at 2048 nodes. Gain in accuracy is marginal even on doubling the number of nodes each time.

3. **Activation Functions**

We tested sigmoid and tanh in the (single) hidden layer. The output layer always uses softmax. Tanh wins by a small margin. The range of tanh is [-1,1] but that of sigmoid is [0,1]. Tanh is thus more flexible as it provides neurons to assume a strongly no state. But MLP has only 1 layer, the effect of tanh is more pronounced in more layers.



4. **Optimiser Function**
   The network aims to find parameters that minimise cost i.e. deviation from actual answer. For this we use Gradient Descent to navigate the derivative of cost function wrt. parameters to find a minima. An optimiser is a specific implementation of gradient descent.
   SGD: Use a single example to update parameters. *adagrad*: It rescales the gradients of each parameters effectively giving each parameter its own learning rate. Also frequently updating parameters get updated in slower increments as compared to rarely occurring ones.
   adam: Like adagrad, it has an adaptive learning rate for each parameter. It behaves like a heavy ball with friction.



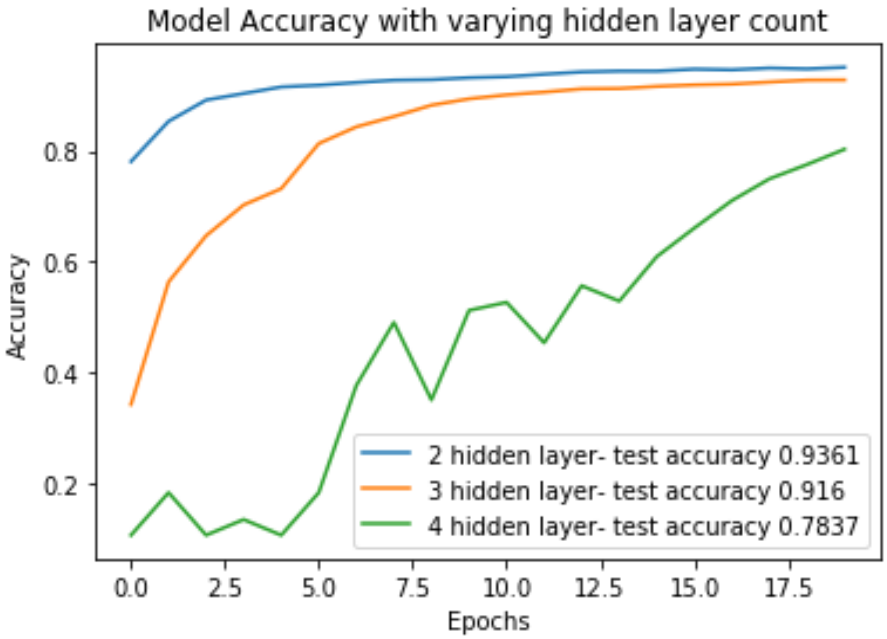
Deep Neural Network

Our base architecture is outlined below:

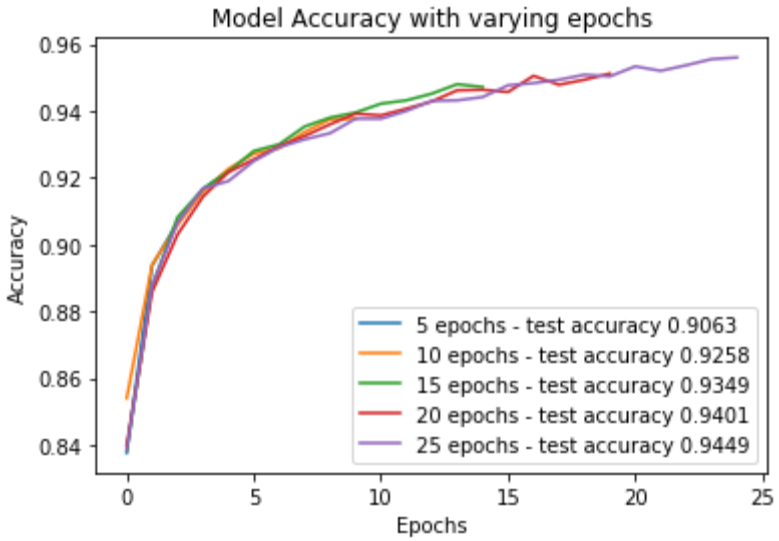
1. We have 2 hidden layers plus the input and the output layer.
2. Each hidden layer has 256 nodes. Input layer has 784 nodes corresponding to each of the pixels in the 28 * 28 image from MNIST. Output layer has 10 nodes which will have one-hot encoding since we have 10 digits 0-9. The image is flattened, so we lose spatial information.
3. Nodes between two adjacent layers are fully connected. Thus between the two hidden layers there are 256 * 256 parameters (edges).
4. In each hidden layer *sigmoid* activation function is default. The final layer has *softmax* activation which modifies each activation (node values) of the layer to lie between 0 and 1 and total sum 1, enabling us to interpret it as a probability.
5. Default epochs is 25 with each epoch being a full training pass over the entire dataset such that each example has been seen once. Thus, an epoch represents N/batch size training iteration, where N is the total number of examples: N = 54,000 in MNIST. Remaining consists of 6,000 images for validation run after each epoch and 10,000 images for final testing.

Variations

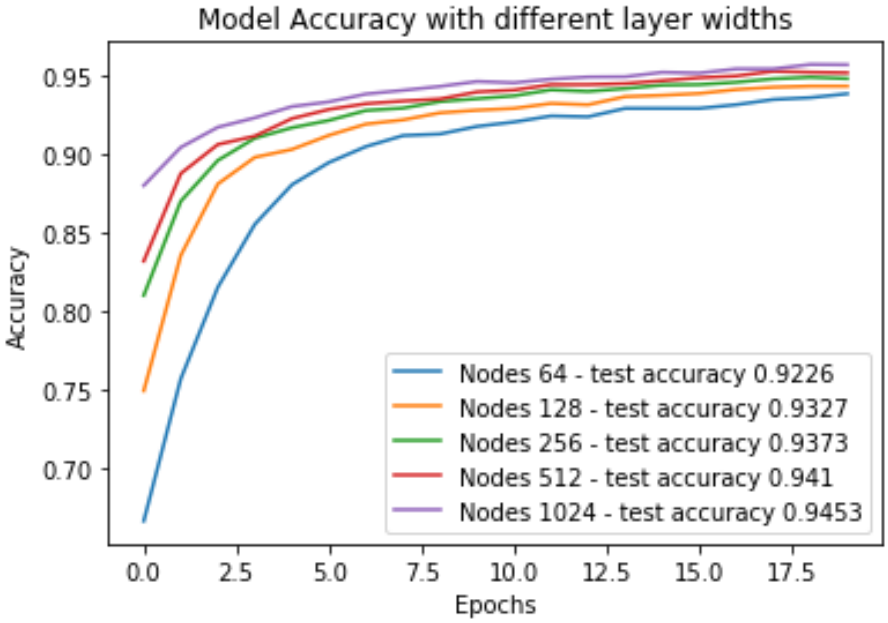
1. **Hidden Layer Count**
   We increase the network depth from 2 to 4. Plotted below is the validation accuracy determined after each epoch (our validation split is 10% of 60,000 training data). The model with 2 hidden layers flatlines after 5 epochs, the one with three layers after 10 epochs while the model with most depth is still learning after 10 epochs. More layers means more parameters means the training time increases substantially. Each forward pass and back-propagation becomes slower.



