Machine Learning
Assignment - 3

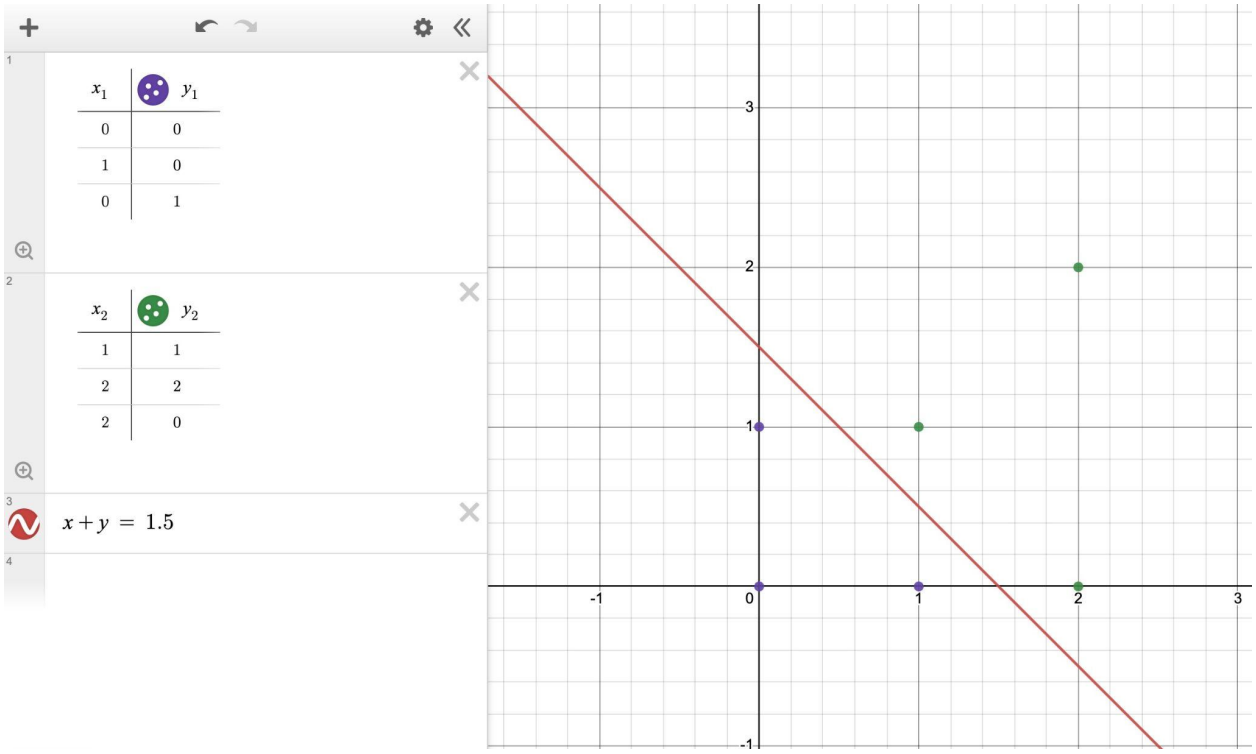
Ashwin Sheoran
2020288

Section-A

Ans 1)

a)

Yes, the points are linearly separable.



Here, for example, we can see that line x+y = 1.5 separates the points

b)
The maximum margin hyperplane is the one between the 2 closest points and maximum distance from them, here this would be the line that will pass through the point (0.5 , 1) and (1.5 , 0) as they are at equal distances from closest point and the line separates the 2 types of points.
The line is $2x+2y = 3$ and the weights will be [1 , 1] (column matrix) as the plane is at equal distance from form the closest points.

c)
Here if we remove the points (0,0) or (2 , 2) then the margin will remain same.
If we remove any other point then margin will increase.
Here we are given points such that margin will increase or remain same depending upon which points we are removing.

d)
Generally when we remove any of the support vectors, the optimal margin increases. The optimal margin depends on all the support vectors and removing them generally changes the margin. Although the margin can increase or remain the same but never decrease when we remove support vectors.

Section-B

Ans 1)

```
class NeuralNetwork:
    def __init__(self , N , hidden_layer_size , lr, A_function ,max_iter , Weight_fun , batch_size ) :
        self.N = N
        self.hidden_layer_size = hidden_layer_size
        self.lr = lr ## learning rate
        self.A_function = A_function
        self.batch_size = batch_size
        self.max_iter = max_iter ## number of epochs
        self.Weight_fun = Weight_fun  # Weight initialization function
        self.layers = []



# ##### Helper functions  for part A
    def forward(self , X):   ## . helper function for going into next layer
        W1 = self.random_init_function(self.hidden_layer_size[0] , self.hidden_layer_size[1])
        ## Weight matrix using function
        A_prev = X   ## initial value at the previous layer

        Z = np.dot(A_prev, W1 )     ## Multiply with weights to get probability

        if(self.A_function == "relu"):
            A = self.relu(Z)

        elif (self.A_function == "tanh"):
            A = self.tanh(Z)

        elif (self.A_function == "linear"):
            A = self.linear(Z)

        elif (self.A_function == "sigmoid"):
            A = self.sigmoid(Z)

        return A

    def loss_function(self, A, y):  ## Helper function
        log_val = - np.log(A[np.arange(len(y)), y.argmax(axis=1)])
        loss = np.sum(log_val)/ len(y)
        return loss
```

