Machine Learning (CSE343/ECE343)

Assignment - 3

Report

Name: Khushdev Pandit
Roll no.: 2020211

Section - A

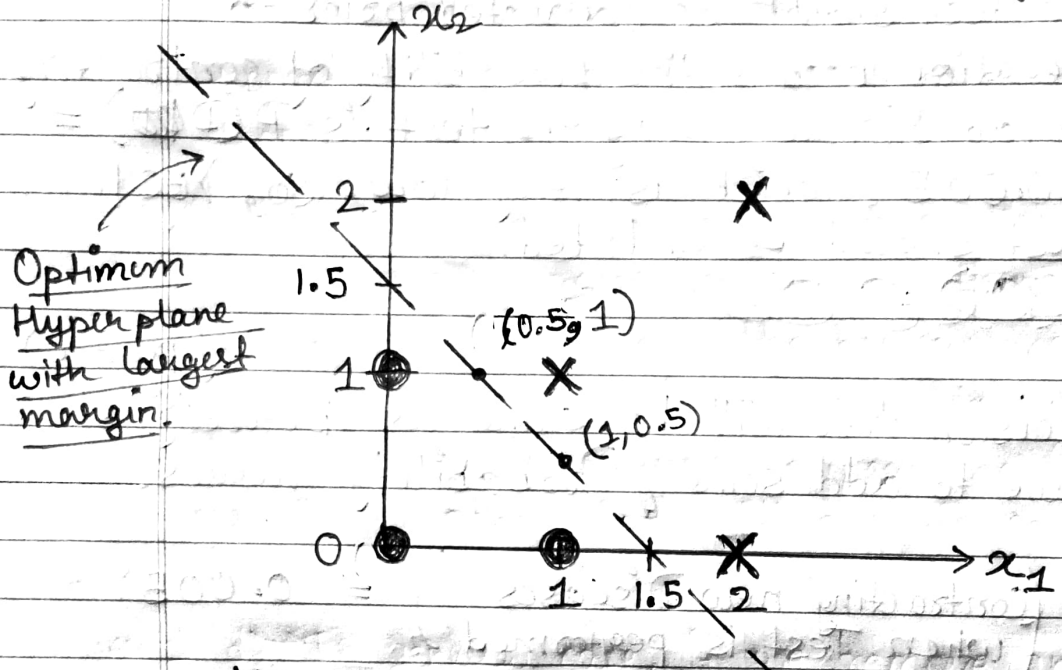
[Theoretical]

Q1

ML - Assignment - 3

Section - A

Answer 1(a) class 'A' examples represented by → ●
class 'B' examples represented by → X



→ Yes, the points are linearly separable.
For example, a line '$x_1 + x_2 = 1.5$' can easily separate two classes 'A' and 'B'.

Answer 1(b) → For each class (A and B), there are two support vectors. (closest samples)
→ So, total of four support vectors are present.

→ These support vectors are:

x_1	x_2	class
1	0	A
2	0	B
0	1	A
1	1	B

⇒ The maximum margin hyperplane will have the largest margin of separation between the supports vectors (of two classes).

⇒ The distance between the closest pairs of support vectors of opposite ~~class~~ classes ~~are~~ is '1'.
So, the margin of ~~spero~~ separation will be $\frac{1}{2} = 0.5$.

⇒ This implies the optimal hyperplane with maximum margin will pass through the points $(x_1, x_2) =$ $(1.5, 0)$, $(0.5, 1)$ and $(1, 0.5)$.

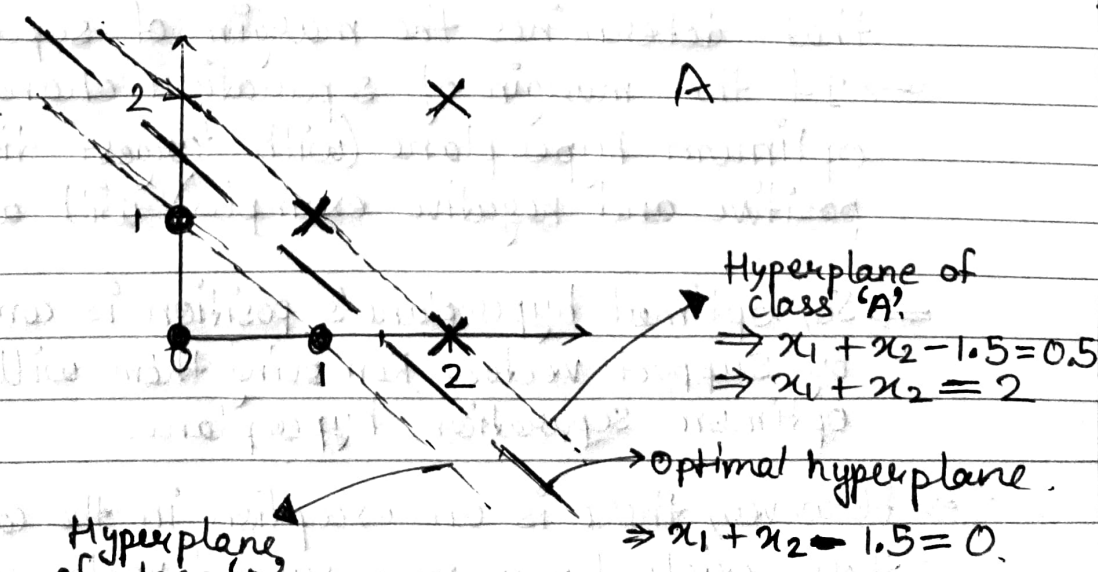
⇒ So, the Optimal hyperplane with maximum margin of separation has equation "$x_1 + x_2 - 1.5 = 0$".

⇒ So, the weight vector corresponding to maximum margin hyperplane is $[W_1 \ W_2] = [1 \ 1]$ and Bias, $b = -1.5$.

$$[W_1 \ W_2] = [1 \ 1]$$

Bias, $b = -1.5$.

Answer 1(c)



→ In this question, even if we remove any one of the support vectors, the optimal hyperplane (decision boundary) with largest margin) will not change.

→ This is because even after removing any one of the support vector, the another remaining support vector (on the hyperplane passing through support vectors) is still going to be there.

→ As there are two supports vectors on each two of the hyperplane, removing one of the support vector, will not affect the hyperplane passing through it. Hence, the margin of separation will remain the same. Hence, the optimal hyperplane (with largest margin of separation) will remain same too.

Answer 1(d) → In general case, if we remove any of the support vectors from the dataset, it will change the position of the optimum hyperplane.

⇒ This is because, it is the support vectors that determines the margin of separation.

⇒ If the margin of separation changes, the optimum hyperplane (with largest distance to positive and negative examples) will also change.

⇒ So, optimal hyperplane's position is completely defined by support vector. Removing them will change the optimum separation hyperplane.

→ However, there is an exception in the case when there exists two or more support vectors on both each of the output labels/classes, same as we saw in question 1(c).

⇒ The optimum separation hyperplane will not change in the exception case, because the margin of separation will not change even after removing any one of the support vectors; as there are multiple support vectors for each output label.

Section - B

[Scratch Implementation]

Q2

In [ ]:

```
import numpy as np
import pandas as pd
import struct
import matplotlib.pyplot as plt
import tensorflow as tf
import warnings
warnings.filterwarnings('ignore')
```

Functions to load Training Data and Testing Data

In [ ]:

```
def load_training_data():
    f_image = open('Data_Q2/train-images.idx3-ubyte', 'rb')
    f_label = open('Data_Q2/train-labels.idx1-ubyte', 'rb')

    magic_number, size, row, column = struct.unpack(">IIII", f_image.read(16))
    image_data = np.fromfile(f_image, dtype=np.uint8)
    print("Training Image Data:")
    print(magic_number, size, row, column)
    print(image_data.shape, '\n')

    magic_number, size = struct.unpack(">II", f_label.read(8))
    image_label = np.fromfile(f_label, dtype=np.uint8)
    print("Training Image Label:")
    print(magic_number, size)
    print(image_label.shape, '\n')

    image_data = np.array(image_data).reshape((size, row, column))
    return image_data, image_label


def load_testing_data():
    f_image = open('Data_Q2/t10k-images.idx3-ubyte', 'rb')
    f_label = open('Data_Q2/t10k-labels.idx1-ubyte', 'rb')

    magic_number, size, row, column = struct.unpack(">IIII", f_image.read(16))
    image_data = np.fromfile(f_image, dtype=np.uint8)
    print("Testing Image Data:")
    print(magic_number, size, row, column)
    print(image_data.shape, '\n')

    magic_number, size = struct.unpack(">II", f_label.read(8))
    image_label = np.fromfile(f_label, dtype=np.uint8)
    print("Testing Image Label:")
    print(magic_number, size)
    print(image_label.shape, '\n')

    image_data = np.array(image_data).reshape((size, row, column))
    return image_data, image_label
```

Loading Training and Testing Data

In [ ]:

```
train_image_data, train_image_label = load_training_data()
test_image_data, test_image_label = load_testing_data()
```

Training Image Data:
2051 60000 28 28
(47040000,)

Training Image Label:
2049 60000
(60000,)

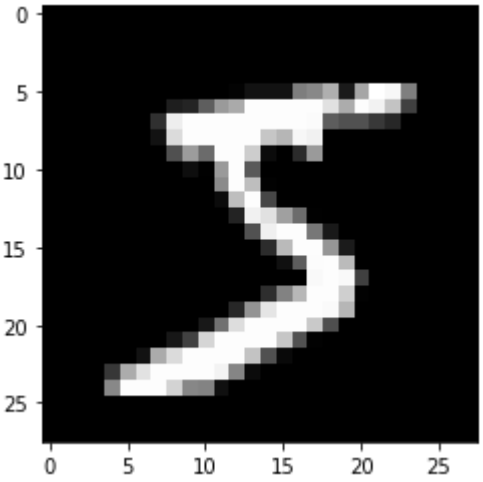
Testing Image Data:
2051 10000 28 28
(7840000,)

Testing Image Label:
2049 10000
(10000,)

Some images and labels in the Training Dataset

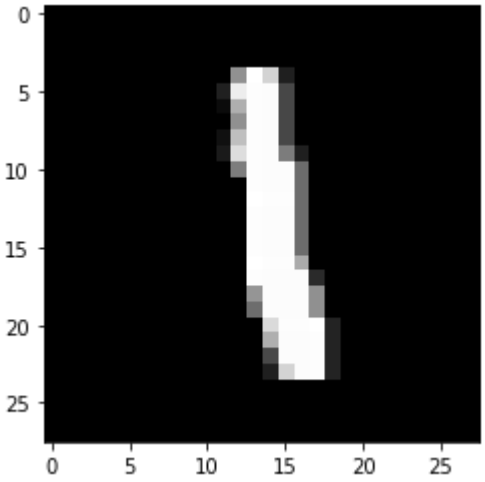
In [ ]:
```
plt.imshow(train_image_data[0], cmap='gray');
print(train_image_label[0])
```

5



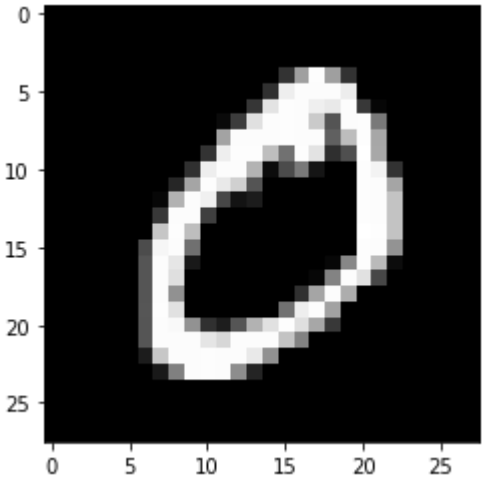
In [ ]:
```
plt.imshow(train_image_data[6], cmap='gray');
print(train_image_label[6])
```

