Saumya Kothari - Introduction to Neural Networks & Deep Learning Project [Part 4]

Part 4 [solved in the other ipynb notebook]

DOMAIN:

Autonomous Vehicles

BUSINESS CONTEXT:

- A Recognising multi-digit numbers in photographs captured at street level is an important component of modern-day map making. A classic example of a corpus of such street-level photographs is Google's Street View imagery composed of hundreds of millions of geo-located 360-degree panoramic images.
- The ability to automatically transcribe an address number from a geo-located patch of pixels and associate the transcribed number with a known street address helps pinpoint, with a high degree of accuracy, the location of the building it represents. More broadly, recognising numbers in photographs is a problem of interest to the optical character recognition community.
- While OCR on constrained domains like document processing is well studied, arbitrary multi-character text recognition in photographs is still highly challenging. This difficulty arises due to the wide variability in the visual appearance of text in the wild on account of a large range of fonts, colours, styles, orientations, and character arrangements.
- The recognition problem is further complicated by environmental factors such as lighting, shadows, specularity, and occlusions as well as by image acquisition factors such as resolution, motion, and focus blurs. In this project, we will use the dataset with images centred around a single digit (many of the images do contain some distractors at the sides). Although we are taking a sample of the data which is simpler, it is more complex than MNIST because of the distractors.

DATA DESCRIPTION:

The SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with the minimal requirement on data formatting but comes from a significantly harder, unsolved, real-world problem (recognising digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images. Where the labels for each of this image are the prominent number in that image i.e. 2,6,7 and 4 respectively. The dataset has been provided in the form of h5py files. You can read about this file format here: http://docs.h5py.org/en/stable/high/dataset.html

Acknowledgement: Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng Reading Digits in Natural Images with Unsupervised Feature Learning NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011. PDF http://ufldl.stanford.edu/housenumbers as the URL for this site when necessary

PROJECT OBJECTIVE:

We will build a digit classifier on the SVHN (Street View Housing Number) dataset. Steps and tasks: [Total Score: 30 points]

1. Import the data.
2. Data pre-processing and visualisation.
3. Design, train, tune and test a neural network image classifier. Hint: Use best approach to refine and tune the data or the model. Be highly experimental here to get the best accuracy out of the model.
4. Plot the training loss, validation loss vs number of epochs and training accuracy, validation accuracy vs number of epochs plot and write your observations on the same.

```
# Mounting Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

```
    Mounted at /content/drive
```

```
# Setting the current working directory
import os;
os.chdir('/content/drive/MyDrive/NeuralNetworks')
```

```
# Importing neccessary packages
import pandas as pd, numpy as np, matplotlib.pyplot as plt, seaborn as sns, h5py
import matplotlib.style as style; style.use('fivethirtyeight')
%matplotlib inline

# Metrics and preprocessing
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix, preci
from sklearn.model_selection import train_test_split
from sklearn import preprocessing

# TF and Keras
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.layers import Activation, Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras import optimizers

# Checking if GPU is found
import tensorflow as tf
device_name = tf.test.gpu_device_name()

#tf.reset_default_graph()
#tf.set_random_seed(42)
```

```
!ls '/content/drive/MyDrive/NeuralNetworks'
```

```
'Part - 4 - Autonomous_Vehicles_SVHN_single_grey1.h5'
```

Load train, validatin and test datasets from h5 file

```
# Read the h5 file
h5_Vehicle = h5py.File('Part - 4 - Autonomous_Vehicles_SVHN_single_grey1.h5', 'r')

# Load the training, validation and test sets
X_train = h5_Vehicle['X_train'][:]
y_train = h5_Vehicle['y_train'][:]
X_val = h5_Vehicle['X_val'][:]
y_val = h5_Vehicle['y_val'][:]
X_test = h5_Vehicle['X_test'][:]
y_test = h5_Vehicle['y_test'][:]

# Close this file

h5_Vehicle.close()

print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_val.shape, y_val.shape)
print('Test set', X_test.shape, y_test.shape)

print('\n')
print('Unique labels in y_train:', np.unique(y_train))
print('Unique labels in y_val:', np.unique(y_val))
print('Unique labels in y_test:', np.unique(y_test))
```

```
Training set (42000, 32, 32) (42000,)
Validation set (60000, 32, 32) (60000,)
Test set (18000, 32, 32) (18000,)


Unique labels in y_train: [0 1 2 3 4 5 6 7 8 9]
Unique labels in y_val: [0 1 2 3 4 5 6 7 8 9]
Unique labels in y_test: [0 1 2 3 4 5 6 7 8 9]
```

Observation: Length of training sets: 42k, validation sets: 60k, test sets: 18k

- Length of training sets: 42000, validation sets: 60000, test sets: 18000
- Size of the images: 32*32
- Number of class: 10

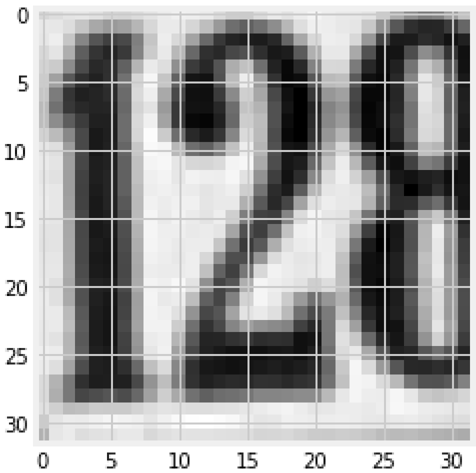
```
# Visualizing first 10 images in the dataset and their labels
plt.figure(figsize = (15, 5))
for i in range(10):
    plt.subplot(1, 10, i+1)
    plt.imshow(X_train[i].reshape((32, 32)),cmap = plt.cm.binary)
    plt.axis('off')
```

```
plt.subplots_adjust(wspace = -0.1, hspace = -0.1)
plt.show()

print('Label for each of the above image: %s' % (y_train[0 : 10]))
```

```
Label for each of the above image: [2 6 7 4 4 0 3 0 7 3]
```

```
print('Checking first image and label in training set: '); print('---'*20)
plt.imshow(X_train[0], cmap = plt.cm.binary)
plt.show()
print('Label:', y_train[0])
```

```
Checking first image and label in training set: 
------------------------------------------------------------
```



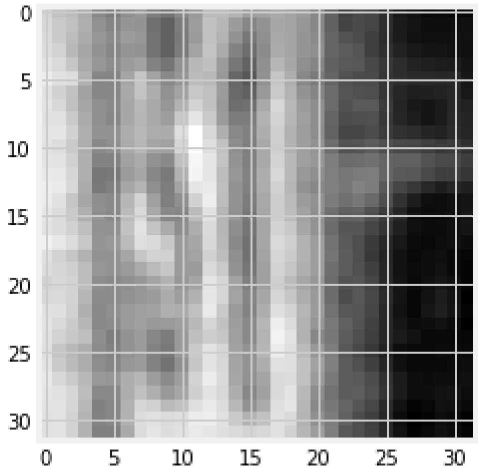

```
Label: 2
```

```
print('Checking first image and label in validation set:'); print('---'*20)
plt.imshow(X_val[0], cmap = plt.cm.binary)
plt.show()
print('Label:', y_val[0])
```

```
Checking first image and label in validation set:
------------------------------------------------------------
```

```
print('Checking first image and label in test set:'); print('---'*20)
plt.imshow(X_test[0], cmap = plt.cm.binary)
plt.show()
print('Label:', y_test[0])
```

```
Checking first image and label in test set:
------------------------------------------------------------
```




```
Label: 1
```

Flatten and normalize the images for Keras

```
# Reshaping the data to make it into two dimensions
print('Reshaping X data: From (n, 32, 32) to (n, 1024)'); print('----'*20)
X_train = X_train.reshape((X_train.shape[0], -1))
X_val = X_val.reshape((X_val.shape[0], -1))
X_test = X_test.reshape((X_test.shape[0], -1))

#Changing the datatype
print('Making sure that the values are float so that we can get decimal points after divis
X_train = X_train.astype('float32')
X_val = X_val.astype('float32')
X_test = X_test.astype('float32')

print('Normalizing the RGB codes by dividing it to the max RGB value'); print('----'*20)
X_train /= 255
X_val /= 255
X_test /= 255

print('Converting y data into categorical (one-hot encoding)'); print('----'*20)
y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)
```

```
Reshaping X data: From (n, 32, 32) to (n, 1024)
--------------------------------------------------------------------------------
Making sure that the values are float so that we can get decimal points after divisi
--------------------------------------------------------------------------------
Normalizing the RGB codes by dividing it to the max RGB value
```

```
--------------------------------------------------------------------------------
Converting y data into categorical (one-hot encoding)
--------------------------------------------------------------------------------
```

```
#Printing SHAPE
print('X_train shape:', X_train.shape)
print('X_val shape:', X_val.shape)
print('X_test shape:', X_test.shape)

print('\n')

print('y_train shape:', y_train.shape)
print('y_val shape:', y_val.shape)
print('y_test shape:', y_test.shape)

print('\n')

print('Number of images in X_train', X_train.shape[0])
print('Number of images in X_val', X_val.shape[0])
print('Number of images in X_test', X_test.shape[0])
```

```
X_train shape: (42000, 1024)
X_val shape: (60000, 1024)
X_test shape: (18000, 1024)


y_train shape: (42000, 10, 2, 2)
y_val shape: (60000, 10, 2, 2)
y_test shape: (18000, 10, 2, 2)


Number of images in X_train 42000
Number of images in X_val 60000
Number of images in X_test 18000
```

Modelling - Baby sitting the learning process

Fully connected linear layer

```
class Linear():
    def __init__(self, in_size, out_size):
        self.W = np.random.randn(in_size, out_size) * 0.01
        self.b = np.zeros((1, out_size))
        self.params = [self.W, self.b]
        self.gradW = None
        self.gradB = None
        self.gradInput = None

    def forward(self, X):
        self.X = X
        self.output = np.dot(X, self.W) + self.b
        return self.output
```

```
    def backward(self, nextgrad):
        self.gradW = np.dot(self.X.T, nextgrad)
        self.gradB = np.sum(nextgrad, axis=0)
        self.gradInput = np.dot(nextgrad, self.W.T)
        return self.gradInput, [self.gradW, self.gradB]
```

ReLU

```
class ReLU():
    def __init__(self):
        self.params = []
        self.gradInput = None

    def forward(self, X):
        self.output = np.maximum(X, 0)
        return self.output

    def backward(self, nextgrad):
        self.gradInput = nextgrad.copy()
        self.gradInput[self.output <=0] = 0
        return self.gradInput, []
```

Softmax function

```
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)
```

Cross entropy loss

```
class CrossEntropy:
    def forward(self, X, y):
        self.m = y.shape[0]
        self.p = softmax(X)
        cross_entropy = -np.log(self.p[range(self.m), y]+1e-16)
        loss = np.sum(cross_entropy) / self.m
        return loss

    def backward(self, X, y):
        y_idx = y.argmax()
        grad = softmax(X)
        grad[range(self.m), y] -= 1
        grad /= self.m
        return grad
```

NN class that enables the forward prop and backward propagation of the entire network

```
class NN():
```

```
class NN():
    def __init__(self, lossfunc = CrossEntropy(), mode = 'train'):
        self.params = []
        self.layers = []
        self.loss_func = lossfunc
        self.grads = []
        self.mode = mode

    def add_layer(self, layer):
        self.layers.append(layer)
        self.params.append(layer.params)

    def forward(self, X):
        for layer in self.layers:
            X = layer.forward(X)
        return X

    def backward(self, nextgrad):
        self.clear_grad_param()
        for layer in reversed(self.layers):
            nextgrad, grad = layer.backward(nextgrad)
            self.grads.append(grad)
        return self.grads

    def train_step(self, X, y):
        out = self.forward(X)
        loss = self.loss_func.forward(out,y)
        nextgrad = self.loss_func.backward(out,y)
        grads = self.backward(nextgrad)
        return loss, grads

    def predict(self, X):
        X = self.forward(X)
        p = softmax(X)
        return np.argmax(p, axis=1)

    def predict_scores(self, X):
        X = self.forward(X)
        p = softmax(X)
        return p

    def clear_grad_param(self):
        self.grads = []
```

Update SGD function with momentum

```
def update(velocity, params, grads, learning_rate=0.01, mu=0.9):
    for v, p, g, in zip(velocity, params, reversed(grads)):
        for i in range(len(g)):
            v[i] = (mu * v[i]) - (learning_rate * g[i])
            p[i] += v[i]
```

Getting minibatches