2. **Epochs**
   We vary the epochs after which the model is tested. Our base model has 2 layers and needs between 5-10 epochs to be reach 90.63% test accuracy (the highest being 94.49%) and then we get diminishing returns.

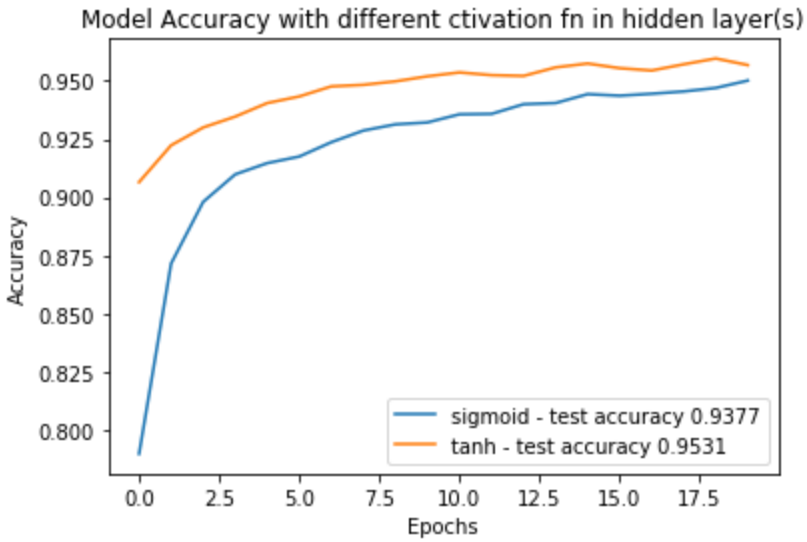


3. **Layer Widths**
   Increasing nodes in a layer drastically increases the number of parameters (edges) since each incremental node needs to be connected to every node in previous layer. This enables the model to hold more data, indeed our accuracy is 94.53% with 1024 nodes per layer, 92.26% with 64 nodes per layer. The test accuracy is increasing but we are doubling nodes for marginal gains.



4. **Activation Function**

   We can see that the learning rate of the Sigmoid function is slow because Sigmoid outputs are not zero centred. It has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive, then the gradient on the weights will become either all be positive, or all negative (depending on the gradient of the whole expression) during backpropagation. This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, it is noticed that once these gradients are added up across a batch of data, the final update for the weights can have variable signs, somewhat mitigating this issue (we can see that it is converging to other activation functions).



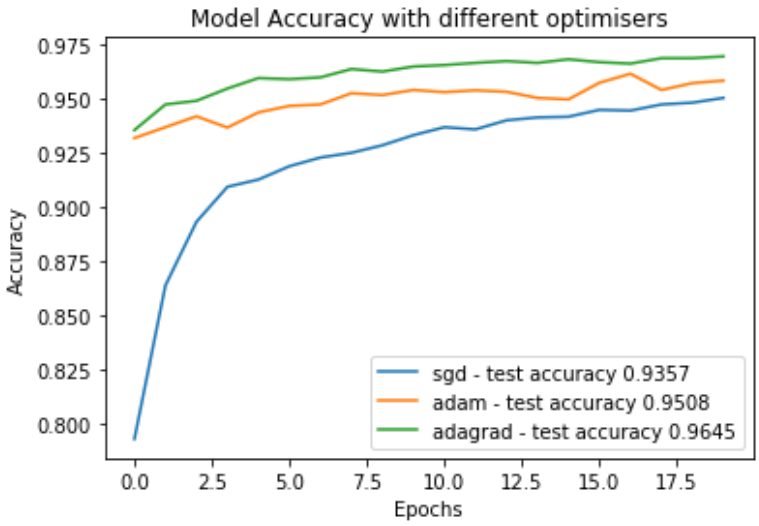
5. **Optimiser Function**

   We use 3 different optimiser functions namely: *SGD, adam, adagrad.*

   The NN aims to find parameters that minimise cost i.e. deviation from actual answer. For this we use gradient descent to navigate the derivative of cost function wrt. parameters to find a minima. An optimiser is a specific implementation of gradient descent.

   SGD: Use a single example to update parameters. *adagrad*: It rescales the gradients of each parameters effectively giving each parameter its own learning rate. Also, frequently updating parameters get updated in slower increments as compared to rarely occurring ones.

   adam: Like adagrad, it has an adaptive learning rate for each parameter. It behaves like a heavy ball with friction.



6. Overfitting

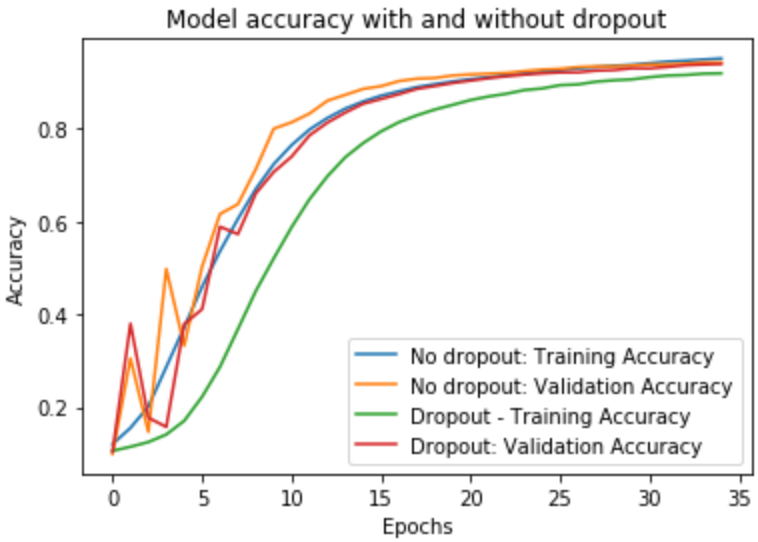
Neural networks are flexible enough that given enough parameters and training they can adjust to the training data so precisely that it cannot generalise to (unseen) test data. This is memorising the answers of training data and thus failing in test data. We use two approaches to combat it:

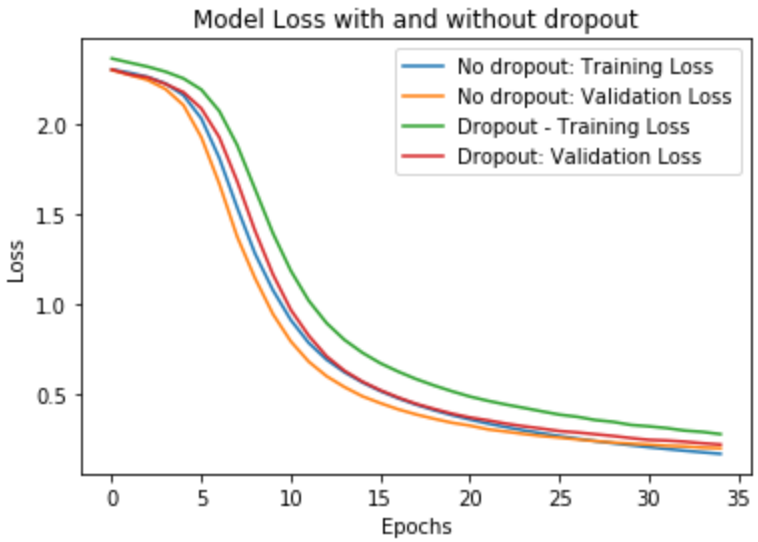
a. Dropout

The network is modified during training to randomly drop nodes and carry out feed-forward and back-propagation on that. Intuitively, this inhibits reliance of the network on specific nodes to memorise specific parts of the training and to generalise better.

We first increased both the network depth, width and epochs so that it is more prone to overfitting. Then we added dropout to see if it helps. Indeed without dropout towards the end training accuracy surpasses validation accuracy which means it overfitted but with dropout the validation accuracy maintains a gap over training accuracy.

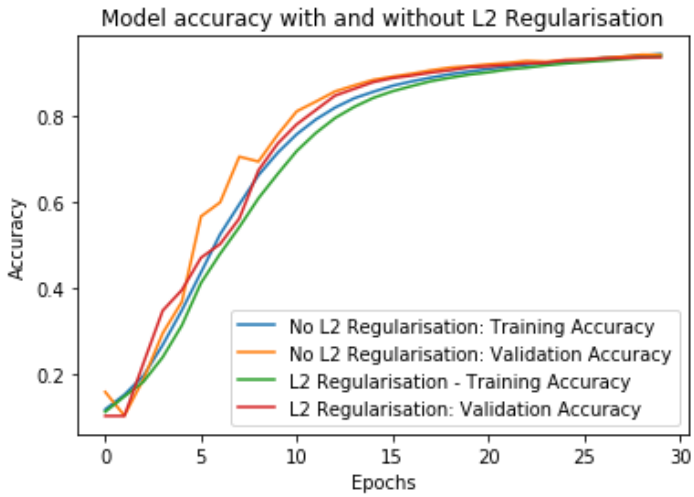
Similarly in the loss graph, if we overfit the training loss will almost reach zero. This is not the case with dropout.

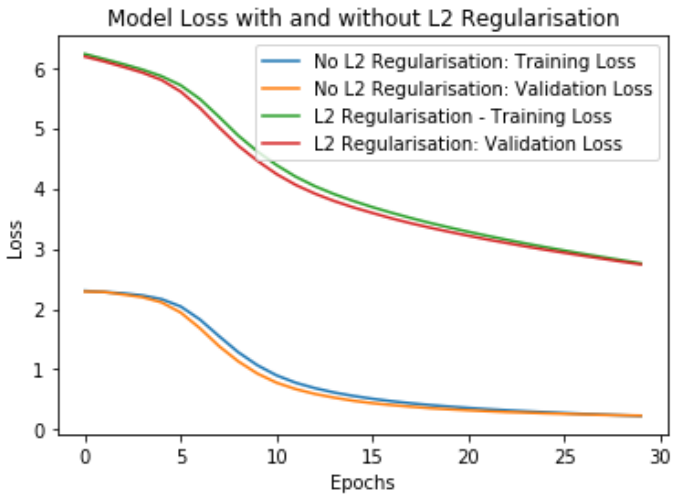




b. L2 Regularisation

It penalizes weights in proportion to the sum of the squares of the weights. This helps drive outlier weights (those with high positive or low negative values) closer to 0 but not quite to 0. Without regularisation the training loss is much closer to 0 meaning that the NN is memorising training data but with it the NN is not able to exceptionally fit with training data.





Deep Convolutional Neural Network

Base architecture :

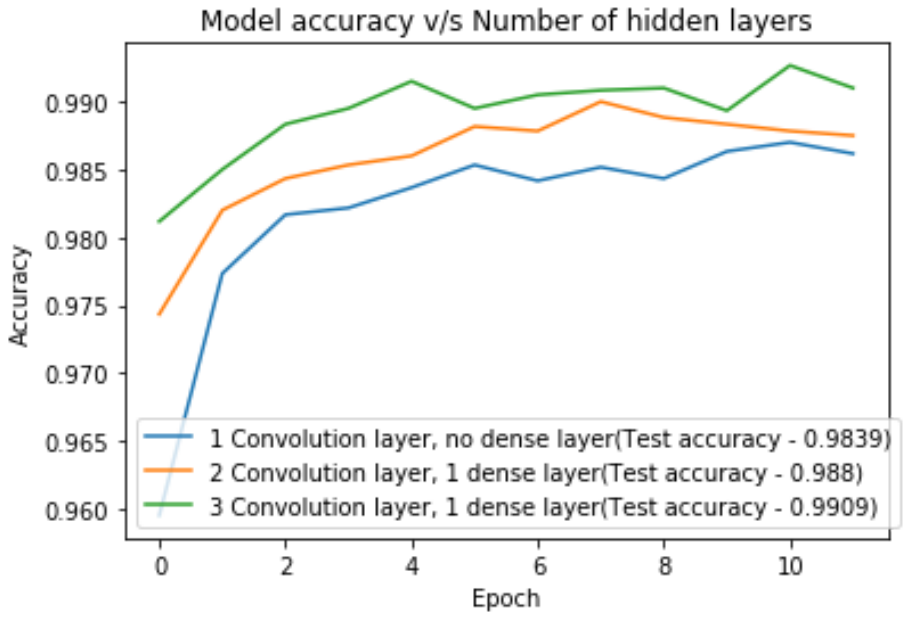
1. Here we don't flatten the image matrix. Hidden layer description:
    a. Convolution layer of size 3x3 with 32 filters.
    b. 2-D max-pooling layer of size 2x2
    c. Dense layer with 128 nodes
2. In each hidden layer, *ReLU* activation function is used. The final layer has *softmax* activation which modifies each activation (node values) of the layer to lie between 0 and 1 and total sum 1, enabling us to interpret it as a probability.
3. Default epochs are 12 with each epoch being a full training pass over the entire dataset such that each example has been seen once. Thus, an epoch represents N/batch size training iteration, where N is the total number of examples: N = 54,000 in MNIST. Remaining consists of 6,000 images for a validation run after each epoch and 10,000 images for final testing. Batch size is chosen as 128.