1



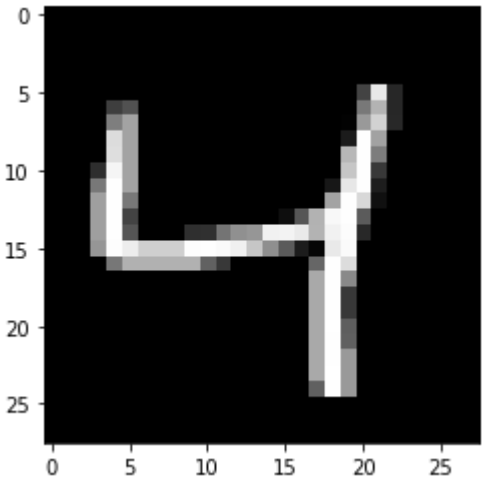
In [ ]:
```
plt.imshow(train_image_data[1], cmap='gray');
print(train_image_label[1])
```

0



In [ ]:
```
plt.imshow(train_image_data[2], cmap='gray');
print(train_image_label[2])
```

4



In [ ]:
```
print(train_image_data.shape)
```

```
print(train_image_label.shape)
print(test_image_data.shape)
print(test_image_label.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

All the outputs/labels of classes

In [ ]:
```
pd.Series(train_image_label).unique()
```

Out[ ]:
```
array([5, 0, 4, 1, 9, 2, 3, 6, 7, 8], dtype=uint8)
```

Running Neural networks on MNSIT Dataset

In [ ]:
```
import numpy as np
import pandas as pd
from NeuralNetwork import NeuralNetwork
import warnings
warnings.filterwarnings("ignore")
```

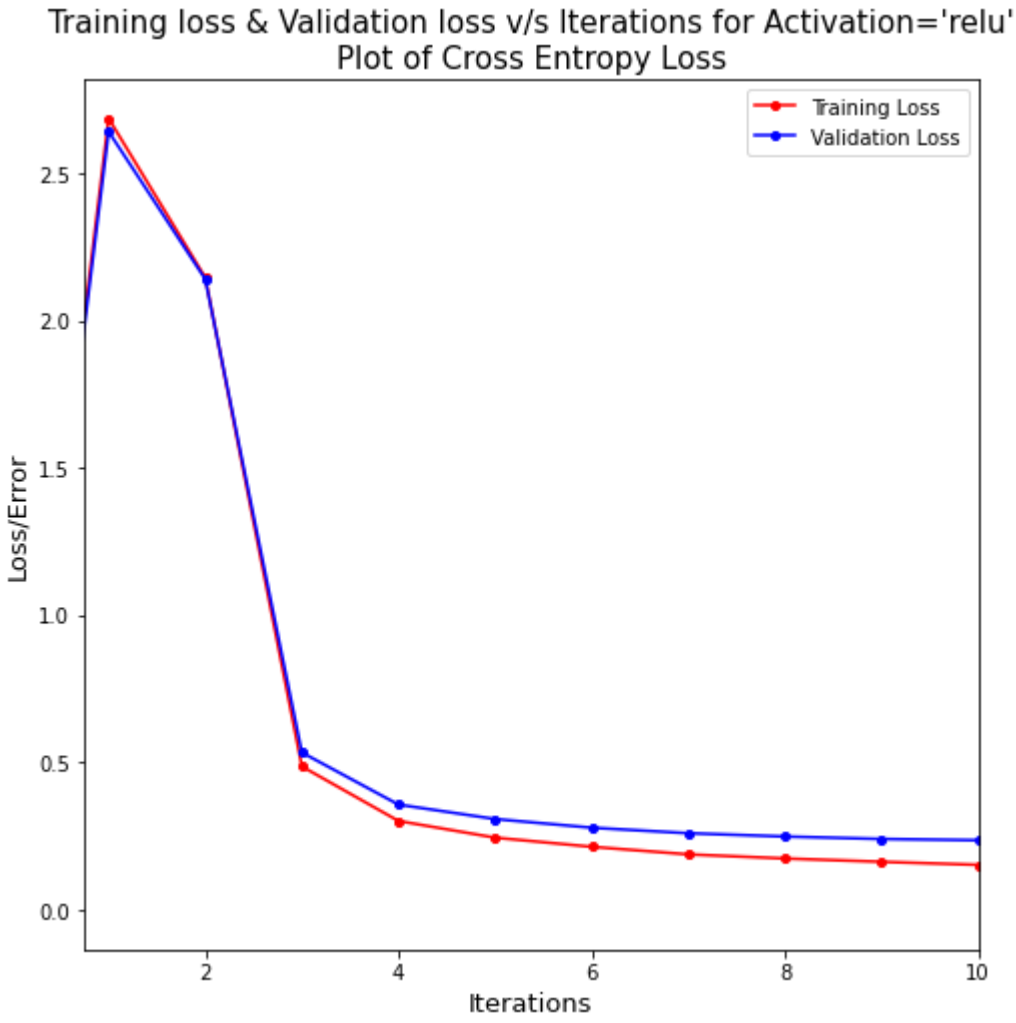
Dividing the Data into Training and Validation dataset

In [ ]:
```
n_classes = 10
n_samples, n_rows, n_cols = train_image_data.shape
n_features = n_rows * n_cols
```

In [ ]:
```
x_train = pd.DataFrame(data=train_image_data.reshape((train_image_data.shape[0], n_rows*n_cols)))
x_test = pd.DataFrame(data=test_image_data.reshape((test_image_data.shape[0], n_rows*n_cols)))
y_train = np.eye(10)[train_image_label]
y_test = np.eye(10)[test_image_label]
```

Running MNSIT dataset on 'Neural network' with Activation function as 'ReLU'

In [ ]:
```
NN_relu = NeuralNetwork(N=4, neuronInEachLayers=[n_features,256,128,64,32,n_classes], lr=1e-4,
                        activation='relu', weightInitType='normal', epochs=10, batchSize=128)

NN_relu.fit(x_train, y_train, x_test, y_test)
```



In [ ]:
```
y_pred = NN_relu.predict(x_test)
y_pred_proba = NN_relu.predict_proba(x_test)
```

Accuracy on Neural Network with ReLU activation function

In [ ]:
```
print("Accuracy on Training Dataset: ", NN_relu.score(x_train, y_train)*100)
print("Accuracy on Validation Dataset: ", NN_relu.score(x_test, y_test)*100)
```

Accuracy on Training Dataset: 98.13499999999999
Accuracy on Validation Dataset: 95.94

Saving the Neural Network with ReLU activation function

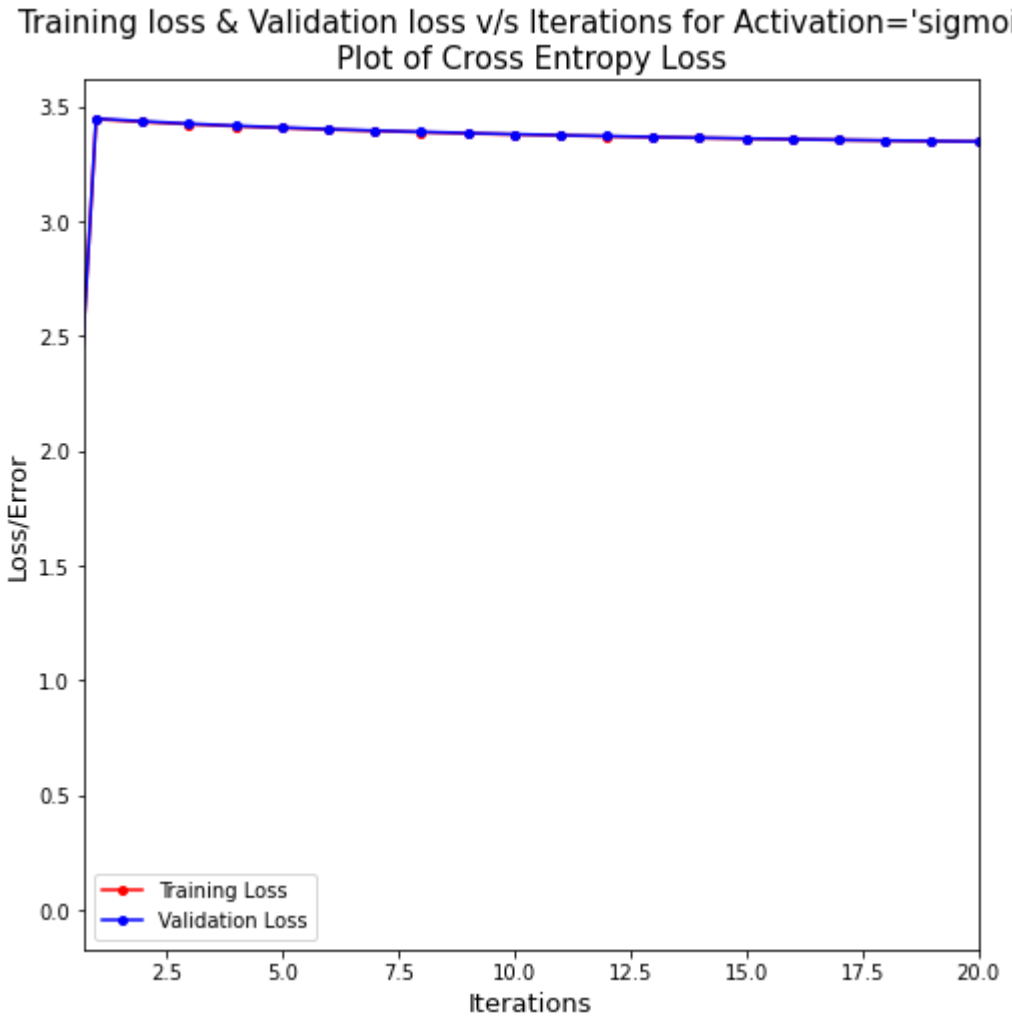
In [ ]:
```
import pickle
with open('Weights/relu.pickle', 'wb') as pickle_out:
    pickle.dump(NN_relu, pickle_out)
```

In [ ]:
```
import pickle
with open('Weights/relu.pickle', 'rb') as pickle_in:
    nn_relu_obj = pickle.load(pickle_in)
    print(nn_relu_obj)
    print(nn_relu_obj.activation.name)
```

```
<NeuralNetwork.NeuralNetwork object at 0x000001D4EDEF3070>
relu
```

Running MNSIT dataset on 'Neural network' with Activation function as 'Sigmoid'

In [ ]:
```
NN_sigmoid = NeuralNetwork(N=4, neuronInEachLayers=[n_features,256,128,64,32,n_classes], lr=1e-5,
                           activation='sigmoid', weightInitType='normal', epochs=20, batchSize=128)

NN_sigmoid.fit(x_train, y_train, x_test, y_test)
```



In [ ]:
```
y_pred = NN_sigmoid.predict(x_test)
y_pred_proba = NN_sigmoid.predict_proba(x_test)
```

Accuracy on Neural Network with Sigmoid activation function

In [ ]:
```
print("Accuracy on Training Dataset: ", NN_sigmoid.score(x_train, y_train)*100)
print("Accuracy on Validation Dataset: ", NN_sigmoid.score(x_test, y_test)*100)
```