```
def minibatch(X, y, minibatch_size):
    n = X.shape[0]
    minibatches = []
    permutation = np.random.permutation(X.shape[0])
    X = X[permutation]
    y = y[permutation]

    for i in range(0, n , minibatch_size):
        X_batch = X[i:i + minibatch_size, :]
        y_batch = y[i:i + minibatch_size, ]
        minibatches.append((X_batch, y_batch))

    return minibatches
```

The Training:

```
def train(net, X_train, y_train, minibatch_size, epoch, learning_rate, mu = 0.9, X_val = N
    validationSet_loss_epochs = []
    minibatches = minibatch(X_train, y_train, minibatch_size)
    minibatches_val = minibatch(X_val, y_val, minibatch_size)

    for i in range(epoch):
        loss_batch = []
        val_loss_batch = []
        velocity = []
        for param_layer in net.params:
            p = [np.zeros_like(param) for param in list(param_layer)]
            velocity.append(p)

        # iterate over mini batches
        for X_mini, y_mini in minibatches:
            loss, grads = net.train_step(X_mini, y_mini)
            loss_batch.append(loss)
            update(velocity, net.params, grads, learning_rate=learning_rate, mu=mu)

        for X_mini_val, y_mini_val in minibatches_val:
            val_loss, _ = net.train_step(X_mini, y_mini)
            val_loss_batch.append(val_loss)

        # accuracy of model at end of epoch after all mini batch updates
        m_train = X_train.shape[0]
        m_val = X_val.shape[0]
        y_train_pred = []
        y_val_pred = []
        y_train1 = []
        y_vall = []
        for j in range(0, m_train, minibatch_size):
            X_tr = X_train[j:j + minibatch_size, : ]
            y_tr = y_train[j:j + minibatch_size,]
            y_train1 = np.append(y_train1, y_tr)
            y_train_pred = np.append(y_train_pred, net.predict(X_tr))
```

```
        for j in range(0, m_val, minibatch_size):
            X_va = X_val[j:j + minibatch_size, : ]
            y_va = y_val[j:j + minibatch_size,]
            y_vall = np.append(y_vall, y_va)
            y_val_pred = np.append(y_val_pred, net.predict(X_va))

        train_acc = check_accuracy(y_train1, y_train_pred)
        val_acc = check_accuracy(y_vall, y_val_pred)

        ## weights
        w = np.array(net.params[0][0])

        ## adding regularization to cost
        mean_train_loss = (sum(loss_batch) / float(len(loss_batch)))
        mean_val_loss = sum(val_loss_batch) / float(len(val_loss_batch))

        validationSet_loss_epochs.append(mean_val_loss)
        if verb:
            if i%50==0:
                print("Epoch {3}/{4}: Loss = {0} | Training Accuracy = {1}".format(mean_tr
    return net, val_acc
```

Checking the accuracy of the model

```
def check_accuracy(y_true, y_pred):
    return np.mean(y_pred == y_true)
```

Invoking created functions

```
# Invoking the model
## input size
input_dim = X_train.shape[1]

def train_and_test_loop(iterations, lr, Lambda, verb = True):
    ## hyperparameters
    iterations = iterations
    learning_rate = lr
    hidden_nodes1 = 10
    output_nodes = 10

    ## define neural net
    nn = NN()
    nn.add_layer(Linear(input_dim, hidden_nodes1))

    nn, val_acc = train(nn, X_train, y_train_o, minibatch_size = 200, epoch = iterations,
                        X_val = X_test, y_val = y_test_o, Lambda = Lambda, verb = verb)
    return val_acc


# Disable the regaularization after checking loss
lr = 0.00001
Lambda = 0
```

```
train_and_test_loop(1, lr, Lambda)
```

```
Epoch 0/1: Loss = 2.302585283088422 | Training Accuracy = 0.10192857142857142
0.0955
```

Increase Lambda(Regularization) and check what it does to our loss function

```
lr = 0.00001
Lambda = 1e4
train_and_test_loop(1, lr, Lambda)
```

```
Epoch 0/1: Loss = 2.3025852165243452 | Training Accuracy = 0.10192857142857142
0.0955
```

```
# Overfitting to a small subset of our dataset (in this case 20 images)

X_train_subset = X_train[0:20]
y_train_subset = y_train[0:20]

X_train = X_train_subset
y_train = y_train_subset

X_train.shape, y_train.shape
```

```
((20, 1024), (20, 10, 2, 2))
```

Making sure that we overfit very small portion of the training data

So, set a small learning rate and turn regularization off In the code below:

- Take the first 20 examples
- turn off regularization(reg=0.0)
- use simple vanilla 'sgd'

```
%time
lr = 0.001
Lambda = 0
train_and_test_loop(5000, lr, Lambda)
```

```
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 5.96 µs
Epoch 0/5000: Loss = 2.302585560613877 | Training Accuracy = 0.25
Epoch 50/5000: Loss = 2.2993515989242357 | Training Accuracy = 0.25
Epoch 100/5000: Loss = 2.2961498719261715 | Training Accuracy = 0.25
Epoch 150/5000: Loss = 2.2929801119822977 | Training Accuracy = 0.25
Epoch 200/5000: Loss = 2.2898420522393947 | Training Accuracy = 0.25
Epoch 250/5000: Loss = 2.2867354266485527 | Training Accuracy = 0.25
Epoch 300/5000: Loss = 2.283659969985244 | Training Accuracy = 0.25
Epoch 350/5000: Loss = 2.280615417869293 | Training Accuracy = 0.25
Epoch 400/5000: Loss = 2.277601506784756 | Training Accuracy = 0.25
Epoch 450/5000: Loss = 2.2746179740996704 | Training Accuracy = 0.25
Epoch 500/5000: Loss = 2.271664558085676 | Training Accuracy = 0.25
Epoch 550/5000: Loss = 2.2687409979374964 | Training Accuracy = 0.25
```

```
Epoch 600/5000: Loss = 2.2658470337922565 | Training Accuracy = 0.25
Epoch 650/5000: Loss = 2.2629824067486353 | Training Accuracy = 0.25
Epoch 700/5000: Loss = 2.2601468588858338 | Training Accuracy = 0.25
Epoch 750/5000: Loss = 2.257340133282354 | Training Accuracy = 0.25
Epoch 800/5000: Loss = 2.2545619740345675 | Training Accuracy = 0.25
Epoch 850/5000: Loss = 2.2518121262750768 | Training Accuracy = 0.25
Epoch 900/5000: Loss = 2.249090336190844 | Training Accuracy = 0.25
Epoch 950/5000: Loss = 2.2463963510410876 | Training Accuracy = 0.25
Epoch 1000/5000: Loss = 2.2437299191749314 | Training Accuracy = 0.25
Epoch 1050/5000: Loss = 2.241090790048798 | Training Accuracy = 0.25
Epoch 1100/5000: Loss = 2.238478714243543 | Training Accuracy = 0.25
Epoch 1150/5000: Loss = 2.2358934434813023 | Training Accuracy = 0.25
Epoch 1200/5000: Loss = 2.233334730642075 | Training Accuracy = 0.25
Epoch 1250/5000: Loss = 2.230802329780003 | Training Accuracy = 0.25
Epoch 1300/5000: Loss = 2.228295996139358 | Training Accuracy = 0.25
Epoch 1350/5000: Loss = 2.2258154861702257 | Training Accuracy = 0.25
Epoch 1400/5000: Loss = 2.223360557543871 | Training Accuracy = 0.25
Epoch 1450/5000: Loss = 2.2209309691677896 | Training Accuracy = 0.25
Epoch 1500/5000: Loss = 2.2185264812004357 | Training Accuracy = 0.25
Epoch 1550/5000: Loss = 2.2161468550656096 | Training Accuracy = 0.25
Epoch 1600/5000: Loss = 2.21379185346652 | Training Accuracy = 0.25
Epoch 1650/5000: Loss = 2.211461240399497 | Training Accuracy = 0.25
Epoch 1700/5000: Loss = 2.2091547811673578 | Training Accuracy = 0.25
Epoch 1750/5000: Loss = 2.2068722423924245 | Training Accuracy = 0.25
Epoch 1800/5000: Loss = 2.2046133920291804 | Training Accuracy = 0.25
Epoch 1850/5000: Loss = 2.2023779993765755 | Training Accuracy = 0.25
Epoch 1900/5000: Loss = 2.200165835089956 | Training Accuracy = 0.25
Epoch 1950/5000: Loss = 2.197976671192638 | Training Accuracy = 0.25
Epoch 2000/5000: Loss = 2.195810281087105 | Training Accuracy = 0.25
Epoch 2050/5000: Loss = 2.193666439565839 | Training Accuracy = 0.25
Epoch 2100/5000: Loss = 2.191544922821767 | Training Accuracy = 0.25
Epoch 2150/5000: Loss = 2.189445508458342 | Training Accuracy = 0.25
Epoch 2200/5000: Loss = 2.187367975499241 | Training Accuracy = 0.25
Epoch 2250/5000: Loss = 2.1853121043976813 | Training Accuracy = 0.25
Epoch 2300/5000: Loss = 2.183277677045359 | Training Accuracy = 0.25
Epoch 2350/5000: Loss = 2.1812644767810068 | Training Accuracy = 0.25
Epoch 2400/5000: Loss = 2.179272288398569 | Training Accuracy = 0.25
Epoch 2450/5000: Loss = 2.1773008981549915 | Training Accuracy = 0.25
Epoch 2500/5000: Loss = 2.1753500937776358 | Training Accuracy = 0.25
Epoch 2550/5000: Loss = 2.17341966447131 | Training Accuracy = 0.25
Epoch 2600/5000: Loss = 2.17150940092491 | Training Accuracy = 0.25
Epoch 2650/5000: Loss = 2.1696190953176986 | Training Accuracy = 0.25
Epoch 2700/5000: Loss = 2.167748541325187 | Training Accuracy = 0.25
Epoch 2750/5000: Loss = 2.1658975341246545 | Training Accuracy = 0.25
Epoch 2800/5000: Loss = 2.164065870400279 | Training Accuracy = 0.25
```

Loading the original dataset again

```
# Read the h5 file
h5_Vehicle = h5py.File('Part - 4 - Autonomous_Vehicles_SVHN_single_grey1.h5', 'r')

# Load the training, validation and test sets
X_train = h5_Vehicle['X_train'][:]
y_train_o = h5_Vehicle['y_train'][:]
X_val = h5_Vehicle['X_val'][:]
y_val_o = h5_Vehicle['y_val'][:]
X_test = h5_Vehicle['X_test'][:]
y_test_o = h5_Vehicle['y_test'][:]
```

```
print('Reshaping X data: (n, 32, 32) => (n, 1024)'); print('--'*40)
X_train = X_train.reshape((X_train.shape[0], -1))
X_val = X_val.reshape((X_val.shape[0], -1))
X_test = X_test.reshape((X_test.shape[0], -1))

print('Making sure that the values are float so that we can get decimal points after divis
X_train = X_train.astype('float32')
X_val = X_val.astype('float32')
X_test = X_test.astype('float32')

print('Normalizing the RGB codes by dividing it to the max RGB value'); print('--'*40)
X_train /= 255
X_val /= 255
X_test /= 255

print('Converting y data into categorical (one-hot encoding)'); print('--'*40)
y_train = to_categorical(y_train_o)
y_val = to_categorical(y_val_o)
y_test = to_categorical(y_test_o)
```

```
Reshaping X data: (n, 32, 32) => (n, 1024)
--------------------------------------------------------------------------------
Making sure that the values are float so that we can get decimal points after divisi
--------------------------------------------------------------------------------
Normalizing the RGB codes by dividing it to the max RGB value
--------------------------------------------------------------------------------
Converting y data into categorical (one-hot encoding)
--------------------------------------------------------------------------------
```

Start with small regularization and find learning rate that makes the loss go down.