Implemented the class __init__ and the helper functions for the fit , predict , predict_proba and score functions

```
## Part A
#
##
    def fit(self , X , Y):
        train_loss = [] ## array for storing the train loss
        for epoch in self.max_iter:
            train_batch_loss = []

            for batch in self.batch_size:
                A = self.forward(X) ## Moving to next nueron
                curr_loss = self.loss_function(A , Y[batch]) ## Calculating loss using the helper function
                train_batch_loss.append(curr_loss)

            train_loss.append( train_batch_loss)
        return train_loss



    def predict_proba(self , X):
        y_proba = []

        output = X;
        for layer in self.layers:
            output = self.forward(output) ## Going forward through all the layers
            y_proba.append(output)
        y_proba = self.softmax(y_proba)     ## using softmax as this is the last layer
        return np.array(y_proba) ## class wise probability


    def predict(self , X):
        y_pred = []
        y_proba = self.predict_proba(X)
        for i in range(len(y_proba)):
            y_pred.append(np.argmax(y_proba[i])) ##Predicting the y by passing the X_train into the predict_proba funct
        return np.array(y_pred)                     ## and getting the y pred

    def score(self , X , Y):
        y_pred = self.predict(X) ## predicting the y
        return np.mean(y_pred== Y)   ## returning he score
```

Implementing the fit, predict_proba, predict and score functions

Ans 2)

```
### B Part

    ## Here gradients are partial derivatives of the functions

    def relu(self, X):
        return X * (X>=0) ### return max(0,X)

    def relu_grad(self, X):
        return 1*(X>=0)     ## return 1 if x>0 else 0

    def leaky_relu(self, X):
        return np.maximum(0.1 * X, X)      ## return 0.01 * X if x<0 else X

    def leaky_relu_grad(self, X):
        if X>0 :
            return X
        else :
            return 0.01*X        ## return 0.01 if x<0 else X

    def sigmoid(self, X):
        return 1/(1+np.exp(-X)) ## returning this value using the definition of sigmoid function

    def sigmoid_grad(self, X):
        return self.sigmoid(X) * (1-self.sigmoid (X)) ## We get this result on taking derivative of the sigmoid functi

    def linear(self, X):   ## in linear we return X only
        return X

    def linear_grad(self, X):   ## the matrix with all 1s instead of values of x
        return np.ones(X.shape)

    def tanh(self, X):   # Compute hyperbolic tangent element-wise ( (e^X - e^-X)/(e^X + e^-X) )
        return np.tanh(X)

    def tanh_grad(self, X):
        return 1 - self.tanh(X)*self.tanh(X)  ## derivative of tanh(x) is 1 - tanh2(x)

    def softmax(self, X):
        exp = np.exp(X)
        return exp/(np.sum(exp))        ## softmax is a normalized exponential function

    def softmax_grad(self, X):
        result_matrix = np.diag(X) ## constructing jacobian matrix of softmax as it is the maxtrix of partial derivati

        for i in range(len(result_matrix)):
            for j in range(len(result_matrix)):
                if i == j:
                    result_matrix[i][j] = (1-X[i])
                else:
                    result_matrix[i][j] = -X[i]*X[j]
        return result_matrix
```

Implemented the activation functions from scratch.

Ans 3)

```
#### Part C

    ## Here shape = (self.hidden_layer_size[0],self.hidden_layer_size[1])

    def zero_init_function(self, shape):
        weight = np.zeros(shape)    ## returns weight with just zero initialization
        return weight

    def random_init_function(self, shape):
        weight = np.random.rand(shape[0], shape[1])     ## return weights with random initialization
        return weight

    def normal_init_function(self, shape):
        weight = np.random.normal(0 , 1 , size = shape , scale = 0.01)
        ##samples from the parameterized normal distribution (Guassian)
        ## Mean 0 , Variance 1
        return weight
```