```
Accuracy on Training Dataset:  10.756666666666668
Accuracy on Validation Dataset:  10.59
```

Saving the Neural Network with Sigmoid activation function

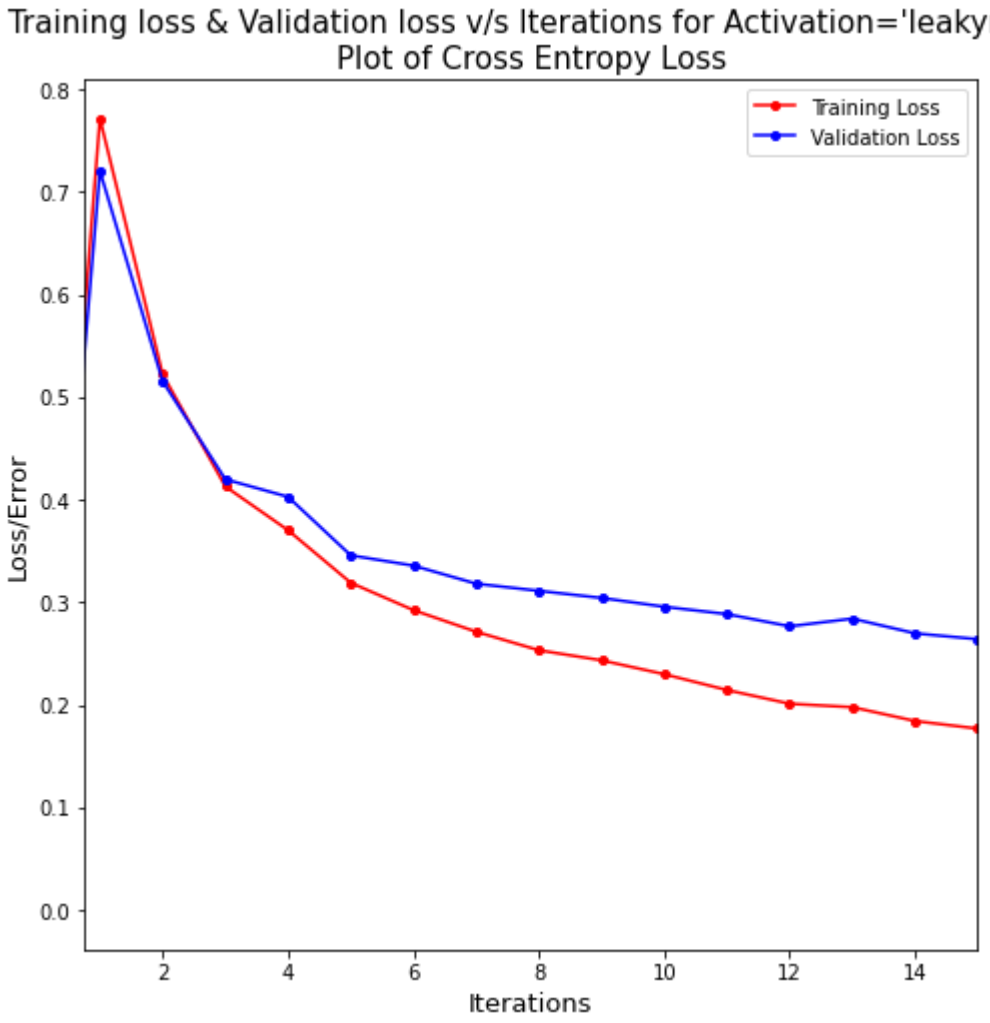
In [ ]:
```
import pickle
with open('Weights/sigmoid.pickle', 'wb') as pickle_out:
    pickle.dump(NN_sigmoid, pickle_out)
```

In [ ]:
```
import pickle
with open('Weights/sigmoid.pickle', 'rb') as pickle_in:
    nn_sigmoid_obj = pickle.load(pickle_in)
    print(nn_sigmoid_obj)
    print(nn_sigmoid_obj.activation.name)
```

```
<NeuralNetwork.NeuralNetwork object at 0x000001D4DC9CECB0>
sigmoid
```

Running MNSIT dataset on 'Neural network' with Activation function as 'Leaky ReLU'

In [ ]:
```
NN_leakyrelu = NeuralNetwork(N=4, neuronInEachLayers=[n_feat ures,256,128,64,32,n_classes], lr=1e-4
                activation='leakyrelu', weightInitType='normal', epochs=15, batchSize=128)

NN_leakyrelu.fit(x_train, y_train, x_test, y_test)
```



In [ ]:
```
y_pred = NN_leakyrelu.predict(x_test)
y_pred_proba = NN_leakyrelu.predict_proba(x_test)
```

Accuracy on Neural Network with Leaky-ReLU activation function

In [ ]:
```
print("Accuracy on Training Dataset: ", NN_leakyrelu.score(x_train, y_train)*100)
print("Accuracy on Validation Dataset: ", NN_leakyrelu.score(x_test, y_test)*100)
```

Accuracy on Training Dataset: 98.08
Accuracy on Validation Dataset: 95.91

Saving the Neural Network with Leaky-ReLU activation function

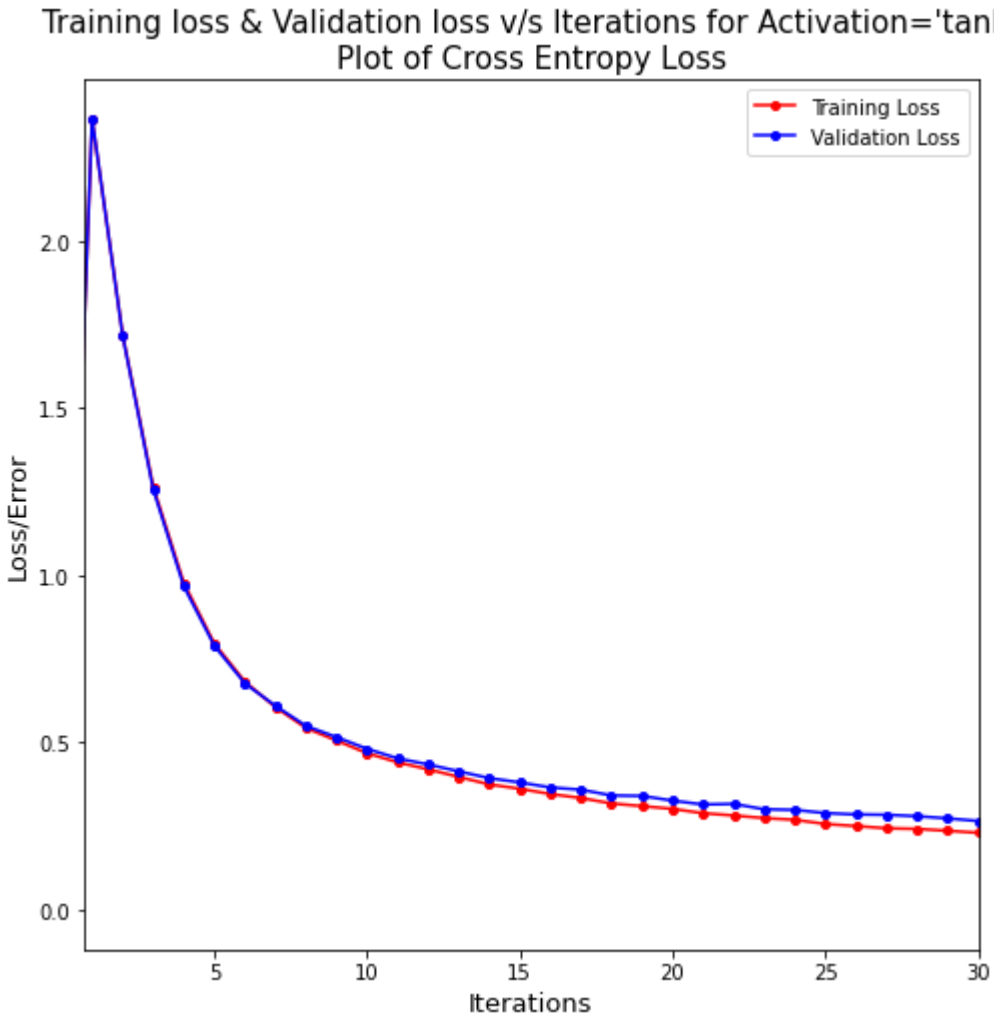
In [ ]:
```
import pickle
with open('Weights/leakyrelu.pickle', 'wb') as pickle_out:
    pickle.dump(NN_leakyrelu, pickle_out)
```

In [ ]:
```
import pickle
with open('Weights/leakyrelu.pickle', 'rb') as pickle_in:
    nn_leakyrelu_obj = pickle.load(pickle_in)
    print(nn_leakyrelu_obj)
    print(nn_leakyrelu_obj.activation.name)
```

```
<NeuralNetwork.NeuralNetwork object at 0x000001D4ED6E7D00>
leakyrelu
```

Running MNSIT dataset on 'Neural network' with Activation function as 'TanH'

In [ ]:
```
NN_tanh = NeuralNetwork(N=4, neuronInEachLayers=[n_features,256,128,64,32,n_classes], lr=1e-4,
                        activation='tanh', weightInitType='normal', epochs=30, batchSize=128)

NN_tanh.fit(x_train, y_train, x_test, y_test)
```



In [ ]:
```
y_pred = NN_tanh.predict(x_test)
y_pred_proba = NN_tanh.predict_proba(x_test)
```

Accuracy on Neural Network with TanH activation function

In [ ]:
```
print("Accuracy on Training Dataset: ", NN_tanh.score(x_train, y_train)*100)
print("Accuracy on Validation Dataset: ", NN_tanh.score(x_test, y_test)*100)
```

```
Accuracy on Training Dataset:  96.67166666666667
Accuracy on Validation Dataset:  95.38
```

Saving the Neural Network with TanH activation function

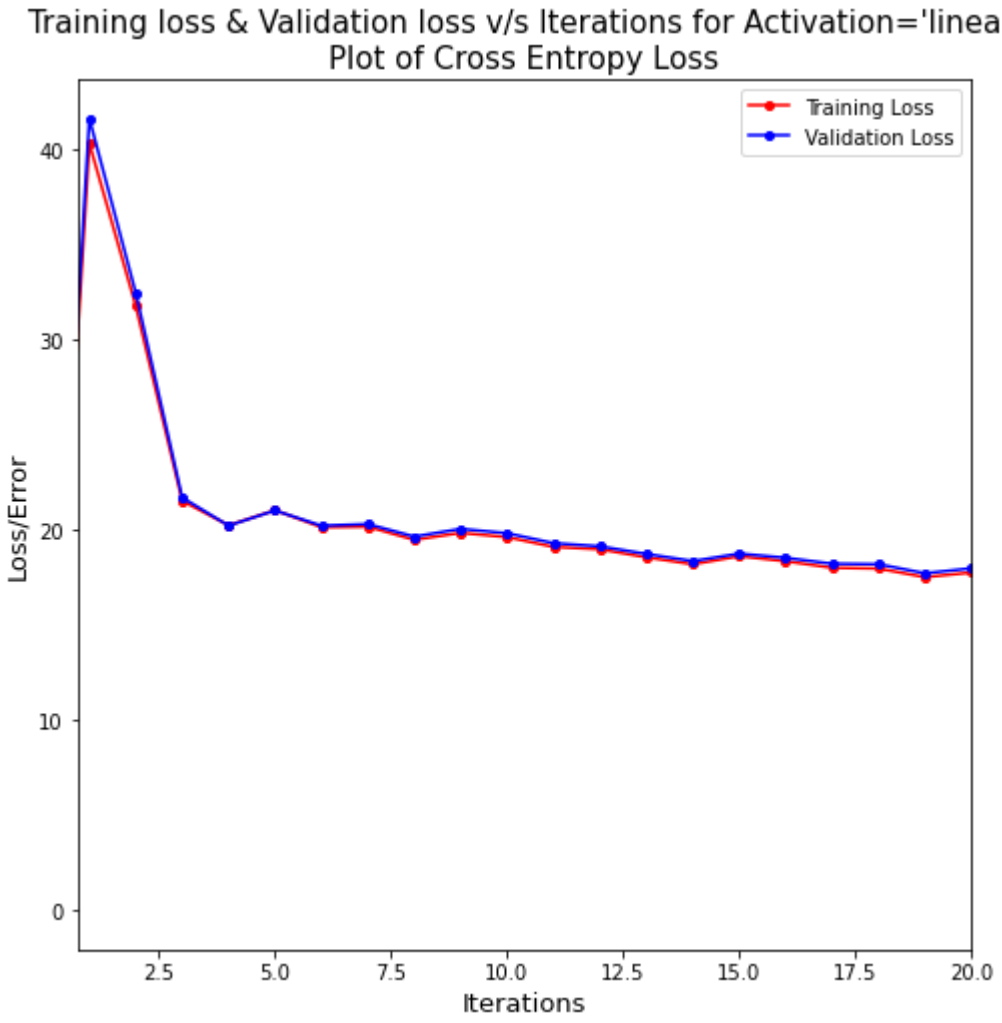
In [ ]:
```
import pickle
with open('Weights/tanh.pickle', 'wb') as pickle_out:
    pickle.dump(NN_tanh, pickle_out)
```

In [ ]:
```
import pickle
with open('Weights/tanh.pickle', 'rb') as pickle_in:
    nn_tanh_obj = pickle.load(pickle_in)
    print(nn_tanh_obj)
    print(nn_tanh_obj.activation.name)
```