```
lr = 1e-7
Lambda = 1e-7
train_and_test_loop(500, lr, Lambda)
```

```
Epoch 0/500: Loss = 2.3100252019112637 | Training Accuracy = 0.10245238095238095
Epoch 50/500: Loss = 2.3075806310263567 | Training Accuracy = 0.10207142857142858
Epoch 100/500: Loss = 2.306031700087846 | Training Accuracy = 0.1015
Epoch 150/500: Loss = 2.305042411579897 | Training Accuracy = 0.10019047619047619
Epoch 200/500: Loss = 2.3044056325756435 | Training Accuracy = 0.09988095238095238
Epoch 250/500: Loss = 2.3039921173807456 | Training Accuracy = 0.09854761904761905
Epoch 300/500: Loss = 2.3037205595440673 | Training Accuracy = 0.09671428571428571
Epoch 350/500: Loss = 2.303539522404581 | Training Accuracy = 0.09511904761904762
Epoch 400/500: Loss = 2.3034163366739833 | Training Accuracy = 0.09416666666666666
Epoch 450/500: Loss = 2.3033301932319805 | Training Accuracy = 0.09264285714285714
0.08572222222222223
```

```
# Changing learning rate to 1e-3

lr = 0.001
Lambda = 1e-7
train_and_test_loop(500, lr, Lambda)
```

```
Epoch 0/500: Loss = 2.3054600951507305 | Training Accuracy = 0.11395238095238096
Epoch 50/500: Loss = 2.259522161398282 | Training Accuracy = 0.19652380952380952
Epoch 100/500: Loss = 2.2509994247963587 | Training Accuracy = 0.20873809523809525
Epoch 150/500: Loss = 2.2466623930996477 | Training Accuracy = 0.2149047619047619
Epoch 200/500: Loss = 2.2437567193893795 | Training Accuracy = 0.21921428571428572
Epoch 250/500: Loss = 2.2415719552403033 | Training Accuracy = 0.22171428571428572
Epoch 300/500: Loss = 2.2398229369841203 | Training Accuracy = 0.2241904761904762
Epoch 350/500: Loss = 2.2383653065434563 | Training Accuracy = 0.22564285714285715
Epoch 400/500: Loss = 2.237115482633667 | Training Accuracy = 0.22628571428571428
Epoch 450/500: Loss = 2.236020749372444 | Training Accuracy = 0.22785714285714287
0.21066666666666667
```

Optimizing hyperparameter and running a deeper search

```
import math
for i in range(1, 10):
    lr = math.pow(10, np.random.uniform(-3.0, -2.0))
    Lambda = math.pow(10, np.random.uniform(-5, 2))
    best_acc = train_and_test_loop(100, lr, Lambda, False)
    print("Try {0}/{1}: Best_val_acc: {2}, lr: {3}, Lambda: {4}\n".format(i, 10, best_acc,
```

```
Try 1/10: Best_val_acc: 0.183, lr: 0.006194881356749269, Lambda: 3.0674849905942456e

Try 2/10: Best_val_acc: 0.204, lr: 0.005446364406380019, Lambda: 51.87240826024816

Try 3/10: Best_val_acc: 0.2088888888888889, lr: 0.0016522382707245898, Lambda: 0.193

Try 4/10: Best_val_acc: 0.19822222222222222, lr: 0.0017057627561567893, Lambda: 2.51

Try 5/10: Best_val_acc: 0.19066666666666668, lr: 0.005152224038963205, Lambda: 1.214

Try 6/10: Best_val_acc: 0.20827777777777778, lr: 0.00921935228718037, Lambda: 0.0004

Try 7/10: Best_val_acc: 0.18577777777777776, lr: 0.00878261288169845, Lambda: 0.0035

Try 8/10: Best_val_acc: 0.19533333333333333, lr: 0.006090543136815944, Lambda: 13.60

Try 9/10: Best_val_acc: 0.18994444444444444, lr: 0.007509709326301158, Lambda: 2.335
```

Observation:

Best accuracy achieved using this method after hyperparameter optimization: ~21%

Modelling - Neural Network

NN model, Sigmoid activation functions, SGD optimizer

```
print('NN model with sigmoid activations'); print('----'*20)
# Initialize the neural network classifier
model = Sequential()
```

```
# Input Layer - adding input layer and activation functions sigmoid
model.add(Dense(128, input_shape = (1024, )))
# Adding activation function
model.add(Activation('sigmoid'))

#Hidden Layer 1 - adding first hidden layer
model.add(Dense(64))
# Adding activation function
model.add(Activation('sigmoid'))

# Output Layer - adding output layer which is of 10 nodes (digits)
model.add(Dense(10))
# Adding activation function - softmax for multiclass classification
model.add(Activation('softmax'))
```

```
NN model with sigmoid activations
--------------------------------------------------------------------------------
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               131200
_________________________________________________________________
activation (Activation)      (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
_________________________________________________________________
activation_1 (Activation)    (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 140,106
Trainable params: 140,106
Non-trainable params: 0
_________________________________________________________________
```

```
# compiling the neural network classifier, sgd optimizer
sgd = optimizers.SGD(lr = 0.01)
model.compile(optimizer = sgd, loss = 'categorical_crossentropy', metrics = ['accuracy'])

# Fitting the neural network for training
history = model.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
Epoch 1/100
210/210 [==============================] - 3s 12ms/step - loss: 2.3747 - accuracy
Epoch 2/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3030 - accuracy
Epoch 3/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3030 - accuracy
Epoch 4/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3029 - accuracy
Epoch 5/100
```

```
210/210 [==============================] - 2s 10ms/step - loss: 2.3028 - accuracy
Epoch 6/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3028 - accuracy
Epoch 7/100
210/210 [==============================] - 2s 9ms/step - loss: 2.3027 - accuracy:
Epoch 8/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3025 - accuracy
Epoch 9/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3024 - accuracy
Epoch 10/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3023 - accuracy
Epoch 11/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3026 - accuracy
Epoch 12/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3021 - accuracy
Epoch 13/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3023 - accuracy
Epoch 14/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3022 - accuracy
Epoch 15/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3023 - accuracy
Epoch 16/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3024 - accuracy
Epoch 17/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3023 - accuracy
Epoch 18/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3020 - accuracy
Epoch 19/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3020 - accuracy
Epoch 20/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3021 - accuracy
Epoch 21/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3020 - accuracy
Epoch 22/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3018 - accuracy
Epoch 23/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3021 - accuracy
Epoch 24/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3019 - accuracy
Epoch 25/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3018 - accuracy
Epoch 26/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3020 - accuracy
Epoch 27/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3017 - accuracy
Epoch 28/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3017 - accuracy
Epoch 29/100
210/210 [==============================] - 2s 10ms/step - loss: 2.3018 - accuracy
```

```
print('Evaluate NN model with sigmoid activations'); print('----'*20)
results1 = model.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results1[1]*100, 2), '%'))
```

```
Evaluate NN model with sigmoid activations
--------------------------------------------------------------------------------
1875/1875 [==============================] - 4s 2ms/step - loss: 2.2991 - accuracy:
Validation accuracy: 14.8
```

NN model, Sigmoid activation functions, SGD optimizer: Changing Learning Rate

```
print('NN model with sigmoid activations - changing learning rate'); print('----'*20)
# compiling the neural network classifier, sgd optimizer
sgd = optimizers.SGD(lr = 0.001)
model.compile(optimizer = sgd, loss = 'categorical_crossentropy', metrics = ['accuracy'])

# Fitting the neural network for training
history = model.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
NN model with sigmoid activations - changing learning rate
--------------------------------------------------------------------------------
Epoch 1/100
210/210 [==============================] - 3s 11ms/step - loss: 2.2990 - accuracy
Epoch 2/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2988 - accuracy
Epoch 3/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2990 - accuracy
Epoch 4/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2989 - accuracy
Epoch 5/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2986 - accuracy
Epoch 6/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2990 - accuracy
Epoch 7/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2990 - accuracy
Epoch 8/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2990 - accuracy
Epoch 9/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2988 - accuracy
Epoch 10/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2988 - accuracy
Epoch 11/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2989 - accuracy
Epoch 12/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2988 - accuracy
Epoch 13/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2990 - accuracy
Epoch 14/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2988 - accuracy
Epoch 15/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2990 - accuracy
Epoch 16/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2987 - accuracy
Epoch 17/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2985 - accuracy
Epoch 18/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2990 - accuracy
Epoch 19/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2989 - accuracy
Epoch 20/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2987 - accuracy
Epoch 21/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2987 - accuracy
Epoch 22/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2990 - accuracy
Epoch 23/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2988 - accuracy
```

```
Epoch 24/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2988 - accuracy
Epoch 25/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2987 - accuracy
Epoch 26/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2987 - accuracy
Epoch 27/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2989 - accuracy
Epoch 28/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2989 - accuracy
```

```
print('Evaluate NN model with sigmoid activations - changing learning rate'); print('--'*4
results1 = model.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results1[1]*100, 2), '%'))
```

```
Evaluate NN model with sigmoid activations - changing learning rate
--------------------------------------------------------------------------------
1875/1875 [==============================] - 4s 2ms/step - loss: 2.2987 - accuracy:
Validation accuracy: 14.45
```

Observation:

- Validation score is very low, changing learning rate further reduces it.
- Optimizing the network in order to better learn the patterns in the dataset.
- Best model out of the above is the one with lower learning rate using SGD optimizer and sigmoid activations.

Let's use ReLU activations and see if the score improves.

```
%time
print('NN model with relu activations and sgd optimizers'); print('--'*40)
# Initialize the neural network classifier
model2 = Sequential()

# Input Layer - adding input layer and activation functions relu
model2.add(Dense(128, input_shape = (1024, )))
# Adding activation function
model2.add(Activation('relu'))

#Hidden Layer 1 - adding first hidden layer
model2.add(Dense(64))
# Adding activation function
model2.add(Activation('relu'))

# Output Layer - adding output layer which is of 10 nodes (digits)
model2.add(Dense(10))
# Adding activation function - softmax for multiclass classification
model2.add(Activation('softmax'))
```

```
CPU times: user 2 µs, sys: 2 µs, total: 4 µs
Wall time: 6.91 µs
```

```
NN model with relu activations and sgd optimizers
--------------------------------------------------------------------------------
```

```
model2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 128)               131200
_________________________________________________________________
activation_3 (Activation)    (None, 128)               0
_________________________________________________________________
dense_4 (Dense)              (None, 64)                8256
_________________________________________________________________
activation_4 (Activation)    (None, 64)                0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                650
_________________________________________________________________
activation_5 (Activation)    (None, 10)                0
=================================================================
Total params: 140,106
Trainable params: 140,106
Non-trainable params: 0
_________________________________________________________________
```

```
# compiling the neural network classifier, sgd optimizer
sgd = optimizers.SGD(lr = 0.01)
model2.compile(optimizer = sgd, loss = 'categorical_crossentropy', metrics = ['accuracy'])

# Fitting the neural network for training
history = model2.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
Epoch 1/100
210/210 [==============================] - 3s 11ms/step - loss: 2.3087 - accuracy
Epoch 2/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2904 - accuracy
Epoch 3/100
210/210 [==============================] - 2s 11ms/step - loss: 2.2793 - accuracy
Epoch 4/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2673 - accuracy
Epoch 5/100
210/210 [==============================] - 2s 11ms/step - loss: 2.2507 - accuracy
Epoch 6/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2340 - accuracy
Epoch 7/100
210/210 [==============================] - 2s 10ms/step - loss: 2.2107 - accuracy
Epoch 8/100
210/210 [==============================] - 2s 10ms/step - loss: 2.1842 - accuracy
Epoch 9/100
210/210 [==============================] - 2s 10ms/step - loss: 2.1492 - accuracy
Epoch 10/100
210/210 [==============================] - 2s 10ms/step - loss: 2.1073 - accuracy
Epoch 11/100
210/210 [==============================] - 2s 10ms/step - loss: 2.0602 - accuracy
Epoch 12/100
210/210 [==============================] - 2s 10ms/step - loss: 2.0047 - accuracy
Epoch 13/100
210/210 [==============================] - 2s 10ms/step - loss: 1.9415 - accuracy
Epoch 14/100
```

```
210/210 [==============================] - 2s 10ms/step - loss: 1.8783 - accuracy
Epoch 15/100
210/210 [==============================] - 2s 10ms/step - loss: 1.8106 - accuracy
Epoch 16/100
210/210 [==============================] - 2s 10ms/step - loss: 1.7501 - accuracy
Epoch 17/100
210/210 [==============================] - 2s 10ms/step - loss: 1.6858 - accuracy
Epoch 18/100
210/210 [==============================] - 2s 10ms/step - loss: 1.6311 - accuracy
Epoch 19/100
210/210 [==============================] - 2s 10ms/step - loss: 1.5787 - accuracy
Epoch 20/100
210/210 [==============================] - 2s 10ms/step - loss: 1.5374 - accuracy
Epoch 21/100
210/210 [==============================] - 2s 11ms/step - loss: 1.4915 - accuracy
Epoch 22/100
210/210 [==============================] - 2s 10ms/step - loss: 1.4524 - accuracy
Epoch 23/100
210/210 [==============================] - 2s 10ms/step - loss: 1.4043 - accuracy
Epoch 24/100
210/210 [==============================] - 2s 10ms/step - loss: 1.3740 - accuracy
Epoch 25/100
210/210 [==============================] - 2s 10ms/step - loss: 1.3432 - accuracy
Epoch 26/100
210/210 [==============================] - 2s 10ms/step - loss: 1.3168 - accuracy
Epoch 27/100
210/210 [==============================] - 2s 10ms/step - loss: 1.2946 - accuracy
Epoch 28/100
210/210 [==============================] - 2s 10ms/step - loss: 1.2705 - accuracy
Epoch 29/100
210/210 [==============================] - 2s 10ms/step - loss: 1.2486 - accuracy
```