Variations

1. **Number of hidden layers**

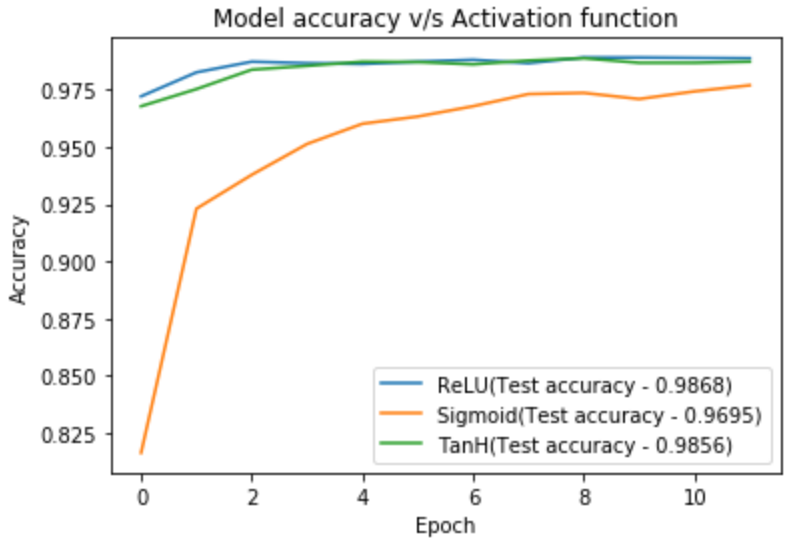
    We observe the accuracy with respect to three different sets of hidden layers:
    1. 1 convolutional layer of size 3x3 with 32 filters (without dense layer).
    2. Same convolution with an additional dense layer of size 128.
    3. 2 convolutional layers with sizes 3x3 and 32 and 64 filters respectively.

    As expected as the number of layers increases, the complexity of neural network increases which gives the network more power to compute more complex functions which result in an increase in the model accuracy.



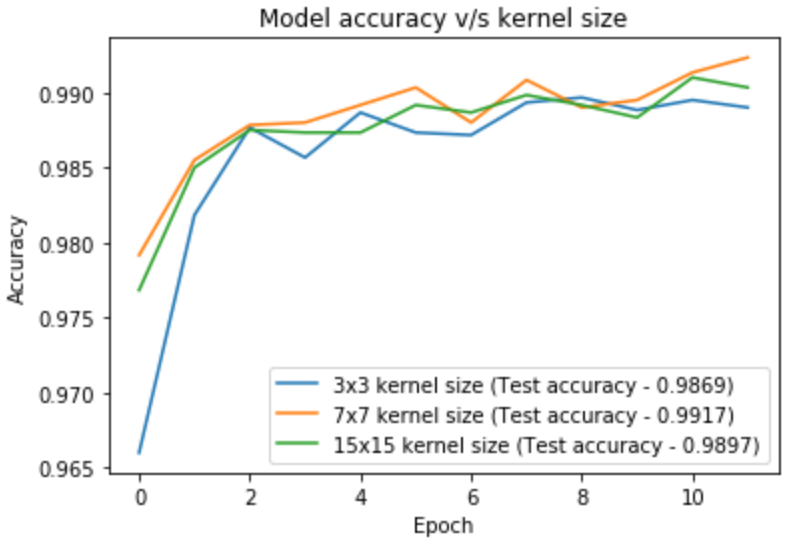
2. **Activation Function**

We observe the accuracy with respect to different activation function in the hidden layers namely ReLU, Sigmoid and Tanh functions.We can see that the learning rate of the Sigmoid function is slow because Sigmoid outputs are not zero centred. It has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive, then the gradient on the weights w will become either all be positive, or all negative (depending on the gradient of the whole expression) during backpropagation. This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue (we can see that it is converging to other activation functions).



3. **Kernel Size**

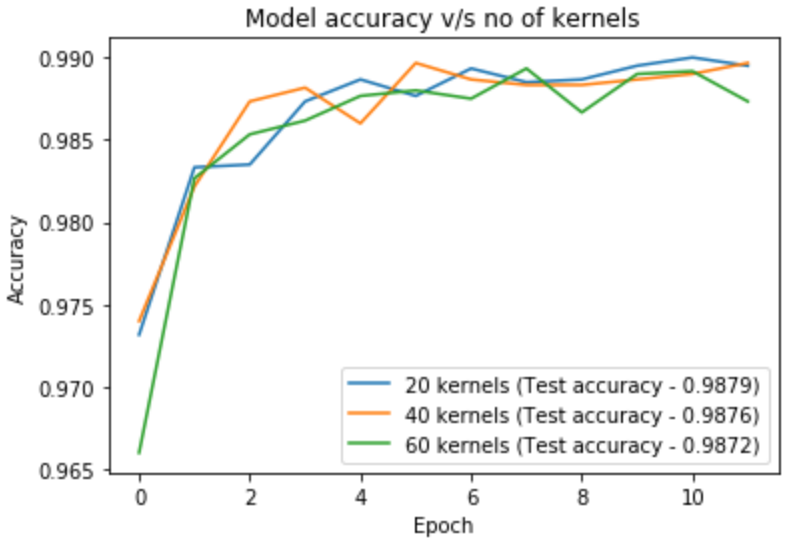
We observed our model on three different kernel sizes 3x3, 7x7 and 15x15.
It can be seen that accuracy is increasing as kernel size increases, but this is not the general case. Small kernel size can lead to too much information and can result in a highly local image vision without image overview. Larger kernel size can overlook the complex image features and could skip the essential details in the images. Thus the determination of a good kernel size is needed. In our case, 7x7 seems to be a good choice.