```
<NeuralNetwork.NeuralNetwork object at 0x000001D4824E9E70>
tanh
```

Running MNSIT dataset on 'Neural network' with Activation function as 'linear/identity'

In [ ]:
```
NN_linear = NeuralNetwork(N=4, neuronInEachLayers=[n_features,256,128,64,32,n_classes], lr=1e-5,
                          activation='linear', weightInitType='normal', epochs=20, batchSize=128)

NN_linear.fit(x_train, y_train, x_test, y_test)
```



In [ ]:
```
y_pred = NN_linear.predict(x_test)
y_pred_proba = NN_linear.predict_proba(x_test)
```

Accuracy on Neural Network with Linear activation function

In [ ]:
```
print("Accuracy on Training Dataset: ", NN_linear.score(x_train, y_train)*100)
print("Accuracy on Validation Dataset: ", NN_linear.score(x_test, y_test)*100)
```

Accuracy on Training Dataset: 81.11333333333334
Accuracy on Validation Dataset: 80.74

Saving the Neural Network with Linear activation function

In [ ]:
```
import pickle
with open('Weights/linear.pickle', 'wb') as pickle_out:
    pickle.dump(NN_linear, pickle_out)
```

In [ ]:
```
import pickle
with open('Weights/linear.pickle', 'rb') as pickle_in:
    nn_linear_obj = pickle.load(pickle_in)
    print(nn_linear_obj)
    print(nn_linear_obj.activation.name)
```

<NeuralNetwork.NeuralNetwork object at 0x000001D4DC9CDCF0>
linear

Section - C

[Inbuilt Algorithms]

Q3

In [ ]:

```
import os
import numpy as np
import pandas as pd
import struct
import matplotlib.pyplot as plt
import tensorflow as tf
import warnings
warnings.filterwarnings('ignore')
```

Functions to load Training Data and Testing Data

In [ ]:

```
def load_training_data():
    f_image = open('Data_Q3/train-images-idx3-ubyte', 'rb')
    f_label = open('Data_Q3/train-labels-idx1-ubyte', 'rb')

    magic_number, size, row, column = struct.unpack(">IIII", f_image.read(16))
    image_data = np.fromfile(f_image, dtype=np.uint8)
    print("Training Image Data:")
    print(magic_number, size, row, column)
    print(image_data.shape, '\n')

    magic_number, size = struct.unpack(">II", f_label.read(8))
    image_label = np.fromfile(f_label, dtype=np.uint8)
    print("Training Image Label:")
    print(magic_number, size)
    print(image_label.shape, '\n')

    image_data = np.array(image_data).reshape((size, row, column))
    return image_data, image_label


def load_testing_data():
    f_image = open('Data_Q3/t10k-images-idx3-ubyte', 'rb')
    f_label = open('Data_Q3/t10k-labels-idx1-ubyte', 'rb')

    magic_number, size, row, column = struct.unpack(">IIII", f_image.read(16))
    image_data = np.fromfile(f_image, dtype=np.uint8)
    print("Testing Image Data:")
    print(magic_number, size, row, column)
    print(image_data.shape, '\n')

    magic_number, size = struct.unpack(">II", f_label.read(8))
    image_label = np.fromfile(f_label, dtype=np.uint8)
    print("Testing Image Label:")
    print(magic_number, size)
    print(image_label.shape, '\n')

    image_data = np.array(image_data).reshape((size, row, column))
    return image_data, image_label
```

Loading Training and Testing Data

In [ ]:

```
train_image_data, train_image_label = load_training_data()
test_image_data, test_image_label = load_testing_data()
```

Training Image Data:
2051 60000 28 28
(47040000,)

Training Image Label:
2049 60000
(60000,)

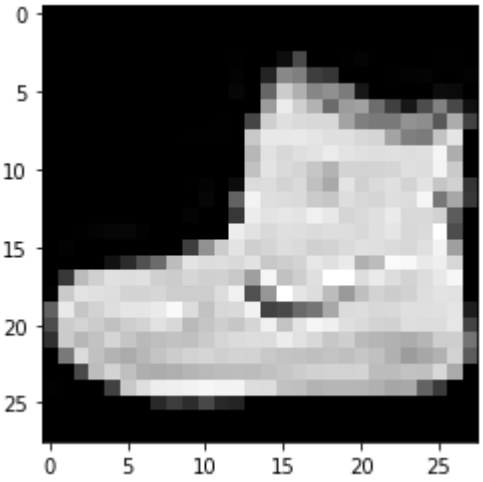
Testing Image Data:
2051 10000 28 28
(7840000,)

Testing Image Label:
2049 10000
(10000,)

Some images and labels in the Training Dataset

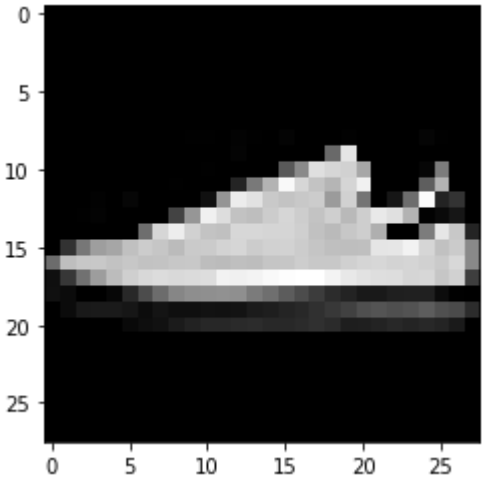
In [ ]:
```
plt.imshow(train_image_data[0], cmap='gray');
print(train_image_label[0])
```

9



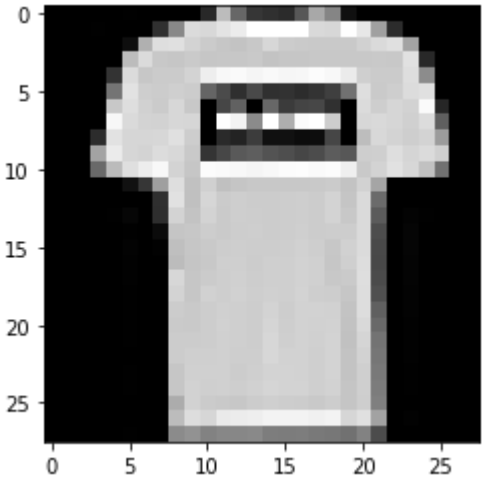
In [ ]:
```
plt.imshow(train_image_data[6], cmap='gray');
print(train_image_label[6])
```

7



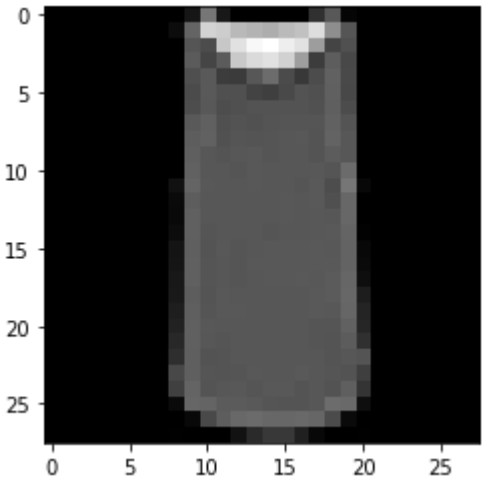
In [ ]:
```
plt.imshow(train_image_data[1], cmap='gray');
print(train_image_label[1])
```

0



In [ ]:
```
plt.imshow(train_image_data[2], cmap='gray');
print(train_image_label[2])
```

0



Q3 (Section-C)

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import log_loss
from sklearn.neural_network import MLPClassifier
```

Dividing the Data into Training and Validation dataset

In [ ]:
```
train_image, valid_image, train_label, valid_label = train_test_split(
                    train_image_data, train_image_label, test_size=0.15)
```

In [ ]:
```
print(train_image.shape)
print(valid_image.shape)
print(train_label.shape)
print(valid_label.shape)
```

```
(51000, 28, 28)
(9000, 28, 28)
(51000,)
(9000,)
```

All the output label classes

In [ ]:
```
pd.Series(train_image_label).unique()
```

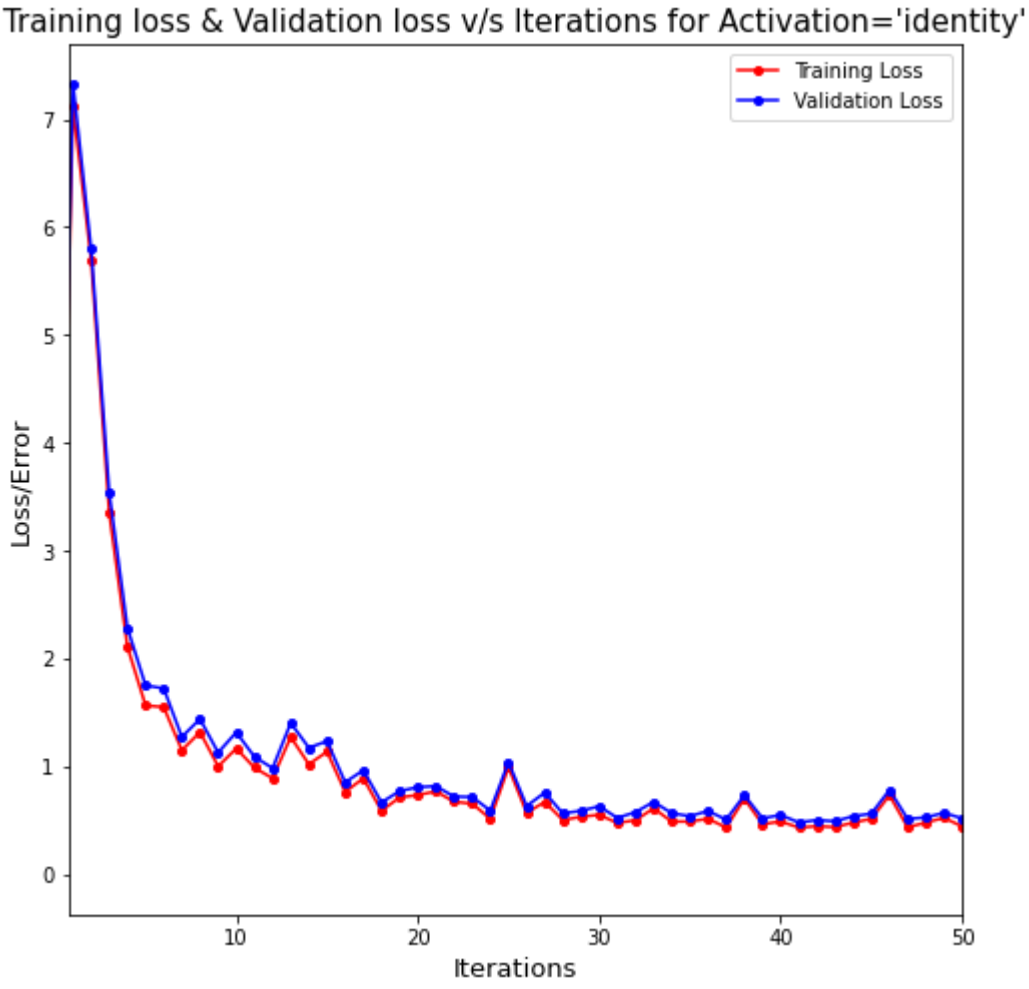
Out[ ]:
```
array([9, 0, 3, 2, 7, 5, 1, 6, 4, 8], dtype=uint8)
```