```
print('Evaluate NN model with relu activations'); print('--'*40)
results2 = model2.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results2[1]*100, 2), '%'))
```

```
Evaluate NN model with relu activations
--------------------------------------------------------------------------------
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7518 - accuracy:
Validation accuracy: 77.84
```

NN model, ReLU activations, SGD optimizer: Changing Learning Rate

```
%time
print('NN model with relu activations and sgd optimizers - changing learning rate'); print
# compiling the neural network classifier, sgd optimizer
sgd = optimizers.SGD(lr = 0.001)
model2.compile(optimizer = sgd, loss = 'categorical_crossentropy', metrics = ['accuracy']]

# Fitting the neural network for training
history = model2.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
CPU times: user 3 µs, sys: 1e+03 ns, total: 4 µs
Wall time: 10.3 µs
```

```
NN model with relu activations and sgd optimizers - changing learning rate
--------------------------------------------------------------------------------
Epoch 1/100
210/210 [==============================] - 3s 11ms/step - loss: 0.6992 - accuracy
Epoch 2/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6903 - accuracy
Epoch 3/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6970 - accuracy
Epoch 4/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6938 - accuracy
Epoch 5/100
210/210 [==============================] - 2s 10ms/step - loss: 0.7019 - accuracy
Epoch 6/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6912 - accuracy
Epoch 7/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6932 - accuracy
Epoch 8/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6854 - accuracy
Epoch 9/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6960 - accuracy
Epoch 10/100
210/210 [==============================] - 2s 9ms/step - loss: 0.6938 - accuracy:
Epoch 11/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6924 - accuracy
Epoch 12/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6855 - accuracy
Epoch 13/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6920 - accuracy
Epoch 14/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6951 - accuracy
Epoch 15/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6910 - accuracy
Epoch 16/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6861 - accuracy
Epoch 17/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6894 - accuracy
Epoch 18/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6870 - accuracy
Epoch 19/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6934 - accuracy
Epoch 20/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6919 - accuracy
Epoch 21/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6811 - accuracy
Epoch 22/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6776 - accuracy
Epoch 23/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6799 - accuracy
Epoch 24/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6920 - accuracy
Epoch 25/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6856 - accuracy
Epoch 26/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6825 - accuracy
Epoch 27/100
```

```
print('Evaluate NN model with relu activations'); print('--'*40)
results2 = model2.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results2[1]*100, 2), '%'))
```

```
Evaluate NN model with relu activations
--------------------------------------------------------------------------------
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6889 - accuracy:
Validation accuracy: 80.11
```

NN model, ReLU activations, Adam optimizer

```
%time
print('NN model with relu activations and adam optimizer'); print('--'*40)
# compiling the neural network classifier, adam optimizer
adam = optimizers.Adam(lr = 0.01)
model2.compile(optimizer = adam, loss = 'categorical_crossentropy', metrics = ['accuracy']

# Fitting the neural network for training
history = model2.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 7.87 µs
NN model with relu activations and adam optimizer
--------------------------------------------------------------------------------
Epoch 1/100
210/210 [==============================] - 3s 11ms/step - loss: 6.6549 - accuracy
Epoch 2/100
210/210 [==============================] - 2s 10ms/step - loss: 1.8963 - accuracy
Epoch 3/100
210/210 [==============================] - 2s 10ms/step - loss: 1.4858 - accuracy
Epoch 4/100
210/210 [==============================] - 2s 11ms/step - loss: 1.2725 - accuracy
Epoch 5/100
210/210 [==============================] - 2s 10ms/step - loss: 1.2166 - accuracy
Epoch 6/100
210/210 [==============================] - 2s 10ms/step - loss: 1.1465 - accuracy
Epoch 7/100
210/210 [==============================] - 2s 10ms/step - loss: 1.1023 - accuracy
Epoch 8/100
210/210 [==============================] - 2s 11ms/step - loss: 1.0603 - accuracy
Epoch 9/100
210/210 [==============================] - 2s 10ms/step - loss: 1.0153 - accuracy
Epoch 10/100
210/210 [==============================] - 2s 11ms/step - loss: 1.0142 - accuracy
Epoch 11/100
210/210 [==============================] - 2s 10ms/step - loss: 0.9965 - accuracy
Epoch 12/100
210/210 [==============================] - 2s 10ms/step - loss: 0.9758 - accuracy
Epoch 13/100
210/210 [==============================] - 2s 10ms/step - loss: 0.9388 - accuracy
Epoch 14/100
210/210 [==============================] - 2s 11ms/step - loss: 0.9701 - accuracy
Epoch 15/100
210/210 [==============================] - 2s 10ms/step - loss: 0.9430 - accuracy
Epoch 16/100
210/210 [==============================] - 2s 11ms/step - loss: 0.9426 - accuracy
Epoch 17/100
210/210 [==============================] - 2s 11ms/step - loss: 0.9335 - accuracy
Epoch 18/100
210/210 [==============================] - 2s 11ms/step - loss: 0.8868 - accuracy
```

```
Epoch 19/100
210/210 [==============================] - 2s 10ms/step - loss: 0.9145 - accuracy
Epoch 20/100
210/210 [==============================] - 2s 10ms/step - loss: 0.9092 - accuracy
Epoch 21/100
210/210 [==============================] - 2s 10ms/step - loss: 0.8933 - accuracy
Epoch 22/100
210/210 [==============================] - 2s 10ms/step - loss: 0.8871 - accuracy
Epoch 23/100
210/210 [==============================] - 2s 10ms/step - loss: 0.8836 - accuracy
Epoch 24/100
210/210 [==============================] - 2s 11ms/step - loss: 0.8631 - accuracy
Epoch 25/100
210/210 [==============================] - 2s 11ms/step - loss: 0.8486 - accuracy
Epoch 26/100
210/210 [==============================] - 2s 11ms/step - loss: 0.8484 - accuracy
Epoch 27/100
210/210 [==============================] - 2s 11ms/step - loss: 0.8694 - accuracy
```

```
print('Evaluate NN model with relu activations'); print('--'*40)
results2 = model2.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results2[1]*100, 2), '%'))
```

```
Evaluate NN model with relu activations
--------------------------------------------------------------------------------
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7597 - accuracy:
Validation accuracy: 76.88
```

NN model, ReLU activations, Adam optimizer: Changing Learning Rate

```
%time
print('NN model with relu activations and adam optimizer'); print('--'*40)
# compiling the neural network classifier, adam optimizer
adam = optimizers.Adam(lr = 0.001)
model2.compile(optimizer = adam, loss = 'categorical_crossentropy', metrics = ['accuracy']

# Fitting the neural network for training
history = model2.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
CPU times: user 3 µs, sys: 1 µs, total: 4 µs
Wall time: 10.5 µs
NN model with relu activations and adam optimizer
--------------------------------------------------------------------------------
Epoch 1/100
210/210 [==============================] - 3s 11ms/step - loss: 0.6387 - accuracy
Epoch 2/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6147 - accuracy
Epoch 3/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6124 - accuracy
Epoch 4/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6152 - accuracy
Epoch 5/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6041 - accuracy
Epoch 6/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6030 - accuracy
```

```
Epoch 7/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6053 - accuracy
Epoch 8/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5964 - accuracy
Epoch 9/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5915 - accuracy
Epoch 10/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6086 - accuracy
Epoch 11/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6018 - accuracy
Epoch 12/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6070 - accuracy
Epoch 13/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6082 - accuracy
Epoch 14/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6008 - accuracy
Epoch 15/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5911 - accuracy
Epoch 16/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5948 - accuracy
Epoch 17/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5969 - accuracy
Epoch 18/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5879 - accuracy
Epoch 19/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5965 - accuracy
Epoch 20/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5931 - accuracy
Epoch 21/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5969 - accuracy
Epoch 22/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6008 - accuracy
Epoch 23/100
210/210 [==============================] - 2s 10ms/step - loss: 0.6040 - accuracy
Epoch 24/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5946 - accuracy
Epoch 25/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5989 - accuracy
Epoch 26/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5921 - accuracy
Epoch 27/100
210/210 [==============================] - 2s 10ms/step - loss: 0.5770 - accuracy
```

```
print('Evaluate NN model with relu activations'); print('--'*40)
results2 = model2.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results2[1]*100, 2), '%'))
```

```
Evaluate NN model with relu activations
--------------------------------------------------------------------------------
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6320 - accuracy:
Validation accuracy: 81.42
```

Observation:

- Improves the accuracy score considerably

- Best accuracy achieved till now is using relu activations, SGD optimizer, changing learning rate to 0.001.

Let's try and change the number of activators and see if the score improves.

NN model, ReLU activations, Changing Number of Activators, SGD optimizers

```
print('NN model with relu activations and changing number of activators'); print('--'*40)
# Initialize the neural network classifier
model3 = Sequential()

# Input Layer - adding input layer and activation functions relu
model3.add(Dense(256, input_shape = (1024, )))
# Adding activation function
model3.add(Activation('relu'))

#Hidden Layer 1 - adding first hidden layer
model3.add(Dense(128))
# Adding activation function
model3.add(Activation('relu'))

#Hidden Layer 2 - Adding second hidden layer
model3.add(Dense(64))
# Adding activation function
model3.add(Activation('relu'))

# Output Layer - adding output layer which is of 10 nodes (digits)
model3.add(Dense(10))
# Adding activation function - softmax for multiclass classification
model3.add(Activation('softmax'))
```

```
NN model with relu activations and changing number of activators
--------------------------------------------------------------------------------
```

```
model3.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 256)	262400
activation_6 (Activation)	(None, 256)	0
dense_7 (Dense)	(None, 128)	32896
activation_7 (Activation)	(None, 128)	0
dense_8 (Dense)	(None, 64)	8256
activation_8 (Activation)	(None, 64)	0

```
dense_9 (Dense)              (None, 10)                650
_________________________________________________________________
activation_9 (Activation)    (None, 10)                0
=================================================================
Total params: 304,202
Trainable params: 304,202
Non-trainable params: 0
_________________________________________________________________
```

```
# compiling the neural network classifier, sgd optimizer
sgd = optimizers.SGD(lr = 0.01)
model3.compile(optimizer = sgd, loss = 'categorical_crossentropy', metrics = ['accuracy'])

# Fitting the neural network for training
history = model3.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
Epoch 1/100
210/210 [==============================] - 4s 18ms/step - loss: 2.3066 - accuracy
Epoch 2/100
210/210 [==============================] - 3s 16ms/step - loss: 2.2862 - accuracy
Epoch 3/100
210/210 [==============================] - 3s 16ms/step - loss: 2.2739 - accuracy
Epoch 4/100
210/210 [==============================] - 3s 16ms/step - loss: 2.2565 - accuracy
Epoch 5/100
210/210 [==============================] - 3s 16ms/step - loss: 2.2340 - accuracy
Epoch 6/100
210/210 [==============================] - 3s 16ms/step - loss: 2.2021 - accuracy
Epoch 7/100
210/210 [==============================] - 3s 16ms/step - loss: 2.1604 - accuracy
Epoch 8/100
210/210 [==============================] - 3s 16ms/step - loss: 2.1026 - accuracy
Epoch 9/100
210/210 [==============================] - 3s 16ms/step - loss: 2.0288 - accuracy
Epoch 10/100
210/210 [==============================] - 3s 16ms/step - loss: 1.9525 - accuracy
Epoch 11/100
210/210 [==============================] - 3s 16ms/step - loss: 1.8731 - accuracy
Epoch 12/100
210/210 [==============================] - 3s 16ms/step - loss: 1.7954 - accuracy
Epoch 13/100
210/210 [==============================] - 3s 16ms/step - loss: 1.7163 - accuracy
Epoch 14/100
210/210 [==============================] - 3s 16ms/step - loss: 1.6472 - accuracy
Epoch 15/100
210/210 [==============================] - 3s 16ms/step - loss: 1.5879 - accuracy
Epoch 16/100
210/210 [==============================] - 3s 16ms/step - loss: 1.5252 - accuracy
Epoch 17/100
210/210 [==============================] - 3s 16ms/step - loss: 1.4742 - accuracy
Epoch 18/100
210/210 [==============================] - 3s 16ms/step - loss: 1.4223 - accuracy
Epoch 19/100
210/210 [==============================] - 3s 16ms/step - loss: 1.4013 - accuracy
Epoch 20/100
210/210 [==============================] - 3s 16ms/step - loss: 1.3327 - accuracy
Epoch 21/100
210/210 [==============================] - 3s 16ms/step - loss: 1.3074 - accuracy
Epoch 22/100
210/210 [==============================] - 3s 17ms/step - loss: 1.2826 - accuracy
```

```
Epoch 23/100
210/210 [==============================] - 3s 16ms/step - loss: 1.2506 - accuracy
Epoch 24/100
210/210 [==============================] - 3s 16ms/step - loss: 1.2173 - accuracy
Epoch 25/100
210/210 [==============================] - 3s 16ms/step - loss: 1.2098 - accuracy
Epoch 26/100
210/210 [==============================] - 3s 16ms/step - loss: 1.1671 - accuracy
Epoch 27/100
210/210 [==============================] - 3s 16ms/step - loss: 1.1587 - accuracy
Epoch 28/100
210/210 [==============================] - 3s 16ms/step - loss: 1.1399 - accuracy
Epoch 29/100
210/210 [==============================] - 3s 16ms/step - loss: 1.1067 - accuracy
```

```
print('Evaluate NN model with relu activations and changing the number of activators'); pr
results3 = model3.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results3[1]*100, 2), '%'))
```

```
Evaluate NN model with relu activations and changing the number of activators
--------------------------------------------------------------------------------
1875/1875 [==============================] - 5s 3ms/step - loss: 0.6237 - accuracy:
Validation accuracy: 81.53
```