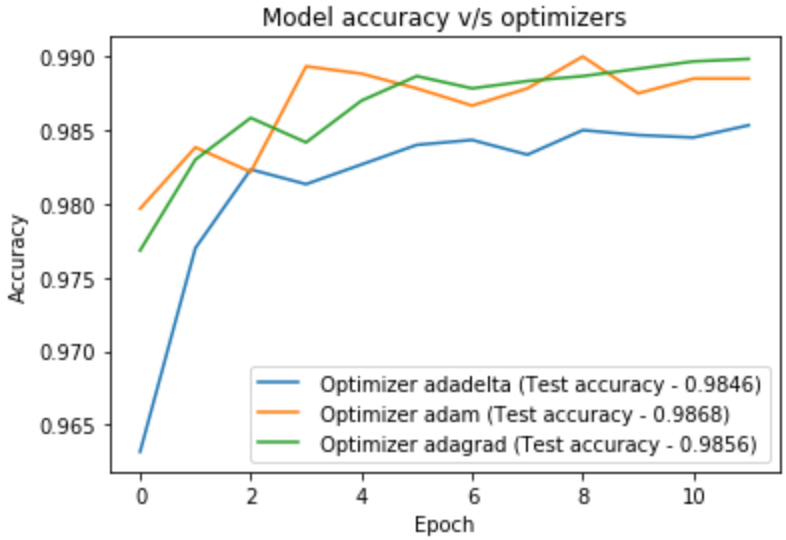
4. **Number of Kernels**

   We observed our data on 20, 40 and 60 kernels.

   The number of feature detectors intuitively is the number of features (like edges, lines, object parts etc) that the network can potentially learn. Each filter generates a feature map. Feature maps allow CNN to learn the explanatory factors within the image, so more filters mean the more the network learns (not necessarily good all the time - we have overfitting).. If we use more number of kernels than relevant features than the remaining kernels will be used as a breathing space, so some of them may be redundant or slightly different. This can be observed from our graph as well, the performance didn't differ much upon increasing the number of kernels.



5. **Optimizer Function**



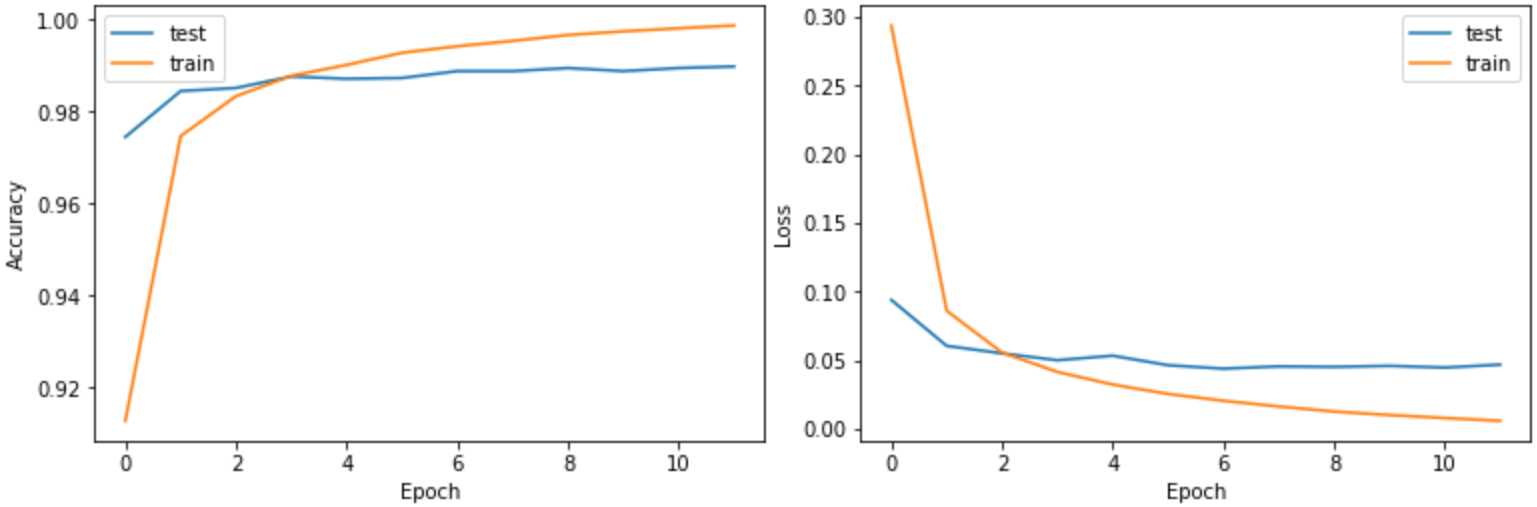
   We used different optimizer function to do the gradient descent convergence faster. The model is observed on adadelta, adam and adagrad optimizers.

   We found that test accuracy is highest for adadelta while it has lowest validation accuracy.

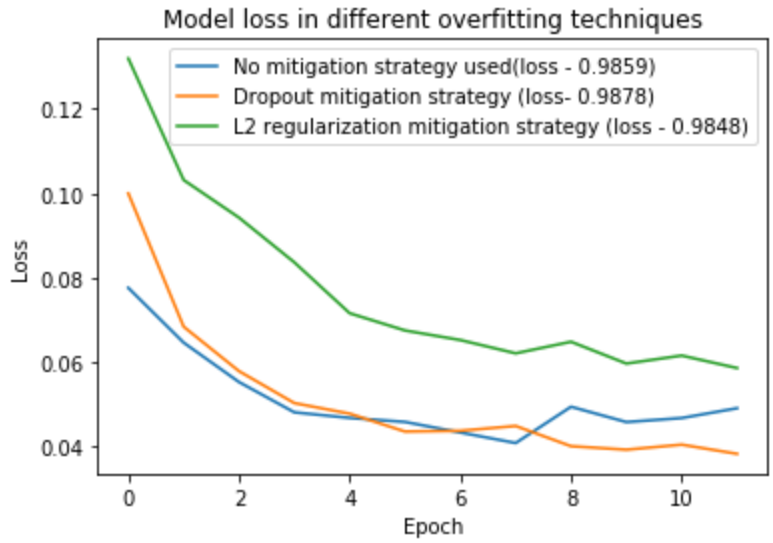
   adagrad: It rescales the gradients of each parameter effectively giving each parameter its own

learning rate. Also, frequently updating parameters get updated in slower increments as compared to rarely occurring ones.
adam: Like adagrad, it has an adaptive learning rate for each parameter. It behaves like a heavy ball with friction.
adadelta : Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done.

6. **Overfitting**



From the above figures, it is clear that our base model suffers from overfitting as train accuracy and loss are way more favourable than the test one.
We can tackle this problem by using different mitigation strategies, here we have explored dropout and l2 regularisation which we explained in deep neural network case. Note that we apply dropout on only dense layers in network and not convolutional layers. This is because shared weights in convolutions mean that convolutional filters are forced to learn from across entire image. This makes them less likely to pick up on local specifics in training data.
Using dropout helped us in reducing the overfitting problem effectively.