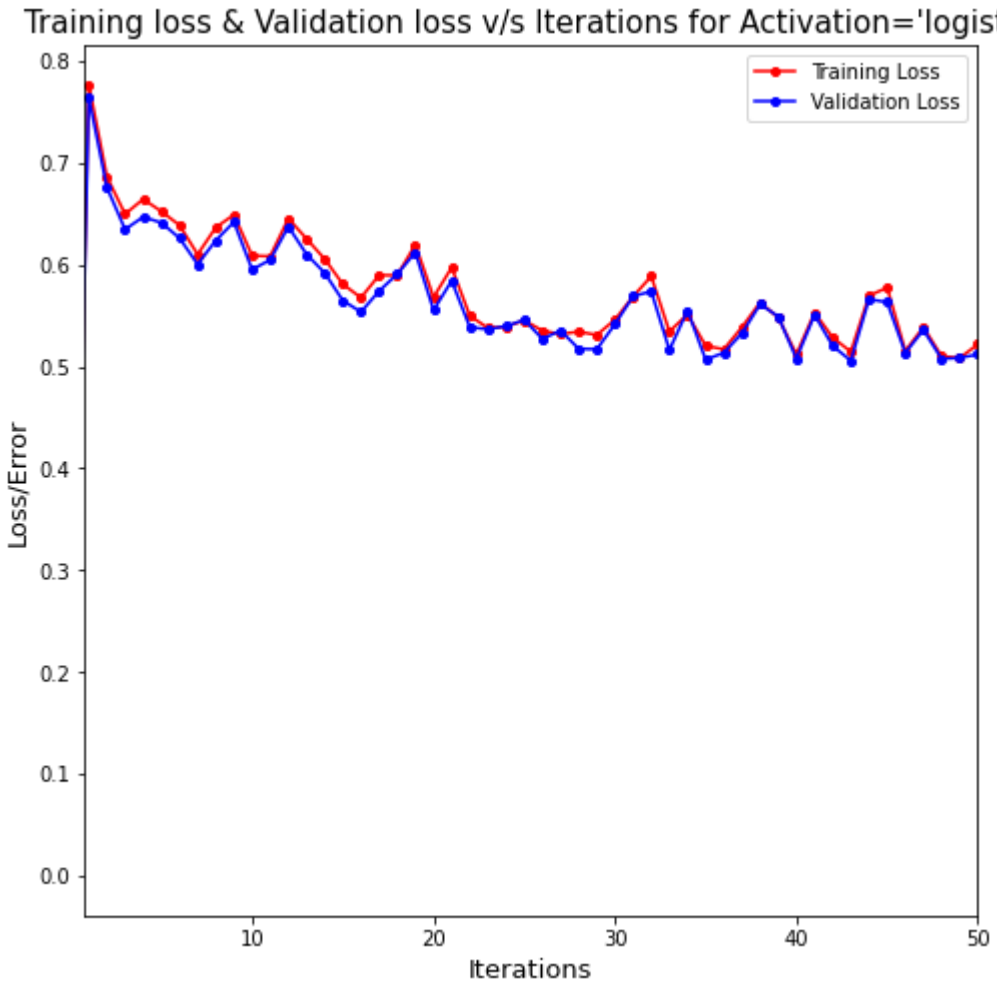
Part (a)

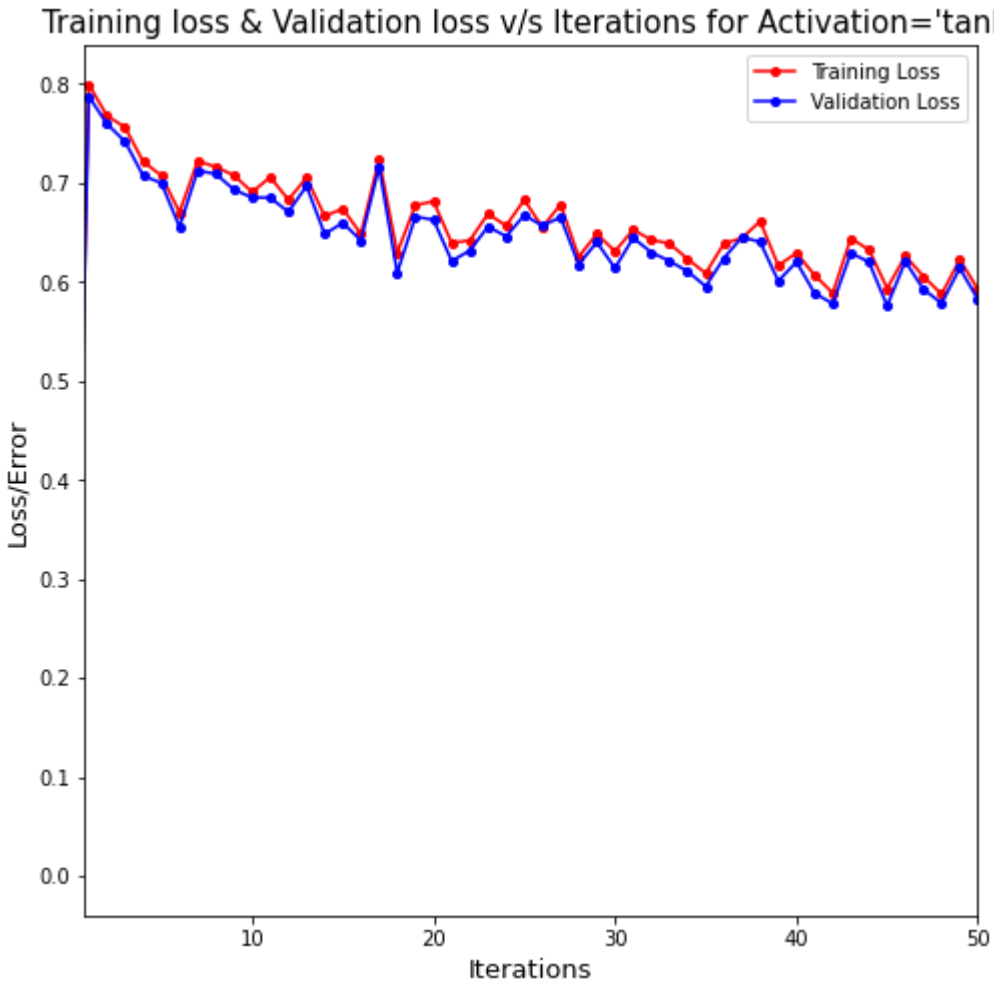
Plot Training loss v/s epochs and validation loss v/s epochs for activations simgoid, ReLU, tanh and linear (with default learning rate).

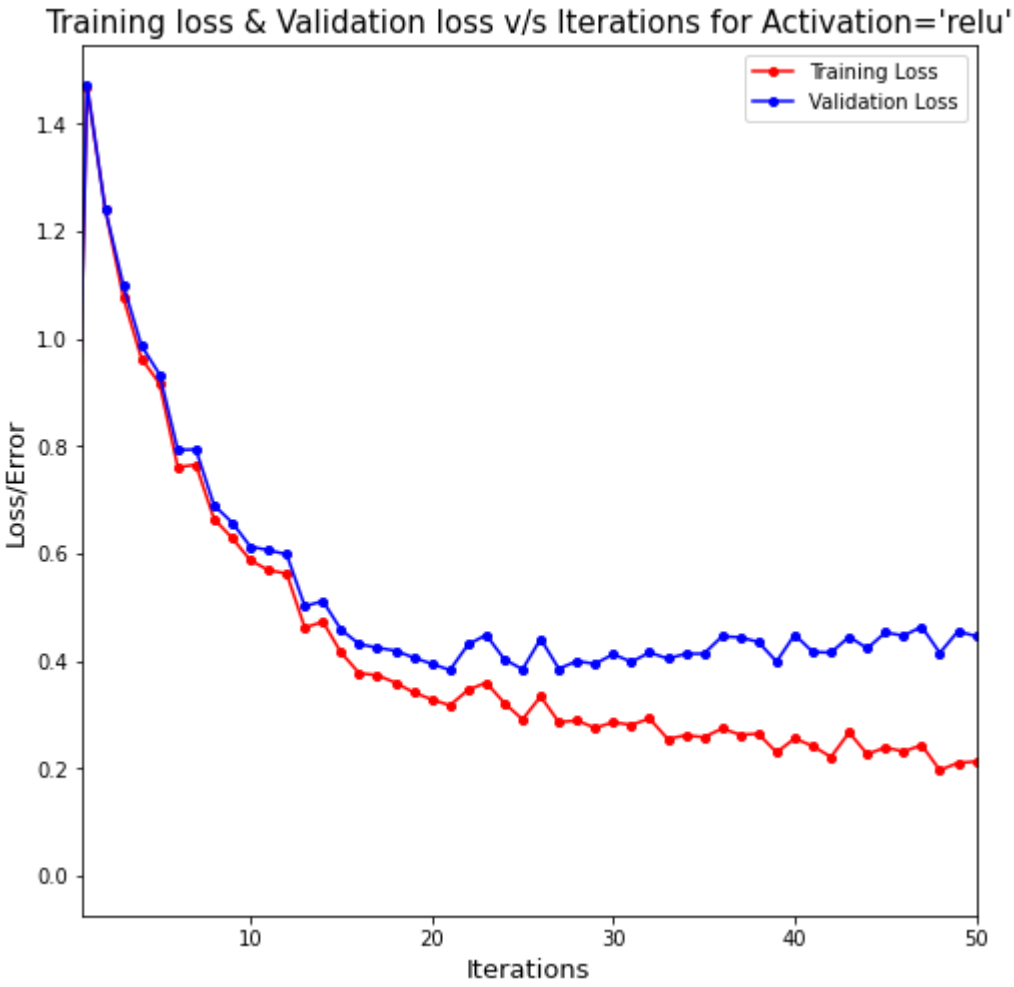
In [ ]:
```
def MLP_activations(activation:str, epochs, batch_size, x_train, y_train, x_valid, y_valid):
    train_loss, validation_loss = [0], [0]

    # Neural network model
    mlp = MLPClassifier(hidden_layer_sizes=(256,32), activation=activation, batch_size=batch_size)

    # Run epochs
    for epoch in range(epochs):
        # Train the Neural network model using partial_fit()
        nsamples, nrows, ncols = x_train.shape
        mlp.partial_fit(x_train.reshape((nsamples, nrows*ncols)), y_train, classes=np.arange(10))

        # Compute Training Loss
        nsamples, nrows, ncols = x_train.shape
        y_true = np.eye(10)[y_train]
        y_pred_proba = mlp.predict_proba(x_train.reshape((nsamples, nrows*ncols)))
        train_loss.append(log_loss(y_true, y_pred_proba))

        # Compute Validation Loss
```

```
        nsamples, nrows, ncols = x_valid.shape
        y_true = np.eye(10)[y_valid]
        y_pred_proba = mlp.predict_proba(x_valid.reshape((nsamples, nrows*ncols)))
        validation_loss.append(log_loss(y_true, y_pred_proba))

    # Plot training & Validation loss
    plt.figure(figsize=(8,8))
    plt.plot(train_loss, 'r-o', label='Training Loss', ms=4)
    plt.plot(validation_loss, 'b-o', label='Validation Loss', ms=4)
    plt.xlabel('Iterations',fontsize=13)
    plt.ylabel('Loss/Error',fontsize=13)
    plt.xlim((0.75,epochs))
    plt.title(f"Training loss & Validation loss v/s Iterations for Activation='{activation}'",fontsize=15)
    plt.legend()
    plt.show()
    return mlp

# Batch size as 128 & epochs as 50
epochs = 50
batch_size = 128
models_activation = []

# Try all Neural networks with all the activation functions
for activations in ['identity', 'logistic', 'tanh', 'relu']:
    mlp = MLP_activations(activations, epochs, batch_size, train_image, train_label, valid_image, valid_lab
    models_activation.append(mlp)
```




Training loss & Validation loss v/s Iterations for Activation='logistic'
Training Loss
Validation Loss
Loss/Error
Iterations
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
10
20
30
40
50


Training loss & Validation loss v/s Iterations for Activation='tanh'
Training Loss
Validation Loss
Loss/Error
Iterations
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
10
20
30
40
50



Accuracy on Neural Networks (MLP) with activation function ['identity', 'logistic', 'tanh', 'relu']

In [ ]:

```
activations = ['identity', 'logistic', 'tanh', 'relu']

for i in range(len(activations)):
    print(f"==> Accuracy on activation fucntion = '{activations[i]}'")
    nsamples, nrows, ncols = train_image.shape
    print(f"Training Accuracy :", models_activation[i].score(train_image.reshape((nsamples, nrows*ncols)),
                                                             train_label))
    nsamples, nrows, ncols = valid_image.shape
    print(f"Validation Accuracy :", models_activation[i].score(valid_image.reshape((nsamples, nrows*ncols)),
                                                               valid_label))
    print()
```

==> Accuracy on activation fucntion = 'identity'
Training Accuracy : 0.8472549019607843
Validation Accuracy : 0.8371111111111111

==> Accuracy on activation fucntion = 'logistic'
Training Accuracy : 0.8115490196078431
Validation Accuracy : 0.8167777777777778