NN model, ReLU activations, Changing Number of Activators, Adam optimizers

```
# compiling the neural network classifier, adam optimizer
adam = optimizers.Adam(lr = 0.001)
model3.compile(optimizer = adam, loss = 'categorical_crossentropy', metrics = ['accuracy']

# Fitting the neural network for training
history = model3.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
Epoch 1/100
210/210 [==============================] - 4s 18ms/step - loss: 1.3323 - accuracy
Epoch 2/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7631 - accuracy
Epoch 3/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7729 - accuracy
Epoch 4/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7550 - accuracy
Epoch 5/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7219 - accuracy
Epoch 6/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7364 - accuracy
Epoch 7/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6935 - accuracy
Epoch 8/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6760 - accuracy
Epoch 9/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6680 - accuracy
Epoch 10/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6601 - accuracy
Epoch 11/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6557 - accuracy
```

```
Epoch 12/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6021 - accuracy
Epoch 13/100
210/210 [==============================] - 3s 16ms/step - loss: 0.6118 - accuracy
Epoch 14/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6110 - accuracy
Epoch 15/100
210/210 [==============================] - 3s 17ms/step - loss: 0.6119 - accuracy
Epoch 16/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5794 - accuracy
Epoch 17/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5689 - accuracy
Epoch 18/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5577 - accuracy
Epoch 19/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5607 - accuracy
Epoch 20/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5259 - accuracy
Epoch 21/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5302 - accuracy
Epoch 22/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5121 - accuracy
Epoch 23/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5137 - accuracy
Epoch 24/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5037 - accuracy
Epoch 25/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5014 - accuracy
Epoch 26/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5060 - accuracy
Epoch 27/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4812 - accuracy
Epoch 28/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4777 - accuracy
Epoch 29/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4827 - accuracy
```

```
print('Evaluate NN model with relu activations and changing the number of activators'); pr
results3 = model3.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results3[1]*100, 2), '%'))
```

```
Evaluate NN model with relu activations and changing the number of activators
--------------------------------------------------------------------------------
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3653 - accuracy:
Validation accuracy: 90.32
```

Observation:

- Adding ReLU activations and changing activators results in improvement of score.
- Best accuracy achieved till now is using relu activations, changing number of activators and Adam optimizers with a learning rate of 0.001

Let's try adding weight initilization

With Weight Initializers

Changing weight initialization scheme can significantly improve training of the model by preventing vanishing gradient problem up to some degree.

NN model, relu activations, SGD optimizers with weight initializers

```
print('NN model with weight initializers'); print('--'*40)
# Initialize the neural network classifier
model4 = Sequential()

# Input Layer - adding input layer and activation functions relu and weight initializer
model4.add(Dense(256, input_shape = (1024, ), kernel_initializer = 'he_normal'))
# Adding activation function
model4.add(Activation('relu'))

#Hidden Layer 1 - adding first hidden layer
model4.add(Dense(128, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding activation function
model4.add(Activation('relu'))

#Hidden Layer 2 - adding second hidden layer
model4.add(Dense(64, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding activation function
model4.add(Activation('relu'))

#Hidden Layer 3 - adding third hidden layer
model4.add(Dense(32, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding activation function
model4.add(Activation('relu'))

# Output Layer - adding output layer which is of 10 nodes (digits)
model4.add(Dense(10, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding activation function
model4.add(Activation('softmax'))
```

```
NN model with weight initializers
--------------------------------------------------------------------------------
```

```
model4.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_10 (Dense)	(None, 256)	262400
activation_10 (Activation)	(None, 256)	0
dense_11 (Dense)	(None, 128)	32896
activation_11 (Activation)	(None, 128)	0

```
dense_12 (Dense)            (None, 64)              8256
_________________________________________________________________
activation_12 (Activation)  (None, 64)              0
_________________________________________________________________
dense_13 (Dense)            (None, 32)              2080
_________________________________________________________________
activation_13 (Activation)  (None, 32)              0
_________________________________________________________________
dense_14 (Dense)            (None, 10)              330
_________________________________________________________________
activation_14 (Activation)  (None, 10)              0
=================================================================
Total params: 305,962
Trainable params: 305,962
Non-trainable params: 0
_________________________________________________________________
```

```
# compiling the neural network classifier, sgd optimizer
sgd = optimizers.SGD(lr = 0.01)
# Adding activation function - softmax for multiclass classification
model4.compile(optimizer = sgd, loss = 'categorical_crossentropy', metrics = ['accuracy'])

# Fitting the neural network for training
history = model4.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
Epoch 1/100
210/210 [==============================] - 4s 17ms/step - loss: 2.3271 - accuracy
Epoch 2/100
210/210 [==============================] - 3s 16ms/step - loss: 2.2814 - accuracy
Epoch 3/100
210/210 [==============================] - 3s 16ms/step - loss: 2.2466 - accuracy
Epoch 4/100
210/210 [==============================] - 3s 16ms/step - loss: 2.1966 - accuracy
Epoch 5/100
210/210 [==============================] - 3s 16ms/step - loss: 2.1332 - accuracy
Epoch 6/100
210/210 [==============================] - 3s 16ms/step - loss: 2.0562 - accuracy
Epoch 7/100
210/210 [==============================] - 3s 16ms/step - loss: 1.9809 - accuracy
Epoch 8/100
210/210 [==============================] - 3s 16ms/step - loss: 1.8863 - accuracy
Epoch 9/100
210/210 [==============================] - 3s 17ms/step - loss: 1.7850 - accuracy
Epoch 10/100
210/210 [==============================] - 4s 17ms/step - loss: 1.6829 - accuracy
Epoch 11/100
210/210 [==============================] - 3s 17ms/step - loss: 1.5950 - accuracy
Epoch 12/100
210/210 [==============================] - 3s 16ms/step - loss: 1.5356 - accuracy
Epoch 13/100
210/210 [==============================] - 4s 17ms/step - loss: 1.4512 - accuracy
Epoch 14/100
210/210 [==============================] - 3s 16ms/step - loss: 1.3887 - accuracy
Epoch 15/100
210/210 [==============================] - 3s 16ms/step - loss: 1.3431 - accuracy
Epoch 16/100
210/210 [==============================] - 3s 16ms/step - loss: 1.2864 - accuracy
Epoch 17/100
210/210 [==============================] - 3s 16ms/step - loss: 1.2374 - accuracy
Epoch 18/100
```

```
210/210 [==============================] - 3s 16ms/step - loss: 1.1964 - accuracy
Epoch 19/100
210/210 [==============================] - 3s 16ms/step - loss: 1.1671 - accuracy
Epoch 20/100
210/210 [==============================] - 3s 16ms/step - loss: 1.1486 - accuracy
Epoch 21/100
210/210 [==============================] - 3s 16ms/step - loss: 1.1202 - accuracy
Epoch 22/100
210/210 [==============================] - 4s 17ms/step - loss: 1.0958 - accuracy
Epoch 23/100
210/210 [==============================] - 3s 16ms/step - loss: 1.0726 - accuracy
Epoch 24/100
210/210 [==============================] - 3s 17ms/step - loss: 1.0503 - accuracy
Epoch 25/100
210/210 [==============================] - 3s 16ms/step - loss: 1.0318 - accuracy
Epoch 26/100
210/210 [==============================] - 3s 16ms/step - loss: 0.9966 - accuracy
Epoch 27/100
210/210 [==============================] - 3s 16ms/step - loss: 0.9748 - accuracy
Epoch 28/100
210/210 [==============================] - 3s 16ms/step - loss: 0.9559 - accuracy
Epoch 29/100
210/210 [==============================] - 3s 16ms/step - loss: 0.9541 - accuracy
```

```
print('NN with weight initializers'); print('--'*40)
results4 = model4.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results4[1]*100, 2), '%'))
```

```
NN with weight initializers
--------------------------------------------------------------------------------
1875/1875 [==============================] - 5s 3ms/step - loss: 0.5270 - accuracy:
Validation accuracy: 84.3
```

NN model, ReLU activations, Adam optimizers with weight initializers

```
# compiling the neural network classifier, adam optimizer
adam = optimizers.Adam(lr = 0.001)
# Adding activation function - softmax for multiclass classification
model4.compile(optimizer = adam, loss = 'categorical_crossentropy', metrics = ['accuracy']

# Fitting the neural network for training
history = model4.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
Epoch 1/100
210/210 [==============================] - 4s 18ms/step - loss: 1.4406 - accuracy
Epoch 2/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7367 - accuracy
Epoch 3/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7105 - accuracy
Epoch 4/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6847 - accuracy
Epoch 5/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6831 - accuracy
Epoch 6/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6605 - accuracy
```

```
Epoch 7/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6631 - accuracy
Epoch 8/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6455 - accuracy
Epoch 9/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6288 - accuracy
Epoch 10/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6303 - accuracy
Epoch 11/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5924 - accuracy
Epoch 12/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5845 - accuracy
Epoch 13/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5734 - accuracy
Epoch 14/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5785 - accuracy
Epoch 15/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5640 - accuracy
Epoch 16/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5477 - accuracy
Epoch 17/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5546 - accuracy
Epoch 18/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5105 - accuracy
Epoch 19/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5272 - accuracy
Epoch 20/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5259 - accuracy
Epoch 21/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4974 - accuracy
Epoch 22/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4993 - accuracy
Epoch 23/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4924 - accuracy
Epoch 24/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4901 - accuracy
Epoch 25/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4697 - accuracy
Epoch 26/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4678 - accuracy
Epoch 27/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4773 - accuracy
Epoch 28/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4689 - accuracy
Epoch 29/100
210/210 [==============================] - 4s 17ms/step - loss: 0.4783 - accuracy
```

```
print('NN with weight initializers'); print('--'*40)
results4 = model4.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results4[1]*100, 2), '%'))
```

```
NN with weight initializers
--------------------------------------------------------------------------------
1875/1875 [==============================] - 5s 3ms/step - loss: 0.4104 - accuracy:
Validation accuracy: 88.23
```

Observation:

- Adding weight initialiers didn't result in improvement of score.
- ReLU activations, changing number of activators, Adam optimizers gives the best score out of the ones tried as of now.

Let's try Batch Normalization

Batch Normalization

Batch Normalization, one of the methods to prevent the "internal covariance shift" problem, has proven to be highly effective. Normalize each mini-batch before nonlinearity.

NN model, relu activations, SGD optimizers with weight initializers and batch normalization

```
print('NN model with batch normalization'); print('--'*40)
# Initialize the neural network classifier
model5 = Sequential()

# Input Layer - adding input layer and activation functions relu and weight initializer
model5.add(Dense(256, input_shape = (1024, ), kernel_initializer = 'he_normal'))
# Adding batch normalization
model5.add(BatchNormalization())
# Adding activation function
model5.add(Activation('relu'))

#Hidden Layer 1 - adding first hidden layer
model5.add(Dense(128, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding batch normalization
model5.add(BatchNormalization())
# Adding activation function
model5.add(Activation('relu'))

#Hidden Layer 2 - adding second hidden layer
model5.add(Dense(64, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding batch normalization
model5.add(BatchNormalization())
# Adding activation function
model5.add(Activation('relu'))

#Hidden Layer 3 - adding third hidden layer
model5.add(Dense(32, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding batch normalization
model5.add(BatchNormalization())
# Adding activation function
model5.add(Activation('relu'))