Logistic Regression Code:

```
# -*- coding: utf-8 -*-
"""LogisticRegression.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1nVmXNSDf4L0kx272zYfOyG2jDkj2LH-i
"""

# Commented out IPython magic to ensure Python compatibility.
# %matplotlib inline
import keras
from keras.datasets import mnist
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.linear_model import LogisticRegression



#load mnist dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data() #everytime loading data won't be so easy :)

# Print to show there are 1797 images (8 by 8 images for a dimensionality of 64)
print("Image Data Shape" , x_train.shape)

# Print to show there are 1797 labels (integers from 0-9)
print("Label Data Shape", y_train.shape)

import numpy as np
plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(x_train[0:5], y_train[0:5])):
    plt.subplot(1, 5, index + 1)
    plt.imshow(image, cmap=plt.cm.gray)
    plt.title('Training: %i\n' % label, fontsize = 20)

print(x_train.shape[0],x_train.shape[1]*x_train.shape[2])
x_train = x_train.reshape(x_train.shape[0],x_train.shape[1]*x_train.shape[2])
x_test = x_test.reshape(x_test.shape[0],x_test.shape[1]*x_test.shape[2])

def logisticModel(solver = 'lbfgs',multi_class = 'auto' , max_iter  =100, C = 1.0,verbose = 1):
    logisticRegr = LogisticRegression(solver = solver,multi_class = 'ovr',max_iter  = max_iter,C=C,verbose =
1)
```

```
    return logisticRegr

logisticRegr = logisticModel()
logisticRegr.fit(x_train,y_train)
y_pred = logisticRegr.predict(x_test)

score = logisticRegr.score(x_train,y_train)
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

print(score)

iters = [50,100,200,300,400,500]
ans=[]
ans2 = []
print (iters)
for x in iters:
    logisticRegr = logisticModel(max_iter = x)
    logisticRegr.fit(x_train, y_train)
    score = logisticRegr.score(x_test, y_test)
    ans.append(score)
print(ans)

plt.plot(iters,ans)
legend = [];
legend.append('Test accuracy')
plt.xlabel('iterations')
plt.ylabel('Accuracy in %')
plt.title('Model Accuracy')
plt.legend(legend, loc='best')
```

Perceptron Code:

```
# -*- coding: utf-8 -*-
"""Perceptron.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1XFj-JYobSWx5elzKkgS107xrMNTt_1V_
"""


import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.regularizers import l2
from matplotlib import pyplot
from random import randint

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Making a copy before flattening for the next code-segment which displays images
x_train_drawing = x_train

image_size = 784 # 28 x 28
x_train = x_train.reshape(x_train.shape[0], image_size)
x_test = x_test.reshape(x_test.shape[0], image_size)

# Convert class vectors to binary class matrices
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

for i in range(64):
    ax = pyplot.subplot(8, 8, i+1)
    ax.axis('off')
    pyplot.imshow(x_train_drawing[randint(0, x_train.shape[0])], cmap='Greys')
```

```
def build_model(layers = 1, layer_sizes=[256] * 1, batch_size=128, epochs=15,
            activation_fn='sigmoid', optimiser_fn='sgd',
            add_dropout=False, dropout_p=0.2, l2_reg=0.0):
   model = Sequential()
   # range(a, b) is exclusive of b
   for l in range(0, layers):
      if (l == 0):
         model.add(Dense(layer_sizes[l], kernel_regularizer=l2(l2_reg), activation=activation_fn,
input_shape=(image_size,)))
      else:
         model.add(Dense(layer_sizes[l], kernel_regularizer=l2(l2_reg), activation=activation_fn))
      if add_dropout:
         model.add(Dropout(dropout_p))

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

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

   history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
                  verbose=True, validation_split=0.1)
   loss, accuracy  = model.evaluate(x_test, y_test, verbose=False)
   return history, loss, accuracy

# Test number of epochs
legend = []
for epoch_count in range(5, 30, 5):
   history, loss, accuracy = build_model(epochs=epoch_count)
   pyplot.plot(history.history['val_acc'])
   legend.append('{} epochs - test accuracy {}'.format(epoch_count, accuracy))

pyplot.title('Model Accuracy with varying epochs')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# Test different layer widths
legend.clear()
pyplot.close()
for nodes in [32, 128, 256, 512, 1024, 2048]:
   history, loss, accuracy = build_model(1, [nodes])
```

```
    pyplot.plot(history.history['val_acc'])
    legend.append('Nodes {} - test accuracy {}'.format(nodes, accuracy))
    print('Nodes {} - test accuracy {}'.format(nodes, accuracy))

    pyplot.title('Model Accuracy with different layer widths')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# Test different activation fn in hidden layer(s)
legend = []
pyplot.close()
for fn in ['sigmoid', 'tanh']:
    history, loss, accuracy = build_model(activation_fn=fn)
    pyplot.plot(history.history['val_acc'])
    legend.append('{} - test accuracy {}'.format(fn, accuracy))

pyplot.title('Model Accuracy with different activation fns')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# Test different optimisation functions
legend = []
pyplot.close()
for fn in ['sgd', 'adam', 'adagrad']:
    history, loss, accuracy = build_model(optimiser_fn=fn)
    pyplot.plot(history.history['val_acc'])
    legend.append('{} - test accuracy {}'.format(fn, accuracy))

pyplot.title('Model Accuracy with different optimisers')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()
```