==> Accuracy on activation fucntion = 'tanh'
Training Accuracy : 0.7748039215686274
Validation Accuracy : 0.7775555555555556

==> Accuracy on activation fucntion = 'relu'
Training Accuracy : 0.9280392156862745
Validation Accuracy : 0.8836666666666667

Observation: Best activation function is ReLU

1) ReLU is observed to be the Best activation function.

```
This is beacuse RelU doesn't activates all the neurons at the same
time.
When the weighted sum of inputs to a neuron in any layer is negative,
then ReLU outputs zero (0). Thus, making that neuron's contribution in
the weighted sum of input to neurons of next layer to be zero. Hence,
RelU doesn't activates all the neurons at the same time.
Also, ReLU has high value of derivative (1 for x > 0), so it helps to
avoid gradient vanishing. This is neccessary for weights updation
because weights are updated using gradients of each layers. Due to this
ReLU networks learns faster, hence shows the best performance. Its loss
curve declines very sharply than any other activation function. Hence,
it shows fast convergence.
```

2) Linear activation is observed to be the 2nd Best activation function after ReLU.

```
Linear activation function activates all the neurons at the same time.
Beacuse, the weighted sum of inputs to a neuron (in any layer) same as
the output of that neuron due to Linear function. (y = x). So, next
layer will receive contributions from all the neurons in previous
layers. Hence, all neurons are activated.
Also, Linear function has high value of derivative (i.e, 1), so weights
will be updated faster beacuse gradients will be high. Its loss curve
also declines sharply. Hence, it shows fast convergence and the best
performance.
```

3) Sigmoid Function proved to be the 3rd Best activation function.

```
Neural networks learns slower with sigmoid activation function.
This is simply beacuse of 'vanishing gradient problem' with the sigmoid
activation function. The sigmoid activation function has derivative
curve/plot with very small value. Due to this the gradients will be
```

small, and weights will be updated very slowly. So, weight doesn't get updated much and leads to very slow convergence in Training and Validation loss. Hence, network will take lot of time to learn the pattern.

4) TanH is the 4th Best activation function.

TanH activation has many similarity with the sigmoid activation functions, due to this its results are quite analogous to the sigmoid activation function. This function also shows poor performance and convergence pattern in Training & Validation loss analogous to sigmoid function.
Gradient values are somewhat stronger for TanH than sigmoid, beacuse derivatives are steeper, hence weights will be updated little faster than sigmoid; but still it faces the problem of 'vanishing gradient' beacuse the gradient are high and non-zero only for small range of values i,e, [-1, 1].
Due to this the gradients will be small, and weights will be updated very slowly. So, weight doesn't get updated much and leads to very slow convergence in Training and Validation loss. Hence, network will take lot of time to learn the pattern. Hence shows poor performance.

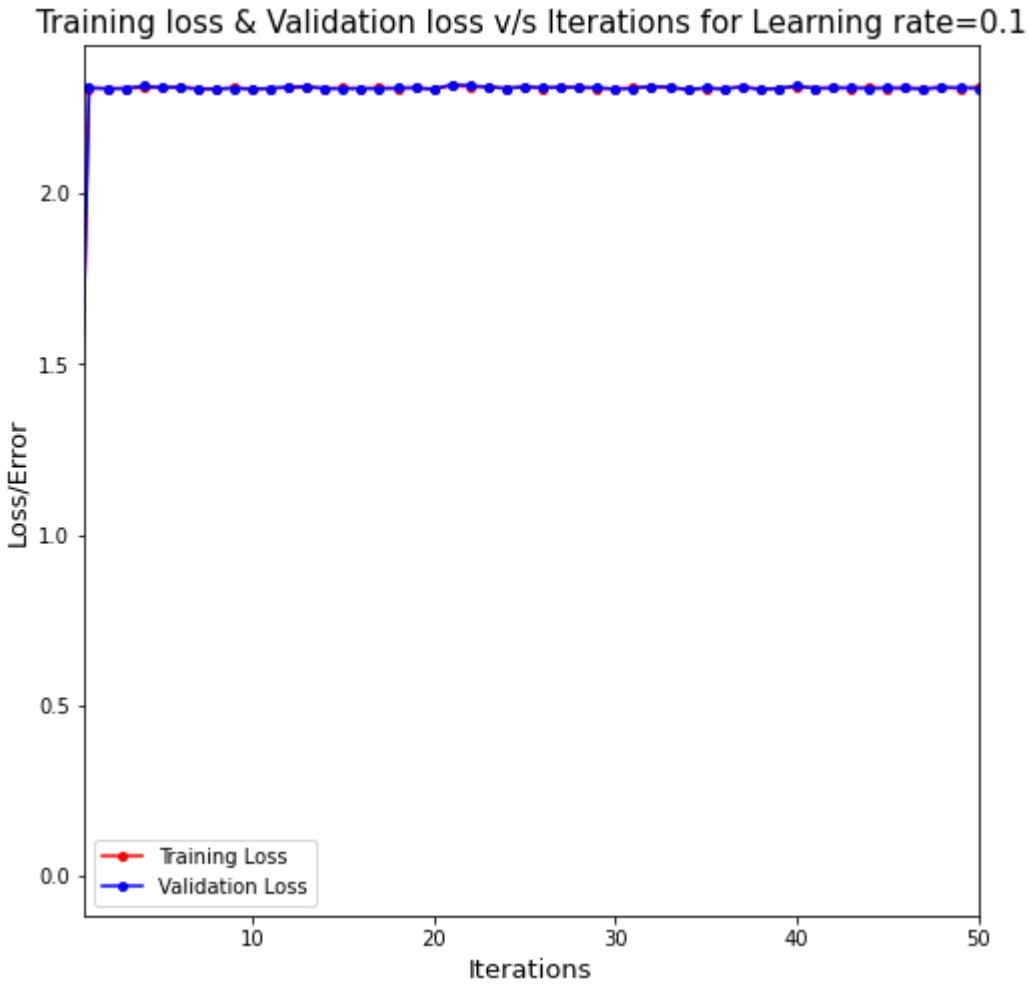
Part (b)

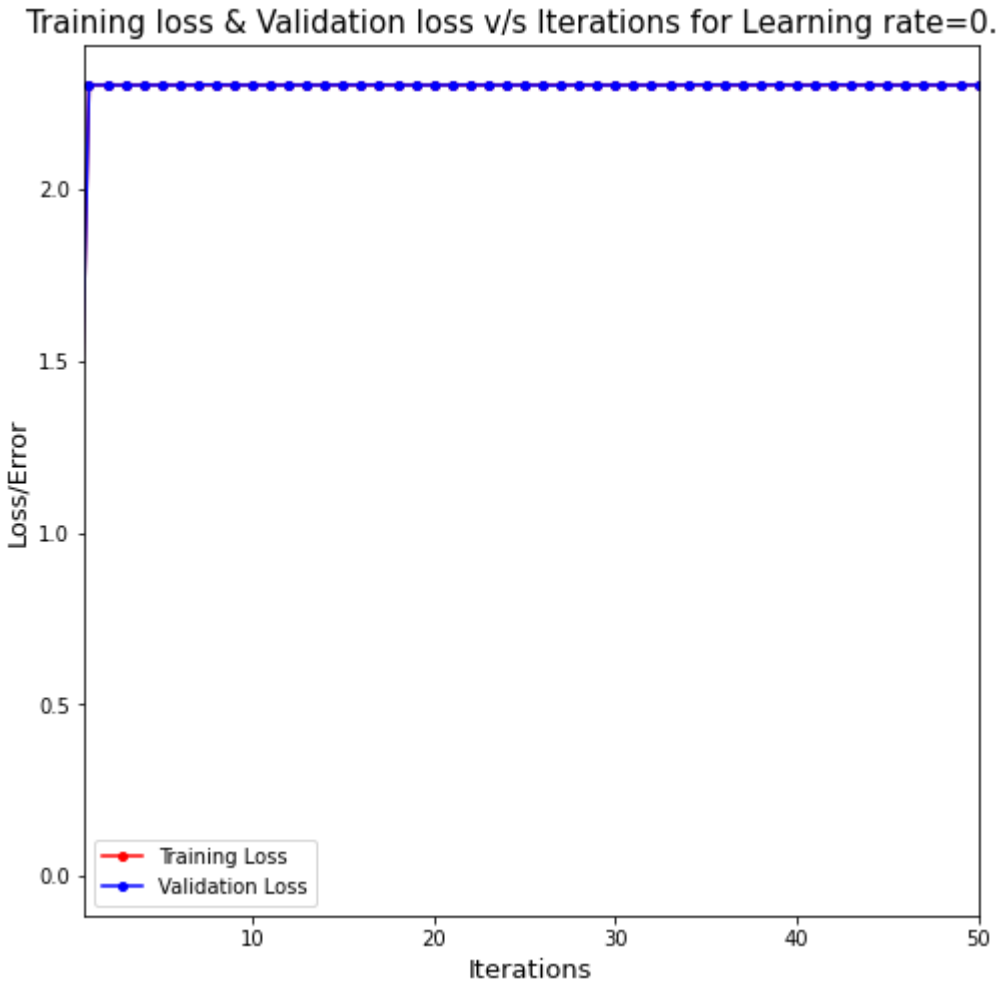
Plot training loss v/s epochs and validation loss v/s epochs for learning rates [0.1, 0.01, 0.001].

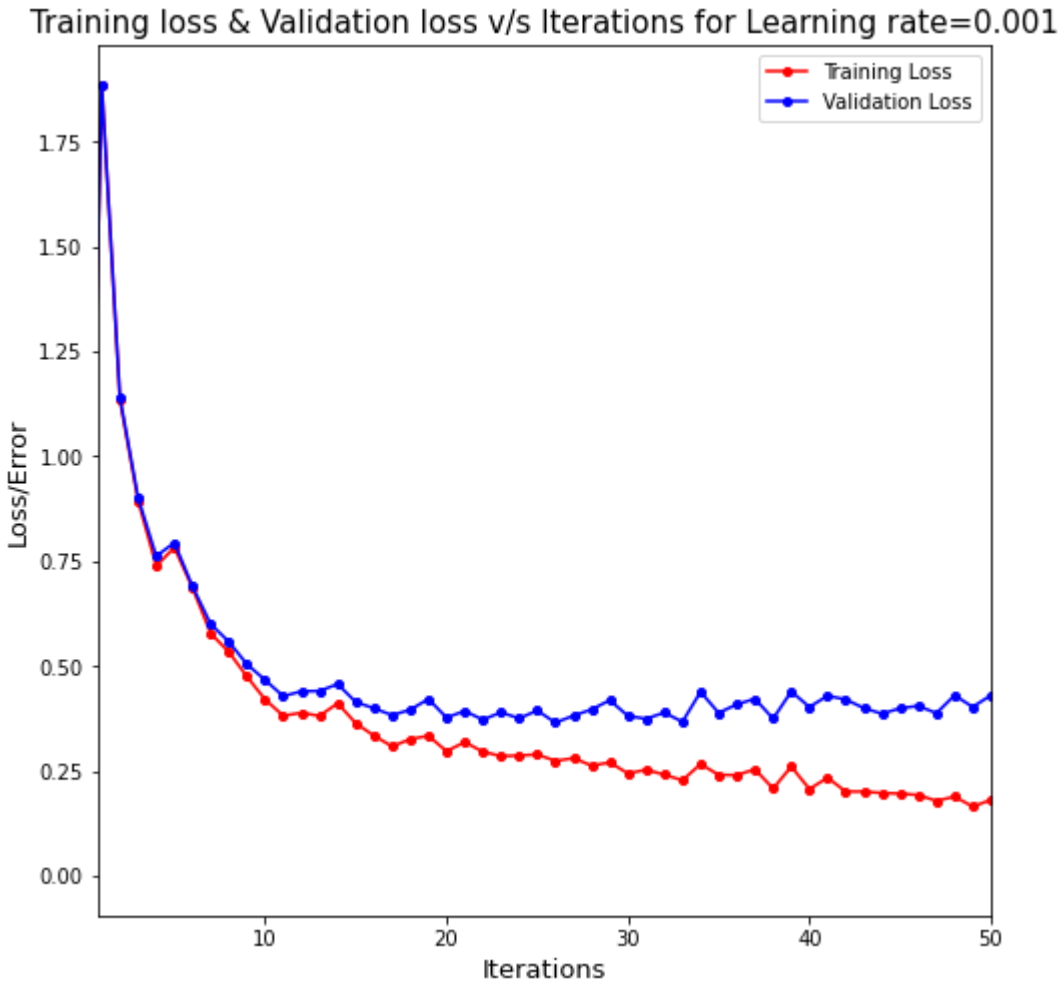
In [ ]:

```
def MLP_learning_rate(lr:int, epochs, batch_size, x_train, y_train, x_valid, y_valid):
    train_loss, validation_loss = [0], [0]

    # Neural network model
    mlp = MLPClassifier(hidden_layer_sizes=(256,32), activation='relu', learning_rate_init=lr,
                        batch_size=batch_size)

    # Run epochs
    for epoch in range(epochs):
        # Train the Neural network model using partial_fit()
        nsamples, nrows, ncols = x_train.shape
        mlp.partial_fit(x_train.reshape((nsamples, nrows*ncols)), y_train, classes=np.arange(10))

        # Compute Training Loss
        nsamples, nrows, ncols = x_train.shape
        y_true = np.eye(10)[y_train]
        y_pred_proba = mlp.predict_proba(x_train.reshape((nsamples, nrows*ncols)))
        train_loss.append(log_loss(y_true, y_pred_proba))

        # Compute Validation Loss
        nsamples, nrows, ncols = x_valid.shape
        y_true = np.eye(10)[y_valid]
        y_pred_proba = mlp.predict_proba(x_valid.reshape((nsamples, nrows*ncols)))
        validation_loss.append(log_loss(y_true, y_pred_proba))

    # Plot training & Validation loss
```

```
    plt.figure(figsize=(8,8))
    plt.plot(train_loss, 'r-o', label='Training Loss', ms=4)
    plt.plot(validation_loss, 'b-o', label='Validation Loss', ms=4)
    plt.xlabel('Iterations', fontsize=13)
    plt.ylabel('Loss/Error', fontsize=13)
    plt.xlim((0.75,epochs))
    plt.title(f"Training loss & Validation loss v/s Iterations for Learning rate={lr}", fontsize=15)
    plt.legend()
    plt.show()
    return mlp

# Batch size as 128 & epochs as 50
epochs = 50
batch_size = 128
models_lr = []

# Try all Neural networks with all the learning rates
for lr in [0.1, 0.01, 0.001]:
    mlp = MLP_learning_rate(lr, epochs, batch_size, train_image, train_label, valid_image, valid_label)
    models_lr.append(mlp)
```