# Output Layer - adding output layer which is of 10 nodes (digits)
model5.add(Dense(10, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding activation function
model5.add(Activation('softmax'))
```

```
    NN model with batch normalization
```

```
--------------------------------------------------------------------------------
```

```
model5.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_15 (Dense)             (None, 256)               262400    
_________________________________________________________________
batch_normalization (BatchNo (None, 256)               1024      
_________________________________________________________________
activation_15 (Activation)   (None, 256)               0         
_________________________________________________________________
dense_16 (Dense)             (None, 128)               32896     
_________________________________________________________________
batch_normalization_1 (Batch (None, 128)               512       
_________________________________________________________________
activation_16 (Activation)   (None, 128)               0         
_________________________________________________________________
dense_17 (Dense)             (None, 64)                8256      
_________________________________________________________________
batch_normalization_2 (Batch (None, 64)                256       
_________________________________________________________________
activation_17 (Activation)   (None, 64)                0         
_________________________________________________________________
dense_18 (Dense)             (None, 32)                2080      
_________________________________________________________________
batch_normalization_3 (Batch (None, 32)                128       
_________________________________________________________________
activation_18 (Activation)   (None, 32)                0         
_________________________________________________________________
dense_19 (Dense)             (None, 10)                330       
_________________________________________________________________
activation_19 (Activation)   (None, 10)                0         
=================================================================
Total params: 307,882
Trainable params: 306,922
Non-trainable params: 960
_________________________________________________________________
```

```
# compiling the neural network classifier, sgd optimizer
sgd = optimizers.SGD(lr = 0.01)
# Adding activation function - softmax for multiclass classification
model5.compile(optimizer = sgd, loss = 'categorical_crossentropy', metrics = ['accuracy'])

# Fitting the neural network for training
history = model5.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
Epoch 1/100
210/210 [==============================] - 5s 20ms/step - loss: 2.4738 - accuracy
Epoch 2/100
210/210 [==============================] - 4s 18ms/step - loss: 1.9336 - accuracy
Epoch 3/100
210/210 [==============================] - 4s 19ms/step - loss: 1.6274 - accuracy
Epoch 4/100
210/210 [==============================] - 4s 19ms/step - loss: 1.4170 - accuracy
Epoch 5/100
210/210 [==============================] - 4s 19ms/step - loss: 1.2611 - accuracy
```

```
Epoch 6/100
210/210 [==============================] - 4s 19ms/step - loss: 1.1453 - accuracy
Epoch 7/100
210/210 [==============================] - 4s 19ms/step - loss: 1.0531 - accuracy
Epoch 8/100
210/210 [==============================] - 4s 19ms/step - loss: 0.9889 - accuracy
Epoch 9/100
210/210 [==============================] - 4s 19ms/step - loss: 0.9332 - accuracy
Epoch 10/100
210/210 [==============================] - 4s 19ms/step - loss: 0.8907 - accuracy
Epoch 11/100
210/210 [==============================] - 4s 19ms/step - loss: 0.8387 - accuracy
Epoch 12/100
210/210 [==============================] - 4s 19ms/step - loss: 0.8076 - accuracy
Epoch 13/100
210/210 [==============================] - 4s 19ms/step - loss: 0.7708 - accuracy
Epoch 14/100
210/210 [==============================] - 4s 19ms/step - loss: 0.7450 - accuracy
Epoch 15/100
210/210 [==============================] - 4s 19ms/step - loss: 0.7151 - accuracy
Epoch 16/100
210/210 [==============================] - 4s 19ms/step - loss: 0.6927 - accuracy
Epoch 17/100
210/210 [==============================] - 4s 18ms/step - loss: 0.6709 - accuracy
Epoch 18/100
210/210 [==============================] - 4s 18ms/step - loss: 0.6592 - accuracy
Epoch 19/100
210/210 [==============================] - 4s 18ms/step - loss: 0.6283 - accuracy
Epoch 20/100
210/210 [==============================] - 4s 19ms/step - loss: 0.6195 - accuracy
Epoch 21/100
210/210 [==============================] - 4s 18ms/step - loss: 0.5937 - accuracy
Epoch 22/100
210/210 [==============================] - 4s 18ms/step - loss: 0.5808 - accuracy
Epoch 23/100
210/210 [==============================] - 4s 19ms/step - loss: 0.5677 - accuracy
Epoch 24/100
210/210 [==============================] - 4s 18ms/step - loss: 0.5479 - accuracy
Epoch 25/100
210/210 [==============================] - 4s 18ms/step - loss: 0.5362 - accuracy
Epoch 26/100
210/210 [==============================] - 4s 19ms/step - loss: 0.5281 - accuracy
Epoch 27/100
210/210 [==============================] - 4s 18ms/step - loss: 0.5125 - accuracy
Epoch 28/100
210/210 [==============================] - 4s 19ms/step - loss: 0.4993 - accuracy
Epoch 29/100
210/210 [==============================] - 4s 19ms/step - loss: 0.4955 - accuracy
```

```
print('NN with batch normalization'); print('--'*40)
results5 = model5.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results5[1]*100, 2), '%'))
```

```
NN with batch normalization
--------------------------------------------------------------------------------
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5233 - accuracy:
Validation accuracy: 85.29
```

NN model, ReLU activations, Adam optimizers with weight initializers and batch normalization

```
# compiling the neural network classifier, adam optimizer
adam = optimizers.Adam(lr = 0.001)
# Adding activation function - softmax for multiclass classification
model5.compile(optimizer = adam, loss = 'categorical_crossentropy', metrics = ['accuracy']

# Fitting the neural network for training
history = model5.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
Epoch 1/100
210/210 [==============================] - 6s 20ms/step - loss: 0.8799 - accuracy
Epoch 2/100
210/210 [==============================] - 4s 19ms/step - loss: 0.5694 - accuracy
Epoch 3/100
210/210 [==============================] - 4s 19ms/step - loss: 0.5365 - accuracy
Epoch 4/100
210/210 [==============================] - 4s 19ms/step - loss: 0.4818 - accuracy
Epoch 5/100
210/210 [==============================] - 4s 19ms/step - loss: 0.4604 - accuracy
Epoch 6/100
210/210 [==============================] - 4s 19ms/step - loss: 0.4360 - accuracy
Epoch 7/100
210/210 [==============================] - 4s 19ms/step - loss: 0.4063 - accuracy
Epoch 8/100
210/210 [==============================] - 4s 19ms/step - loss: 0.3859 - accuracy
Epoch 9/100
210/210 [==============================] - 4s 20ms/step - loss: 0.3858 - accuracy
Epoch 10/100
210/210 [==============================] - 4s 19ms/step - loss: 0.3734 - accuracy
Epoch 11/100
210/210 [==============================] - 4s 19ms/step - loss: 0.3563 - accuracy
Epoch 12/100
210/210 [==============================] - 4s 19ms/step - loss: 0.3435 - accuracy
Epoch 13/100
210/210 [==============================] - 4s 19ms/step - loss: 0.3393 - accuracy
Epoch 14/100
210/210 [==============================] - 4s 19ms/step - loss: 0.3206 - accuracy
Epoch 15/100
210/210 [==============================] - 4s 20ms/step - loss: 0.3214 - accuracy
Epoch 16/100
210/210 [==============================] - 4s 20ms/step - loss: 0.3150 - accuracy
Epoch 17/100
210/210 [==============================] - 4s 21ms/step - loss: 0.2998 - accuracy
Epoch 18/100
210/210 [==============================] - 4s 20ms/step - loss: 0.2991 - accuracy
Epoch 19/100
210/210 [==============================] - 4s 20ms/step - loss: 0.2881 - accuracy
Epoch 20/100
210/210 [==============================] - 4s 20ms/step - loss: 0.2770 - accuracy
Epoch 21/100
210/210 [==============================] - 4s 20ms/step - loss: 0.2791 - accuracy
Epoch 22/100
210/210 [==============================] - 4s 20ms/step - loss: 0.2633 - accuracy
Epoch 23/100
210/210 [==============================] - 4s 20ms/step - loss: 0.2506 - accuracy
Epoch 24/100
210/210 [==============================] - 4s 19ms/step - loss: 0.2464 - accuracy
```

```
Epoch 25/100
210/210 [==============================] - 4s 19ms/step - loss: 0.2428 - accuracy
Epoch 26/100
210/210 [==============================] - 4s 19ms/step - loss: 0.2266 - accuracy
Epoch 27/100
210/210 [==============================] - 4s 19ms/step - loss: 0.2236 - accuracy
Epoch 28/100
210/210 [==============================] - 4s 19ms/step - loss: 0.2216 - accuracy
Epoch 29/100
```

```
print('NN with batch normalization'); print('--'*40)
results5 = model5.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results5[1]*100, 2), '%'))
```

```
NN with batch normalization
--------------------------------------------------------------------------------
1875/1875 [==============================] - 6s 3ms/step - loss: 0.8550 - accuracy:
Validation accuracy: 82.05
```

Observation:

- Batch normalization didn't result in improvement of score.
- ReLU activations, changing number of activators, Adam optimizers achieved the best score.

Let's try Batch Normalization with Dropout

Dropout

NN model, relu activations, SGD optimizers with weight initializers, batch normalization and dropout

```
print('NN model with dropout - sgd optimizer'); print('--'*40)
# Initialize the neural network classifier
model6 = Sequential()
# Input Layer - adding input layer and activation functions relu and weight initializer
model6.add(Dense(512, input_shape = (1024, ), kernel_initializer = 'he_normal'))
# Adding batch normalization
model6.add(BatchNormalization())
# Adding activation function
model6.add(Activation('relu'))
# Adding dropout layer
model6.add(Dropout(0.2))

#Hidden Layer 1 - adding first hidden layer
model6.add(Dense(256, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding batch normalization
model6.add(BatchNormalization())
# Adding activation function
model6.add(Activation('relu'))
```

```
# Adding dropout layer
model6.add(Dropout(0.2))

#Hidden Layer 2 - adding second hidden layer
model6.add(Dense(128, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding batch normalization
model6.add(BatchNormalization())
# Adding activation function
model6.add(Activation('relu'))
# Adding dropout layer
model6.add(Dropout(0.2))

#Hidden Layer 3 - adding third hidden layer
model6.add(Dense(64, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding batch normalization
model6.add(BatchNormalization())
# Adding activation function
model6.add(Activation('relu'))
# Adding dropout layer
model6.add(Dropout(0.2))

#Hidden Layer 4 - adding fourth hidden layer
model6.add(Dense(32, kernel_initializer = 'he_normal', bias_initializer = 'he_uniform'))
# Adding batch normalization
model6.add(BatchNormalization())
# Adding activation function
model6.add(Activation('relu'))
# Adding dropout layer
model6.add(Dropout(0.2))

# Output Layer - adding output layer which is of 10 nodes (digits)
model6.add(Dense(10, kernel_initializer = 'he_normal',bias_initializer = 'he_uniform'))
# Adding activation function
model6.add(Activation('softmax'))
```

```
NN model with dropout - sgd optimizer
--------------------------------------------------------------------------------
```

```
model6.summary()
```

```
Model: "sequential_5"
```

Layer (type)	Output Shape	Param #
dense_20 (Dense)	(None, 512)	524800
batch_normalization_4 (Batch	(None, 512)	2048
activation_20 (Activation)	(None, 512)	0
dropout (Dropout)	(None, 512)	0
dense_21 (Dense)	(None, 256)	131328
batch_normalization_5 (Batch	(None, 256)	1024

```
activation_21 (Activation)   (None, 256)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_22 (Dense)             (None, 128)               32896
_________________________________________________________________
batch_normalization_6 (Batch (None, 128)               512
_________________________________________________________________
activation_22 (Activation)   (None, 128)               0
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_23 (Dense)             (None, 64)                8256
_________________________________________________________________
batch_normalization_7 (Batch (None, 64)                256
_________________________________________________________________
activation_23 (Activation)   (None, 64)                0
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_24 (Dense)             (None, 32)                2080
_________________________________________________________________
batch_normalization_8 (Batch (None, 32)                128
_________________________________________________________________
activation_24 (Activation)   (None, 32)                0
_________________________________________________________________
dropout_4 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_25 (Dense)             (None, 10)                330
_________________________________________________________________
activation_25 (Activation)   (None, 10)                0
=================================================================
Total params: 703,658
Trainable params: 701,674
Non-trainable params: 1,984
_________________________________________________________________
```

```
# compiling the neural network classifier, sgd optimizer
sgd = optimizers.SGD(lr = 0.01)
model6.compile(optimizer = sgd, loss = 'categorical_crossentropy', metrics = ['accuracy'])