Neural Network Code:

```
# -*- coding: utf-8 -*-
"""NeuralNetwork.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1zQHv_gmFFImO_m4bCxgtJz3IR4RCoi-m
"""


import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.regularizers import l2
from matplotlib import pyplot
from random import randint

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Making a copy before flattening for the next code-segment which displays images
x_train_drawing = x_train

image_size = 784 # 28 x 28
x_train = x_train.reshape(x_train.shape[0], image_size)
x_test = x_test.reshape(x_test.shape[0], image_size)

# Convert class vectors to binary class matrices
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

for i in range(64):
    ax = pyplot.subplot(8, 8, i+1)
    ax.axis('off')
    pyplot.imshow(x_train_drawing[randint(0, x_train.shape[0])], cmap='Greys')
```

```
def build_model(layers = 2, layer_sizes=[256] * 2, batch_size=128, epochs=20,
            activation_fn='sigmoid', optimiser_fn='sgd',
            add_dropout=False, dropout_p=0.2, l2_reg=0.0):
   model = Sequential()
   # range(a, b) is exclusive of b
   for l in range(0, layers):
      if (l == 0):
         model.add(Dense(layer_sizes[l], kernel_regularizer=l2(l2_reg), activation=activation_fn,
input_shape=(image_size,)))
      else:
         model.add(Dense(layer_sizes[l], kernel_regularizer=l2(l2_reg), activation=activation_fn))
      if add_dropout:
         model.add(Dropout(dropout_p))

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

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

   history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
                  verbose=True, validation_split=0.1)
   loss, accuracy  = model.evaluate(x_test, y_test, verbose=False)
   return history, loss, accuracy

# Test different optimisation functions
legend = []
pyplot.close()
for fn in ['sgd', 'adam', 'adagrad']:
   history, loss, accuracy = build_model(optimiser_fn=fn)
   pyplot.plot(history.history['val_acc'])
   legend.append('{} - test accuracy {}'.format(fn, accuracy))

pyplot.title('Model Accuracy with different optimisers')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# Test number of epochs
legend = []
pyplot.close()
for epoch_count in range(5, 30, 5):
```

```
    history, loss, accuracy = build_model(epochs=epoch_count)
    pyplot.plot(history.history['val_acc'])
    legend.append('{} epochs - test accuracy {}'.format(epoch_count, accuracy))

pyplot.title('Model Accuracy with varying epochs')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# Test Number of hidden layers
legend.clear()
pyplot.close()
for l in range(2, 5):
    history, loss, accuracy = build_model(l, [256] * l)
    pyplot.plot(history.history['val_acc'])
    legend.append('{} hidden layer- test accuracy {}'.format(l, accuracy))

pyplot.title('Model Accuracy with varying hidden layer count')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# Test different layer widths
legend.clear()
pyplot.close()
for nodes in [32, 128, 256, 512, 1024, 2048]:
    history, loss, accuracy = build_model(2, [nodes] * 2)
    pyplot.plot(history.history['val_acc'])
    legend.append('Nodes {} - test accuracy {}'.format(nodes, accuracy))
    print('Nodes {} - test accuracy {}'.format(nodes, accuracy))

    pyplot.title('Model Accuracy with different layer widths')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# Test different activation fn in hidden layer(s)
legend.clear()
pyplot.close()
```

```
for fn in ['sigmoid', 'tanh']:
    history, loss, accuracy = build_model(activation_fn=fn)
    pyplot.plot(history.history['val_acc'])
    legend.append('{} - test accuracy {}'.format(fn, accuracy))

pyplot.title('Model Accuracy with different ctivation fn in hidden layer(s)')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# L2 Regularisation Accuracy
legend = []
pyplot.close()

history, loss, accuracy = build_model(layers=4, layer_sizes=[1024] * 4, epochs=30)
pyplot.plot(history.history['acc'])
legend.append('No L2 Regularisation: Training Accuracy')
pyplot.plot(history.history['val_acc'])
legend.append('No L2 Regularisation: Validation Accuracy')
print('Test Accuracy - {}'.format(accuracy))

history, loss, accuracy = build_model(layers=4, layer_sizes=[1024] * 4, epochs=30,l2_reg=0.001)
pyplot.plot(history.history['acc'])
legend.append('L2 Regularisation - Training Accuracy')
pyplot.plot(history.history['val_acc'])
legend.append('L2 Regularisation: Validation Accuracy')
print('Test Accuracy - {}'.format(accuracy))

pyplot.title('Model accuracy with and without L2 Regularisation')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# L2 Regularisation Loss
legend = []
pyplot.close()

history, loss, accuracy = build_model(layers=4, layer_sizes=[1024] * 4, epochs=30)
pyplot.plot(history.history['loss'])
legend.append('No L2 Regularisation: Training Loss')
```

```
pyplot.plot(history.history['val_loss'])
legend.append('No L2 Regularisation: Validation Loss')


history, loss, accuracy = build_model(layers=4, layer_sizes=[1024] * 4, epochs=30,l2_reg=0.001)
pyplot.plot(history.history['loss'])
legend.append('L2 Regularisation - Training Loss')
pyplot.plot(history.history['val_loss'])
legend.append('L2 Regularisation: Validation Loss')

pyplot.title('Model Loss with and without L2 Regularisation')
pyplot.ylabel('Loss')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# Dropout Loss
legend.clear()
pyplot.close()

history, loss, accuracy = build_model(layers=4, layer_sizes=[1024] * 4, epochs=35)
pyplot.plot(history.history['loss'])
legend.append('No dropout: Training Loss')
pyplot.plot(history.history['val_loss'])
legend.append('No dropout: Validation Loss')


history, loss, accuracy = build_model(layers=4, layer_sizes=[1024] * 4, epochs=35, add_dropout=True,
dropout_p=0.2)
pyplot.plot(history.history['loss'])
legend.append('Dropout - Training Loss')
pyplot.plot(history.history['val_loss'])
legend.append('Dropout: Validation Loss')

pyplot.title('Model Loss with and without dropout')
pyplot.ylabel('Loss')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()

# Dropout Accuracy
legend.clear()
```

```
pyplot.close()

history, loss, accuracy = build_model(layers=4, layer_sizes=[1024] * 4, epochs=35)
pyplot.plot(history.history['acc'])
legend.append('No dropout: Training Accuracy')
pyplot.plot(history.history['val_acc'])
legend.append('No dropout: Validation Accuracy')


history, loss, accuracy = build_model(layers=4, layer_sizes=[1024] * 4, epochs=35, add_dropout=True,
dropout_p=0.2)
pyplot.plot(history.history['acc'])
legend.append('Dropout - Training Accuracy')
pyplot.plot(history.history['val_acc'])
legend.append('Dropout: Validation Accuracy')

pyplot.title('Model accuracy with and without dropout')
pyplot.ylabel('Accuracy')
pyplot.xlabel('Epochs')
pyplot.legend(legend, loc='best')
pyplot.show()
```

Convolutional Neural Network (Preliminary Plots):

```
# -*- coding: utf-8 -*-
"""cnn_mnist.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1Ml_4ExEuZEwq2WtlzfpbHm6M54QA12K4
"""


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 import backend as K
import matplotlib.pyplot as plt

batch_size = 128
num_classes = 10
epochs = 12

# input image dimensions
img_rows, img_cols = 28, 28

# the data, split between train and test sets
(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, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

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

```
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# 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)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
# model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
# model.add(Dropout(0.5))
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=.1)
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
plt.plot(history.history['val_acc'])
 plt.plot(history.history['acc'])
 plt.legend(['test','train'],loc='best')
 plt.ylabel('Accuracy')
 plt.xlabel('Epoch')
plt.plot(history.history['val_loss'])
 plt.plot(history.history['loss'])
 plt.legend(['test','train'],loc='best')
 plt.ylabel('Loss')
 plt.xlabel('Epoch')
```

Convolutional Neural Network Test:

```
# -*- coding: utf-8 -*-
"""ConvolutionalNeuralNetwork.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1jJX72bKRW60L4MsOf9gPapzrNjzJ54wZ
"""

import os
from os.path import dirname, abspath

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

# __file__ = "cnn.txt"

PROJECT_DIR = abspath('')
PLOTS_DIR = os.path.join(PROJECT_DIR, "plots")
if not os.path.exists(PLOTS_DIR):
    os.makedirs(PLOTS_DIR)

img_rows, img_cols = 28, 28
epochs = 12
batch_size = 128
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, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
#     x_train = x_train[1:10]
#     x_test = x_test[1:10]
    input_shape = (1, img_rows, img_cols)
else:
```

```
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
#     x_train = x_train[1:10]
#     x_test = x_test[1:10]
    input_shape = (img_rows, img_cols, 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 create_model(no_of_layers,no_of_kernels,kernel_size,dense,dropout,active_func='relu',l2_reg =0.0):
    model = Sequential()
    model.add(Conv2D(no_of_kernels[0], (kernel_size[0],kernel_size[0]),
                     activation=active_func,
                     input_shape=input_shape, kernel_regularizer=l2(l2_reg)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    if no_of_layers > 1:
        for i in range(1,no_of_layers):
            model.add(Conv2D(no_of_kernels[i], (kernel_size[i],kernel_size[i]), activation=active_func,
kernel_regularizer=l2(l2_reg)))
            model.add(MaxPooling2D(pool_size=(2, 2)))
    if dropout:
        model.add(Dropout(0.25))
    model.add(Flatten())
    if dense:
        model.add(Dense(128, activation=active_func))
        if dropout:
            model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))

    return model

def evaluate(model,optimizer_func='adadelta'):
    model.summary()
    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=optimizer_func,
                  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)
    print('Test loss:', score[0])
    print('Test accuracy:', score[1])
    return history, score

def test_no_of_layers(plot_type='layers'):
    kernel_size = [3,3]
    no_of_kernels = [32,64]
    no_of_layers = [1,1,2]
    dense = False
    legend = []
    legstring = ['no','1','1']
    for i in range(3):
        model = create_model(no_of_layers[i],no_of_kernels,kernel_size,dense,False)
        dense = True
        h,s = evaluate(model)
        plt.plot(h.history['val_acc'])
        legend.append('{} Convolution layer, {} dense layer(Test accuracy - {})'.format(i+1,legstring[i],s[1]))

    plt.title('Model accuracy v/s Number of hidden layers')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(legend, loc='best')
    plt.savefig(PLOTS_DIR + '/accuracy_cnn_{}.png'.format(plot_type))
    plt.show()
    plt.close()

def test_activation_function(plot_type='activation'):
    kernel_size = [3,3]
    no_of_kernels = [32,64]
    dense=True
    active='relu'
    legend = []
    legstring= ['ReLU','Sigmoid','TanH']
    for i in range(3):
```

```
        model = create_model(1,no_of_kernels,kernel_size,dense,False,active_func=active)
        if i==0:
            active='sigmoid'
        if i==1:
            active = 'tanh'
        h,s = evaluate(model)
        plt.plot(h.history['val_acc'])
        legend.append('{}(Test accuracy - {})'.format(legstring[i],s[1]))

    plt.title('Model accuracy v/s Activation function')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(legend, loc='best')
    plt.savefig(PLOTS_DIR + '/accuracy_cnn_{}.png'.format(plot_type))
    plt.show()
    plt.close()

def test_kernel_size(plot_type='kernel_size'):
    kernel_size=[3,7,15]
    no_of_kernels = [32,64]
    dense = True
    legstring = ['3x3','7x7','15x15']
    legend = []
    for i in range(3):
        model = create_model(1,no_of_kernels,[kernel_size[i]]*2,dense,False)
        h,s = evaluate(model)
        plt.plot(h.history['val_acc'])
        legend.append('{} kernel size (Test accuracy - {})'.format(legstring[i],s[1]))

    plt.title('Model accuracy v/s kernel size')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(legend, loc='best')
    plt.savefig(PLOTS_DIR + '/accuracy_cnn_{}.png'.format(plot_type))
    plt.show()
    plt.close()

def test_no_of_kernels(plot_type = 'no_of_kernels'):
    kernel_size=[3,3]
    no_of_kernels = [32,64]
    dense = True
    legstring = ['20','40','60']
```

```
    legend = []
    for i in range(3):
        model = create_model(1,no_of_kernels,kernel_size,dense,False)
        h,s = evaluate(model)
        plt.plot(h.history['val_acc'])
        legend.append('{} kernels (Test accuracy - {})'.format(legstring[i],s[1]))

    plt.title('Model accuracy v/s no of kernels')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(legend, loc='best')
    plt.savefig(PLOTS_DIR + '/accuracy_cnn_{}.png'.format(plot_type))
    plt.show()
    plt.close()

def test_overfitting(plot_type = 'overfitting'):
    kernel_size=[3,3]
    no_of_kernels = [32,64]
    dense = True
    legend = []
    epcoh = 40
    h1,s1 = evaluate(create_model(1,no_of_kernels,kernel_size,dense,False))
    h2,s2 = evaluate(create_model(1,no_of_kernels,kernel_size,dense,True))
    h3,s3 = evaluate(create_model(1,no_of_kernels,kernel_size,dense,False,l2_reg=0.01))
    plt.plot(h1.history['val_loss'])
    plt.plot(h2.history['val_loss'])
    plt.plot(h3.history['val_loss'])
    plt.title('Model loss in different overfitting techniques')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    legend1 = 'No mitigation strategy used(loss - {})'.format(s1[1])
    legend2 = 'Dropout mitigation strategy (loss- {})'.format(s2[1])
    legend3 = 'L2 regularization mitigation strategy (loss - {})'.format(s3[1])
    plt.legend([legend1,legend2,legend3], loc='best')
    plt.savefig(PLOTS_DIR + '/loss_cnn_{}.png'.format(plot_type))
    plt.show()
    epoch = 12
    plt.close()

if __name__ == '__main__':
#     test_no_of_layers()
#     test_activation_function()
```

```
#    test_kernel_size()
#    test_no_of_kernels()
    test_overfitting()
    pass

def test_optimization():
    kernel_size = [3,3]
    no_of_kernels = [32,64]
    no_of_layers = [1,1,2]
    dense = False
    legend = []
    legstring = ['adadelta','adam','adagrad']
    optimizer = ['adadelta','adam','adagrad']
    for i in range(3):
        model = create_model(1,no_of_kernels,kernel_size,dense,False)
        dense = True
        h,s = evaluate(model,optimizer_func=optimizer[i])
        plt.plot(h.history['val_acc'])
        legend.append(' Optimizer {} (Test accuracy - {})'.format(legstring[i],s[1]))

    plt.title('Model accuracy v/s optimizers')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(legend, loc='best')
#    plt.savefig(PLOTS_DIR + '/accuracy_cnn_{}.png'.format(plot_type))
    plt.show()
    plt.close()

test_optimization()
```