Training loss & Validation loss v/s Iterations for Learning rate=0.01
Loss/Error
Iterations
Training Loss
Validation Loss
0.0
0.5
1.0
1.5
2.0
10
20
30
40
50



Accuracy on Neural Networks (MLP) with learning rates [0.1, 0.01, 0.001].

In [ ]:

```
lr = [0.1, 0.01, 0.001]

for i in range(len(lr)):
    print(f"==> Accuracy with Learning Rate='{lr[i]}'")
    nsamples, nrows, ncols = train_image.shape
    print(f"Training Accuracy:", models_lr[i].score(train_image.reshape((nsamples, nrows*ncols)),
                                                    train_label))
    nsamples, nrows, ncols = valid_image.shape
    print(f"Validation Accuracy:", models_lr[i].score(valid_image.reshape((nsamples, nrows*ncols)),
                                                    valid_label))
    print()
```

```
==> Accuracy with Learning Rate='0.1'
Training Accuracy: 0.09937254901960785
Validation Accuracy: 0.10355555555555555

==> Accuracy with Learning Rate='0.01'
Training Accuracy: 0.10035294117647059
Validation Accuracy: 0.098

==> Accuracy with Learning Rate='0.001'
Training Accuracy: 0.938078431372549
Validation Accuracy: 0.8895555555555555
```

Observation: Best Learning rate is 0.001

1) For large Learning rate 0.1 and 0.01:

When the learning rate is set to high value like "0.1" and "0.01", the neural network is not able to learn the correct pattern in the data. The weights will be updated faster due to large learning rates.
This means weights will be taking huge steps downhill and likely to skip the optimal minima point. This will make the learning process unstable.

Further, it may also happens that the with the huge learning rate, models weights may undergo gradient explosion. In the gradient explosion, the model weights quickly become very large and likey be NAN while training; this happens beacuse of huge updates in weights as learning rate is high.

This insight can also be gained from the Training loss and Validation loss plots, as the both the loss are not able to converge to a minimum value.
Beacuse of these reasons, the model becomes unstable and uncapable to learn from the training data.

2) For small Learning rate 0.001:

The weights will be updated slowly due to small learning rate. This means weights will be taking small steps downhill and likely to converge and reach the optimal minima point. This will make the learning process quite stable.
Also, with the small learning rate, weights won't be updated faster and hence the problem of gradient explosion is avoided.

This insight can also be gained from the Training loss and Validation loss plots, as the both the loss had been converged to a minimum value. Hence, the model becomes stable and capable to learn from the training data.

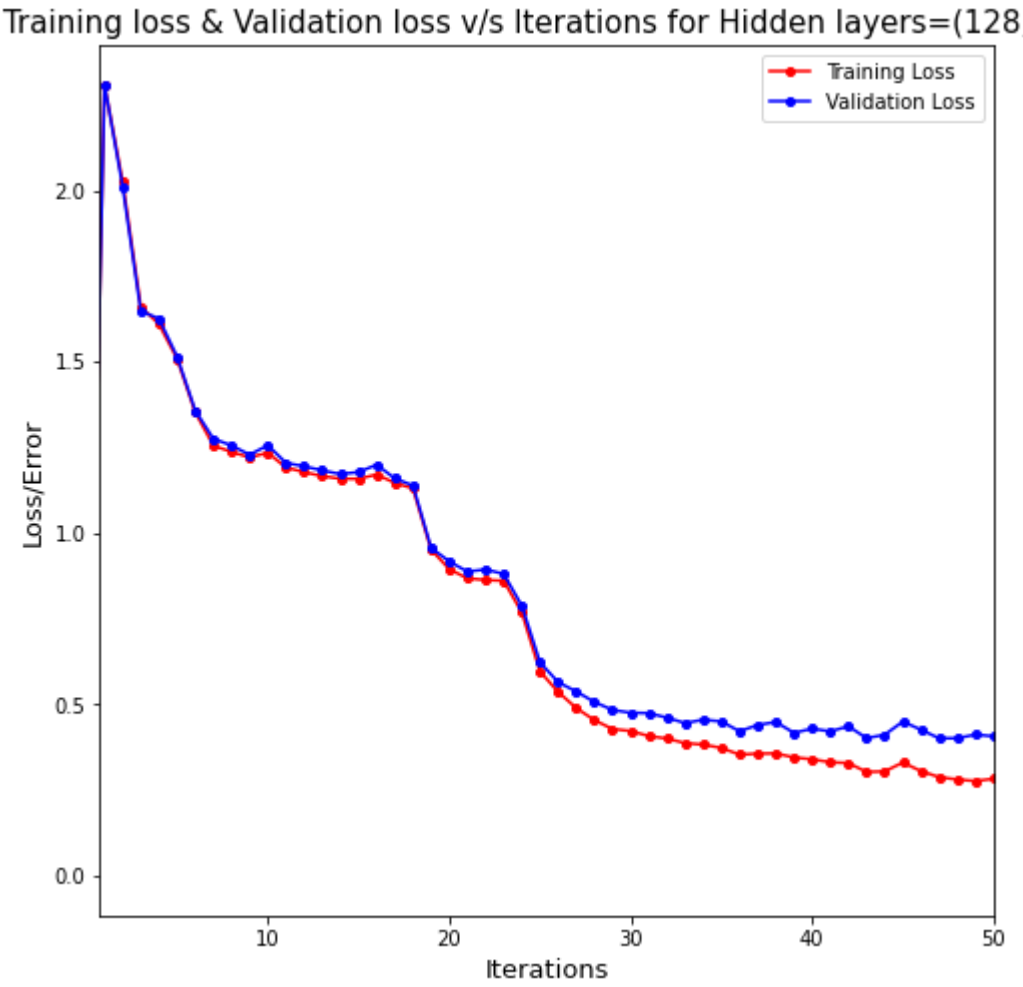
Part (c)

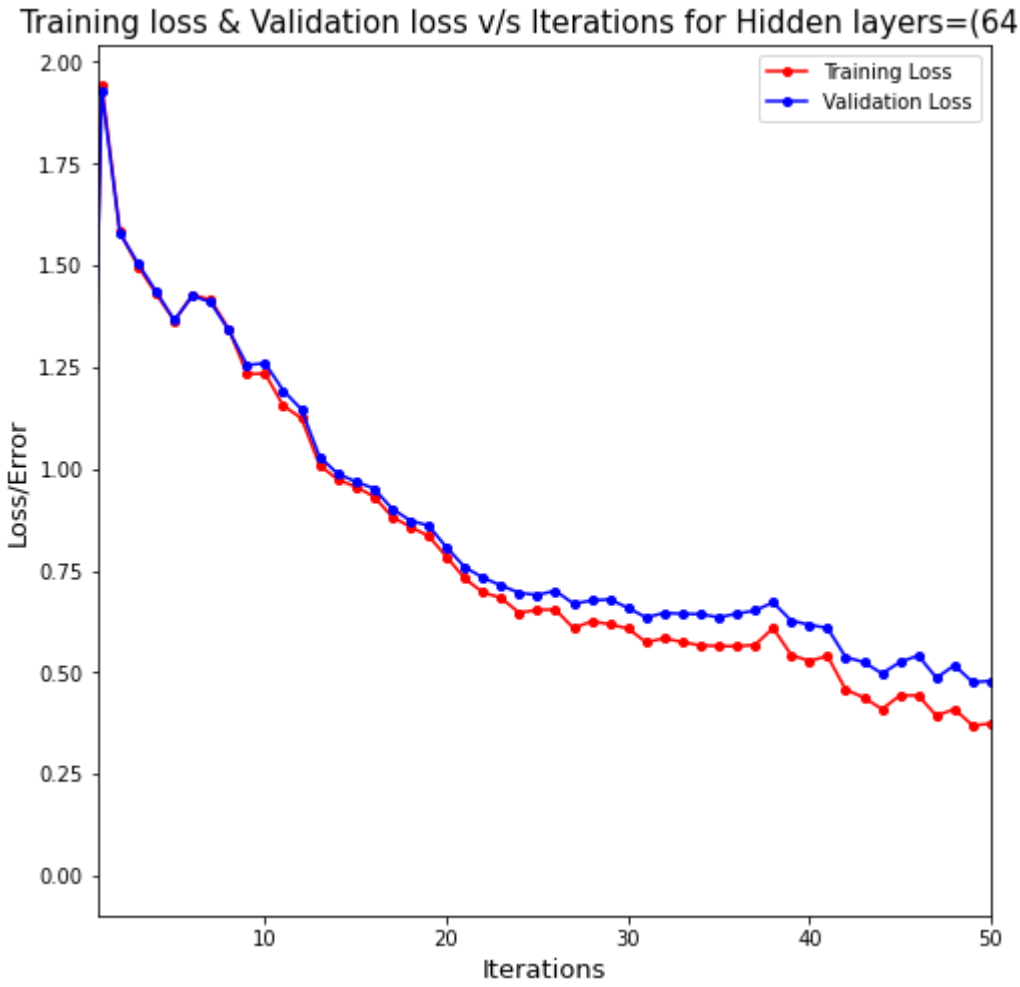
Decrease the number of neurons in each layer to various values. And plot training loss v/s epochs.