# Adding activation function - softmax for multiclass classification
history = model6.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
Epoch 1/100
210/210 [==============================] - 10s 42ms/step - loss: 2.7871 - accurac
Epoch 2/100
210/210 [==============================] - 8s 38ms/step - loss: 2.4490 - accuracy
Epoch 3/100
210/210 [==============================] - 8s 39ms/step - loss: 2.3463 - accuracy
Epoch 4/100
210/210 [==============================] - 8s 38ms/step - loss: 2.2731 - accuracy
Epoch 5/100
210/210 [==============================] - 8s 38ms/step - loss: 2.2140 - accuracy
Epoch 6/100
210/210 [==============================] - 8s 38ms/step - loss: 2.1494 - accuracy
Epoch 7/100
210/210 [==============================] - 8s 38ms/step - loss: 2.0896 - accuracy
Epoch 8/100
```

```
210/210 [==============================] - 8s 38ms/step - loss: 2.0234 - accuracy
Epoch 9/100
210/210 [==============================] - 8s 37ms/step - loss: 1.9597 - accuracy
Epoch 10/100
210/210 [==============================] - 8s 37ms/step - loss: 1.8960 - accuracy
Epoch 11/100
210/210 [==============================] - 8s 37ms/step - loss: 1.8418 - accuracy
Epoch 12/100
210/210 [==============================] - 8s 37ms/step - loss: 1.7897 - accuracy
Epoch 13/100
210/210 [==============================] - 8s 37ms/step - loss: 1.7352 - accuracy
Epoch 14/100
210/210 [==============================] - 8s 37ms/step - loss: 1.6885 - accuracy
Epoch 15/100
210/210 [==============================] - 8s 37ms/step - loss: 1.6424 - accuracy
Epoch 16/100
210/210 [==============================] - 8s 37ms/step - loss: 1.6000 - accuracy
Epoch 17/100
210/210 [==============================] - 8s 37ms/step - loss: 1.5475 - accuracy
Epoch 18/100
210/210 [==============================] - 8s 37ms/step - loss: 1.5152 - accuracy
Epoch 19/100
210/210 [==============================] - 8s 37ms/step - loss: 1.4732 - accuracy
Epoch 20/100
210/210 [==============================] - 8s 37ms/step - loss: 1.4412 - accuracy
Epoch 21/100
210/210 [==============================] - 8s 37ms/step - loss: 1.3988 - accuracy
Epoch 22/100
210/210 [==============================] - 8s 37ms/step - loss: 1.3712 - accuracy
Epoch 23/100
210/210 [==============================] - 8s 37ms/step - loss: 1.3433 - accuracy
Epoch 24/100
210/210 [==============================] - 8s 38ms/step - loss: 1.3174 - accuracy
Epoch 25/100
210/210 [==============================] - 8s 38ms/step - loss: 1.2905 - accuracy
Epoch 26/100
210/210 [==============================] - 8s 38ms/step - loss: 1.2761 - accuracy
Epoch 27/100
210/210 [==============================] - 8s 38ms/step - loss: 1.2486 - accuracy
Epoch 28/100
210/210 [==============================] - 8s 37ms/step - loss: 1.2242 - accuracy
Epoch 29/100
210/210 [==============================] - 8s 38ms/step - loss: 1.2048 - accuracy
```

```
print('NN model with dropout - sgd optimizer'); print('--'*40)
results6 = model6.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results6[1]*100, 2), '%'))
```

```
NN model with dropout - sgd optimizer
--------------------------------------------------------------------------------
1875/1875 [==============================] - 9s 5ms/step - loss: 0.5061 - accuracy:
Validation accuracy: 84.39
```

```
# compiling the neural network classifier, adam optimizer
adam = optimizers.Adam(lr = 0.001)
model6.compile(optimizer = adam, loss = 'categorical_crossentropy', metrics = ['accuracy']
```

```
# Adding activation function - softmax for multiclass classification
history = model6.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,

    Epoch 1/100
    210/210 [==============================] - 10s 40ms/step - loss: 1.1438 - accurac
    Epoch 2/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.9114 - accuracy
    Epoch 3/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.8632 - accuracy
    Epoch 4/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.8052 - accuracy
    Epoch 5/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.7721 - accuracy
    Epoch 6/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.7367 - accuracy
    Epoch 7/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.7128 - accuracy
    Epoch 8/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.6973 - accuracy
    Epoch 9/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.6547 - accuracy
    Epoch 10/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.6577 - accuracy
    Epoch 11/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.6336 - accuracy
    Epoch 12/100
    210/210 [==============================] - 8s 39ms/step - loss: 0.6240 - accuracy
    Epoch 13/100
    210/210 [==============================] - 8s 39ms/step - loss: 0.6207 - accuracy
    Epoch 14/100
    210/210 [==============================] - 8s 40ms/step - loss: 0.5847 - accuracy
    Epoch 15/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.5827 - accuracy
    Epoch 16/100
    210/210 [==============================] - 8s 39ms/step - loss: 0.5841 - accuracy
    Epoch 17/100
    210/210 [==============================] - 8s 39ms/step - loss: 0.5671 - accuracy
    Epoch 18/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.5577 - accuracy
    Epoch 19/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.5412 - accuracy
    Epoch 20/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.5392 - accuracy
    Epoch 21/100
    210/210 [==============================] - 8s 39ms/step - loss: 0.5322 - accuracy
    Epoch 22/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.5238 - accuracy
    Epoch 23/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.5237 - accuracy
    Epoch 24/100
    210/210 [==============================] - 8s 39ms/step - loss: 0.5147 - accuracy
    Epoch 25/100
    210/210 [==============================] - 8s 39ms/step - loss: 0.5036 - accuracy
    Epoch 26/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.4914 - accuracy
    Epoch 27/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.4835 - accuracy
    Epoch 28/100
    210/210 [==============================] - 8s 38ms/step - loss: 0.4745 - accuracy
    Epoch 29/100
    210/210 [==============================] - 8s 39ms/step - loss: 0.4740 - accuracy
```

```
print('NN model with dropout - adam optimizer'); print('--'*40)
results6 = model6.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results6[1]*100, 2), '%'))
```

```
NN model with dropout - adam optimizer
--------------------------------------------------------------------------------
1875/1875 [==============================] - 9s 5ms/step - loss: 0.5062 - accuracy:
Validation accuracy: 83.92
```

Observation:

- Didn't result in any improvement of score.
- NN model, relu activations, SGD optimizers with weight initializers and batch normalization is still the best model.

Let's try Batch Normalization and Dropout with Adam optimizer

Prediction on test dataset using Model 3 - ReLU activations, Adam optimizers

```
print('NN model with relu activations and changing number of activators'); print('--'*40)
# Initialize the neural network classifier
model3 = Sequential()

# Input Layer - adding input layer and activation functions relu
model3.add(Dense(256, input_shape = (1024, )))
# Adding activation function
model3.add(Activation('relu'))

#Hidden Layer 1 - adding first hidden layer
model3.add(Dense(128))
# Adding activation function
model3.add(Activation('relu'))

#Hidden Layer 2 - Adding second hidden layer
model3.add(Dense(64))
# Adding activation function
model3.add(Activation('relu'))

# Output Layer - adding output layer which is of 10 nodes (digits)
model3.add(Dense(10))
# Adding activation function - softmax for multiclass classification
model3.add(Activation('softmax'))
```

```
NN model with relu activations and changing number of activators
--------------------------------------------------------------------------------
```

```
# compiling the neural network classifier, adam optimizer
adam = optimizers.Adam(lr = 0.001)
model3.compile(optimizer = adam, loss = 'categorical_crossentropy', metrics = ['accuracy']

# Fitting the neural network for training
history = model3.fit(X_train, y_train, validation_data = (X_val, y_val), batch_size = 200,
```

```
Epoch 1/100
210/210 [==============================] - 4s 18ms/step - loss: 2.3123 - accuracy
Epoch 2/100
210/210 [==============================] - 4s 17ms/step - loss: 2.0376 - accuracy
Epoch 3/100
210/210 [==============================] - 3s 17ms/step - loss: 1.4453 - accuracy
Epoch 4/100
210/210 [==============================] - 4s 17ms/step - loss: 1.2240 - accuracy
Epoch 5/100
210/210 [==============================] - 3s 16ms/step - loss: 1.1127 - accuracy
Epoch 6/100
210/210 [==============================] - 3s 16ms/step - loss: 1.0129 - accuracy
Epoch 7/100
210/210 [==============================] - 3s 17ms/step - loss: 0.9686 - accuracy
Epoch 8/100
210/210 [==============================] - 3s 17ms/step - loss: 0.9385 - accuracy
Epoch 9/100
210/210 [==============================] - 3s 17ms/step - loss: 0.9025 - accuracy
Epoch 10/100
210/210 [==============================] - 4s 17ms/step - loss: 0.8574 - accuracy
Epoch 11/100
210/210 [==============================] - 4s 17ms/step - loss: 0.8338 - accuracy
Epoch 12/100
210/210 [==============================] - 3s 17ms/step - loss: 0.8057 - accuracy
Epoch 13/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7883 - accuracy
Epoch 14/100
210/210 [==============================] - 3s 17ms/step - loss: 0.7682 - accuracy
Epoch 15/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7554 - accuracy
Epoch 16/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7323 - accuracy
Epoch 17/100
210/210 [==============================] - 4s 17ms/step - loss: 0.7231 - accuracy
Epoch 18/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6864 - accuracy
Epoch 19/100
210/210 [==============================] - 4s 18ms/step - loss: 0.6913 - accuracy
Epoch 20/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6821 - accuracy
Epoch 21/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6554 - accuracy
Epoch 22/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6529 - accuracy
Epoch 23/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6326 - accuracy
Epoch 24/100
210/210 [==============================] - 3s 17ms/step - loss: 0.6199 - accuracy
Epoch 25/100
210/210 [==============================] - 3s 16ms/step - loss: 0.6239 - accuracy
Epoch 26/100
210/210 [==============================] - 4s 17ms/step - loss: 0.6110 - accuracy
Epoch 27/100
```

```
210/210 [==============================] - 4s 17ms/step - loss: 0.5959 - accuracy
Epoch 28/100
210/210 [==============================] - 4s 17ms/step - loss: 0.5851 - accuracy
Epoch 29/100
```

```
print('NN with batch normalization'); print('--'*40)
results3 = model3.evaluate(X_val, y_val)
print('Validation accuracy: {}'.format(round(results3[1]*100, 2), '%'))
```

```
NN with batch normalization
--------------------------------------------------------------------------------
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3955 - accuracy:
Validation accuracy: 88.63
```

```
print('Testing the model on test dataset')
predictions = model3.predict_classes(X_test)
score = model3.evaluate(X_test, y_test)
print('Test loss :', score[0])
print('Test accuracy :', score[1])
```

```
Testing the model on test dataset
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/sequential.py:
  warnings.warn('`model.predict_classes()` is deprecated and '
563/563 [==============================] - 2s 3ms/step - loss: 0.6680 - accuracy: 0.
Test loss : 0.6679835319519043
Test accuracy : 0.8281111121177673
```

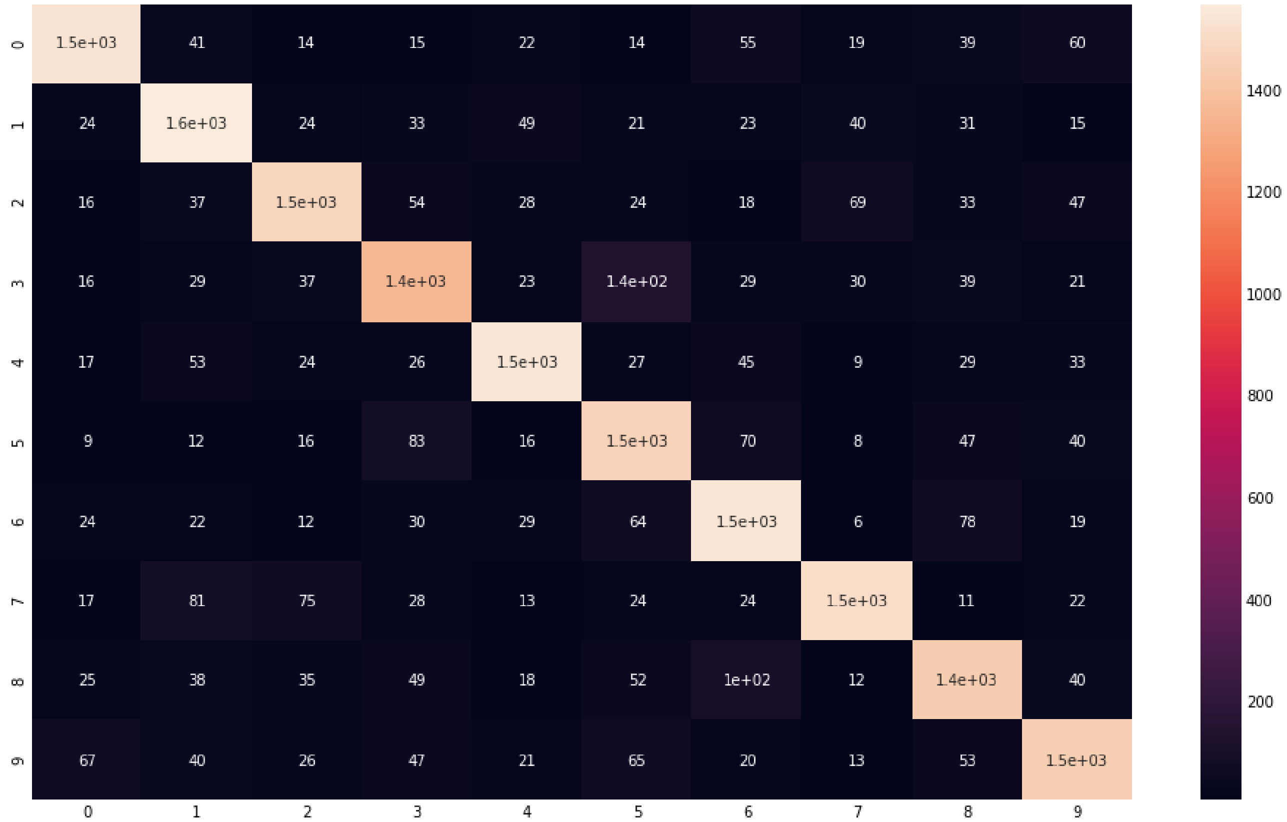
```
print('Classification Report'); print('--'*40)
print(classification_report(y_test_o, predictions))
```

```
Classification Report
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.88      0.85      0.86      1814
           1       0.82      0.86      0.84      1828
           2       0.85      0.82      0.83      1803
           3       0.79      0.79      0.79      1719
           4       0.88      0.85      0.87      1812
           5       0.77      0.83      0.80      1768
           6       0.80      0.84      0.82      1832
           7       0.88      0.84      0.86      1808
           8       0.80      0.80      0.80      1812
           9       0.83      0.80      0.82      1804

    accuracy                           0.83     18000
   macro avg       0.83      0.83      0.83     18000
weighted avg       0.83      0.83      0.83     18000
```

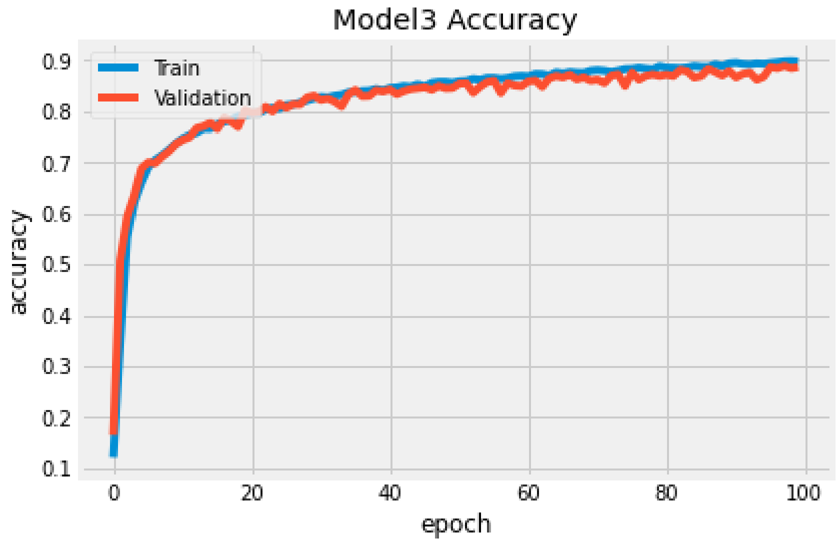
```
print('Confusion matrix')
plt.figure(figsize = (15, 10))
```

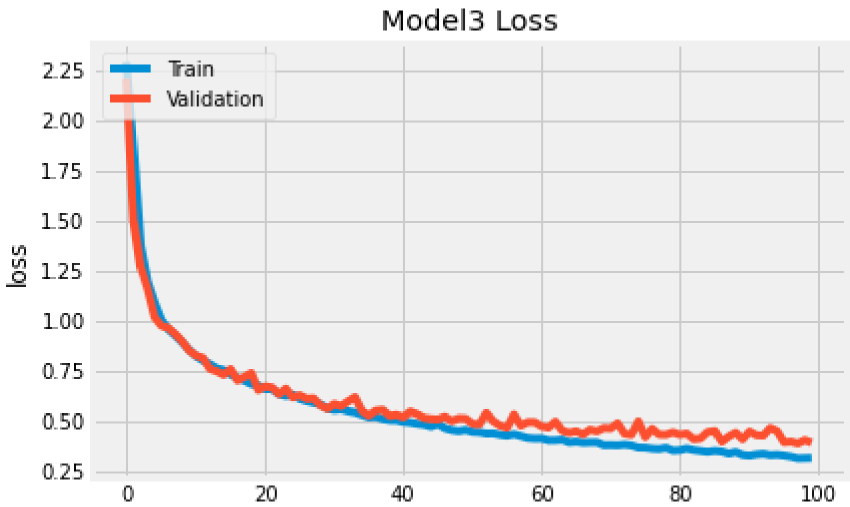
```
sns.heatmap(confusion_matrix(y_test_o, predictions), annot = True)
```

```
Confusion matrix
<matplotlib.axes._subplots.AxesSubplot at 0x7fcef37a0e90>
```




```
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model3 Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['Train', 'Validation'], loc = 'upper left')
plt.show()

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model3 Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['Train', 'Validation'], loc = 'upper left')
plt.show()
```



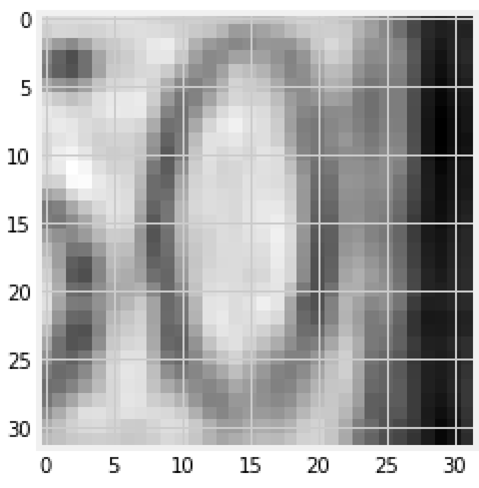



```
model3.predict_classes(X_test)[5]
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/sequential.py:
  warnings.warn('`model.predict_classes()` is deprecated and '
9
```

```
#Showing the image
plt.imshow(X_test[20].reshape(32, 32), cmap = 'gray')
```

```
<matplotlib.image.AxesImage at 0x7fcef2df01d0>
```

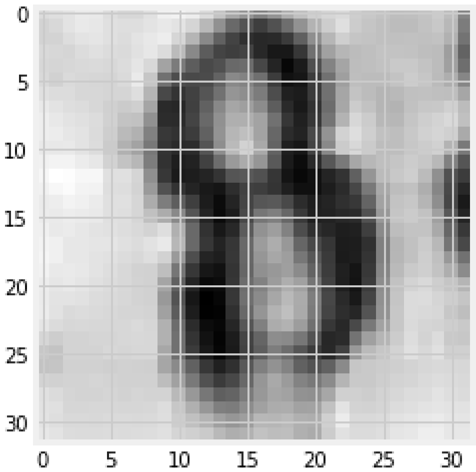


```
model3.predict_classes(X_test)[20]
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/sequential.py:
  warnings.warn('`model.predict_classes()` is deprecated and '
0
```

```
plt.imshow(X_test[10].reshape(32, 32), cmap = 'gray')
```

```
<matplotlib.image.AxesImage at 0x7fcef2d4a8d0>
```




```
model3.predict_classes(X_test)[10]
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/sequential.py:
  warnings.warn('`model.predict_classes()` is deprecated and '
6
```

Conclusion

- Evaluated the accuracy using two methods i.e. baby sitting the NN and NN through API.
- Followed all the required steps starting with loading the datasets to performing hyperparameter optimization and running a finer search by using a finer range.
- Explored different options in optimizers, number of activators, learning rate and activation methods in NN through API.
- Found that baby sitting process achieved the best accuracy of 21% using hyper parameter optimization.
- It might have been further improved but that's the trade off vs time taken to run the script.
- NN through API method achieved best accuracy of 90% on validation set.
- Also printed the classification report, visualized the confusion matrix and summarized history for accuracy and loss.

```
!pip install nbconvert
```

```
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-packages (
```

```
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: nbformat>=4.4 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: jinja2>=2.4 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-packages (fro
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/python3.7/d
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pyparsing>=2.0.2 in /usr/local/lib/python3.7/dist-pac
```

```
!jupyter nbconvert --to html test.ipynb
```

```
--output=<Unicode> (NbConvertApp.output_base)
    Default: ''
    overwrite base name use for output files. can only be used when converting
    one notebook at a time.
--post=<DottedOrNone> (NbConvertApp.postprocessor_class)
    Default: u''
    PostProcessor class used to write the results of the conversion
--config=<Unicode> (JupyterApp.config_file)
    Default: u''
    Full path of a config file.

To see all available configurables, use `--help-all`

Examples
--------

    The simplest way to use nbconvert is

    > jupyter nbconvert mynotebook.ipynb

    which will convert mynotebook.ipynb to the default format (probably HTML).

    You can specify the export format with `--to`.
    Options include ['asciidoc', 'custom', 'html', 'latex', 'markdown', 'notebook


    > jupyter nbconvert --to latex mynotebook.ipynb

    Both HTML and LaTeX support multiple output templates. LaTeX includes
    'base', 'article' and 'report'.  HTML includes 'basic' and 'full'. You
    can specify the flavor of the format used.

    > jupyter nbconvert --to html --template basic mynotebook.ipynb

    You can also pipe the output to stdout, rather than a file

    > jupyter nbconvert mynotebook.ipynb --stdout

    PDF is generated via latex
```

```
        PDF is generated via latex

        > jupyter nbconvert mynotebook.ipynb --to pdf

        You can get (and serve) a Reveal.js-powered slideshow

        > jupyter nbconvert myslides.ipynb --to slides --post serve

        Multiple notebooks can be given at the command line in a couple of
        different ways:

        > jupyter nbconvert notebook*.ipynb
        > jupyter nbconvert notebook1.ipynb notebook2.ipynb

        or you can specify the notebooks list in a config file, containing::

            c.NbConvertApp.notebooks = ["my_notebook.ipynb"]

        > jupyter nbconvert --config mycfg.py
```

```
!jupyter nbconvert --to html test.ipynb
```

```
    --output=<Unicode> (NbConvertApp.output_base)
        Default: ''
        overwrite base name use for output files. can only be used when converting
        one notebook at a time.
    --post=<DottedOrNone> (NbConvertApp.postprocessor_class)
        Default: u''
        PostProcessor class used to write the results of the conversion
    --config=<Unicode> (JupyterApp.config_file)
        Default: u''
        Full path of a config file.

    To see all available configurables, use `--help-all`

    Examples
    --------

        The simplest way to use nbconvert is

        > jupyter nbconvert mynotebook.ipynb

        which will convert mynotebook.ipynb to the default format (probably HTML).

        You can specify the export format with `--to`.
        Options include ['asciidoc', 'custom', 'html', 'latex', 'markdown', 'notebook

        > jupyter nbconvert --to latex mynotebook.ipynb

        Both HTML and LaTeX support multiple output templates. LaTeX includes
        'base', 'article' and 'report'.  HTML includes 'basic' and 'full'. You
        can specify the flavor of the format used.

        > jupyter nbconvert --to html --template basic mynotebook.ipynb

        You can also pipe the output to stdout, rather than a file

        > jupyter nbconvert mynotebook.ipynb --stdout
```

```
PDF is generated via latex

> jupyter nbconvert mynotebook.ipynb --to pdf

You can get (and serve) a Reveal.js-powered slideshow

> jupyter nbconvert myslides.ipynb --to slides --post serve

Multiple notebooks can be given at the command line in a couple of
different ways:

> jupyter nbconvert notebook*.ipynb
> jupyter nbconvert notebook1.ipynb notebook2.ipynb

or you can specify the notebooks list in a config file, containing::

    c.NbConvertApp.notebooks = ["my_notebook.ipynb"]

> jupyter nbconvert --config mycfg.py
```

✓ 0s completed at 6:30 PM