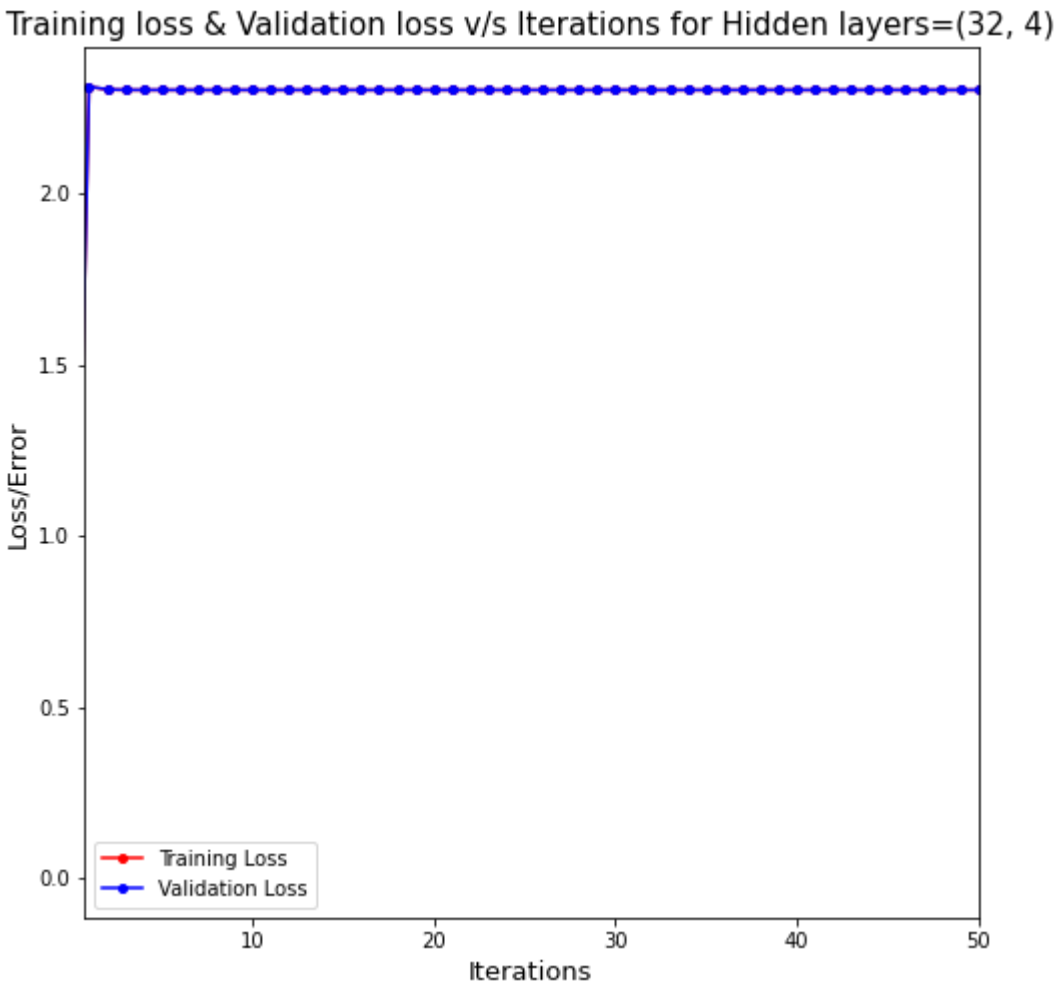
In [ ]:

```
def MLP_hidden_layers(hidden_layer, epochs, batch_size, x_train, y_train, x_valid, y_valid):
    train_loss, validation_loss = [0], [0]

    # Neural network model
    mlp = MLPClassifier(hidden_layer_sizes=hidden_layer, activation='relu', learning_rate_init=0.001,
                        batch_size=batch_size)

    # Run epochs
    for epoch in range(epochs):
```

```
        nsamples, nrows, ncols = x_train.shape
        mlp.partial_fit(x_train.reshape((nsamples, nrows*ncols)), y_train, classes=np.arange(10))

        # Compute Training Loss
        nsamples, nrows, ncols = x_train.shape
        y_true = np.eye(10)[y_train]
        y_pred_proba = mlp.predict_proba(x_train.reshape((nsamples, nrows*ncols)))
        train_loss.append(log_loss(y_true, y_pred_proba))

        # Compute Validation Loss
        nsamples, nrows, ncols = x_valid.shape
        y_true = np.eye(10)[y_valid]
        y_pred_proba = mlp.predict_proba(x_valid.reshape((nsamples, nrows*ncols)))
        validation_loss.append(log_loss(y_true, y_pred_proba))

    # Plot training & Validation loss
    plt.figure(figsize=(8,8))
    plt.plot(train_loss, 'r-o', label='Training Loss', ms=4)
    plt.plot(validation_loss, 'b-o', label='Validation Loss', ms=4)
    plt.xlabel('Iterations', fontsize=13)
    plt.ylabel('Loss/Error', fontsize=13)
    plt.xlim((0.75,epochs))
    plt.title(f'Training loss & Validation loss v/s Iterations for Hidden layers={hidden_layer}',
              fontsize=15)
    plt.legend()
    plt.show()
    return mlp

# Batch size as 128 & epochs as 50
epochs = 50
batch_size = 128
models_layer = []

# Try all Neural networks with decreasing neurons in Hidden layers ((128,16), (64,8), (32,4))
for hidden_layer in ((128,16), (64,8), (32,4)):
    mlp = MLP_hidden_layers(hidden_layer, epochs, batch_size, train_image, train_label, valid_image,
                            valid_label)
    models_layer.append(mlp)
```


Training loss & Validation loss v/s Iterations for Hidden layers=(128, 16)
Training Loss
Validation Loss
Loss/Error
Iterations
0.0
0.5
1.0
1.5
2.0
10
20
30
40
50


Training loss & Validation loss v/s Iterations for Hidden layers=(64, 8)
Training Loss
Validation Loss
Loss/Error
Iterations
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
10
20
30
40
50



Accuracy on Neural Networks (MLP) with Hidden layers ((128,16), (64,8), (32,4)).

In [ ]:

```
hidden_layer = ((128,16), (64,8), (32,4))

for i in range(len(hidden_layer)):
    print(f"==> Accuracy with hidden layer='{hidden_layer[i]}'")
    nsamples, nrows, ncols = train_image.shape
    print(f"Training Accuracy:", models_layer[i].score(train_image.reshape((nsamples, nrows*ncols)),
                                                       train_label))
    nsamples, nrows, ncols = valid_image.shape
    print(f"Validation Accuracy:", models_layer[i].score(valid_image.reshape((nsamples, nrows*ncols)),
                                                         valid_label))
    print()
```

```
==> Accuracy with hidden layer='(128, 16)'
Training Accuracy: 0.8987058823529411
Validation Accuracy: 0.8645555555555555

==> Accuracy with hidden layer='(64, 8)'
Training Accuracy: 0.873686274509804
Validation Accuracy: 0.8537777777777777

==> Accuracy with hidden layer='(32, 4)'
Training Accuracy: 0.10041176470588235
Validation Accuracy: 0.09766666666666667
```

Observation & Justification for Decreasing Hidden layers

When the no. of neurons in each hidden layers are decreased, then the complexity of the neural network also decreases.

When we use too few neurons in the hidden layers, underfitting of the models occurs as the complexity of the model decreases. The neural network model becomes unable to learn from the training data. This is exactly same as we observe when the hidden layers size is (32,4), the model is unable to learn well from the Training data and the loss hasn't converged to a local minima. Also, it shows the poor accuracy of 10%. This problem is due to underfitting of model due to low complexity of neural network.

When the neurons in the hidden layers are reasonable enough to learn from the training data, then there is enough complexity of the model to learn from the data. But as the neurons in the hidden layer keeps on decreasing, then basically the complexity of the neural network model decreases, hence the Bias (accuracy of the match) will also decrease. This is exactly what we observe for Hidden layer (128,16) and (64,8). The neural network with hidden layer (128,16) will have a higher complexity than the network with hidden layer (64,8); hence its performance on Training set & Validation set (accuracy) is higher than the network with hidden layer (64,8).

> So, we conclude that on decreasing the no. of neurons in hidden layers, decreases the complexity of the network & increases the Bias in the neural network model.

Part (d)

Perform grid search on appropriate parameters of MLPClassifier.

In [ ]:
```
from sklearn.model_selection import GridSearchCV

# Grid parameter for grid search
grid_parameters = {
    'hidden_layer_sizes': [(256,32), (128,16), (64,8), (32,4)],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['sgd', 'adam'],
    'max_iter' : [15,25,50]
}

# Classifiers for MLP and GridSearch
mlp = MLPClassifier()
clf = GridSearchCV(mlp, grid_parameters, n_jobs=-1, cv=3)

# Run Grid search
nsamples, nrows, ncols = train_image.shape
clf.fit(train_image.reshape((nsamples, nrows*ncols)), train_label)
```

Save the GridSearch classifer

In [ ]:
```
import pickle
with open('Weights/gridsearch.pickle', 'wb') as pickle_in:
    pickle.dump(clf, pickle_in)
```

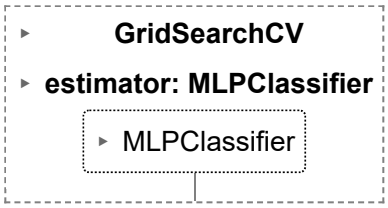
Load the GridSearch classifer

In [ ]:
```
import pickle
with open('Weights/gridsearch.pickle', 'rb') as pickle_in:
    clf_grid = pickle.load(pickle_in)
    print(clf_grid)
```

```
GridSearchCV(cv=3, estimator=MLPClassifier(), n_jobs=-1,
             param_grid={'activation': ['identity', 'logistic', 'tanh', 'relu'],
                         'hidden_layer_sizes': [(256, 32), (128, 16), (64, 8),
                                                (32, 4)],
                         'max_iter': [15, 25, 50], 'solver': ['sgd', 'adam']})
```

In [ ]:
```
clf
```

Out[ ]:



Best parameter values based on GridSearch

In [ ]:
```
print('Best parameters found:\n')
print(clf.best_params_)
```

```
Best parameters found:

{'activation': 'relu', 'hidden_layer_sizes': (256, 32), 'max_iter': 50, 'solver': 'adam'}
```

Best accuracy value recorded while running GridSearch

In [ ]:
```
print('Best accuracy recorded is: ', end=' ')
print(clf.best_score_)
```

```
Best accuracy recorded is:  0.8653333333333334
```

Analysis of parameters

Grid-search is basically used to find the optimal hyperparameters for a model which results in the most 'accurate' predictions.

1) Reason for activation = ReLU

```
ReLU is observed to be the Best activation function. This is beacuse
RelU doesn't activates all the      neurons at the same time.
When the weighted sum of inputs to a neuron in any layer is negative,
then ReLU outputs zero (0). Thus, making that neuron's contribution in
the weighted sum of input to neurons of next layer to be zero. It
activates the right neurons at the right time during Training process.

Also, ReLU has high value of derivative/gradient (1 for x > 0), this
```

helps to avoid gradient vanishing. This is neccessary for weights updation because weights are updated using gradients of each layers. So, weights will be updated by large value, unlike logistic or tanh function where due to low gradient weights are updated slowly. Due to this ReLU networks learns faster, hence shows the best performance. Its loss curve declines very sharply than any other activation function. Hence, it shows fast convergence and best performance.

So, the best activation would be ReLU.

2) Reason for hidden_layer_sizes = (256, 32)

When the neurons in the hidden layers are reasonable enough to learn from the training data, then there is enough complexity of the model to learn from the data.
But as the neurons in the hidden layer keeps on decreasing, then basically the complexity of the neural network model decreases, hence the Bias (accuracy of the match) will also decrease.
This is exactly what we observe for all the Hidden layer. The neural network with hidden layer (128,16) will have a higher complexity than the network with hidden layer (64,8) and (32,4); hence its performance (accuracy metric) will be higher than the network with hidden layer (64,8) & (32,4).

So, the best hidden_layer size would be (256,32).

3) Reason for max_iter/epoches = 50

Epoches that we run during training process are important. More we run the epoches during training, more the loss function will decrease and more it will converge. Hence the performance metrics (accuracy, etc) will improve (increase) on the Training and Validation Dataset, as the loss will converge with greater epoches.

So, the best epochs/max_iter would be 50.

4) Reason for solver = adam

Adam is an optimization algorithm (just like SGD or Mini-Batch GD) that provides more efficient neural network training mechanism than SGD. It is the most effective optimization technique in large data sets.
Adam optimization method is an extension of SGD for solving non-convex problems much faster using fewer resources. Also, adam is much faster than the SGD.

So, the best optimization solver technique will be adam.