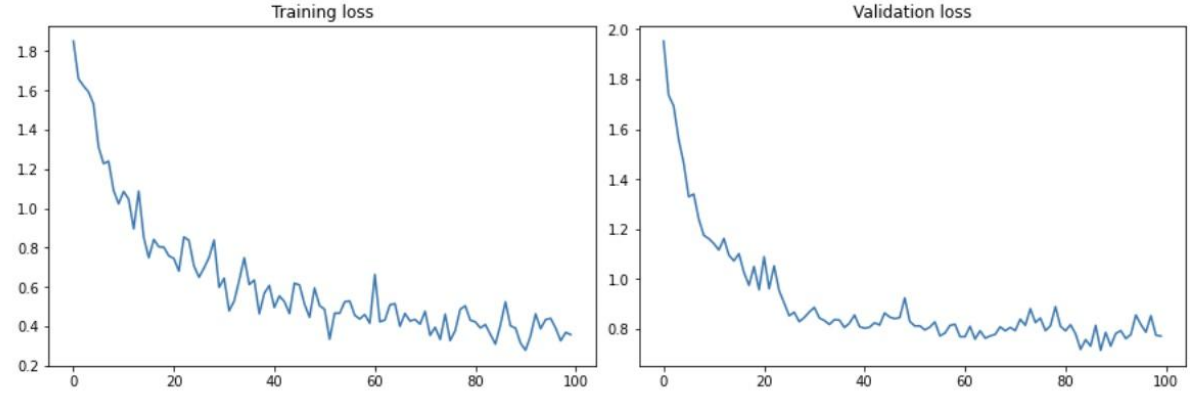
Implemented the Weight Initialization function

Section-C

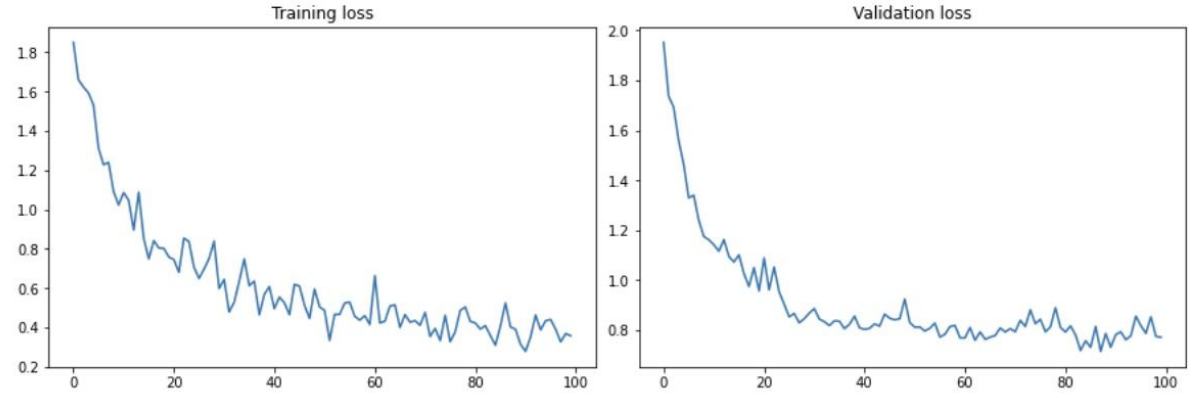
Ans 1)

a)

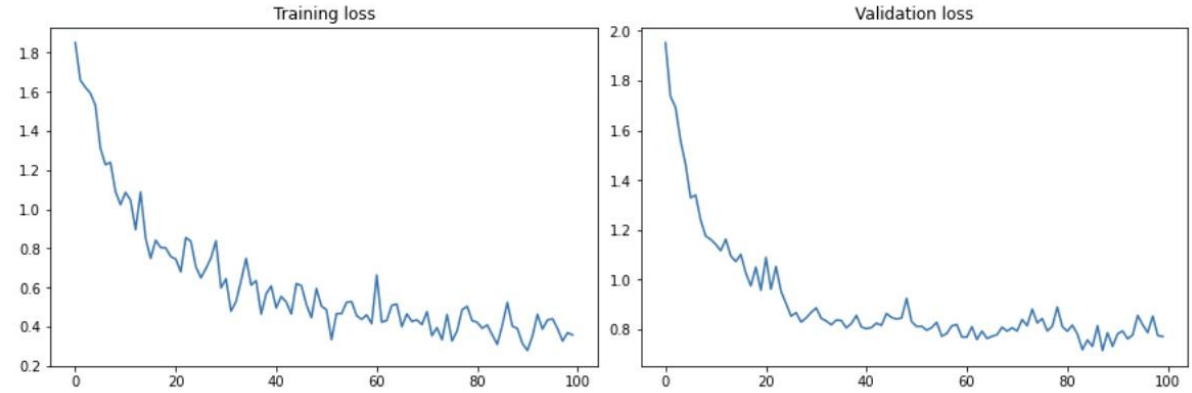
```
For Relu
Training loss is
0.3567434685167139

Validation loss is
0.7718632652836106
```



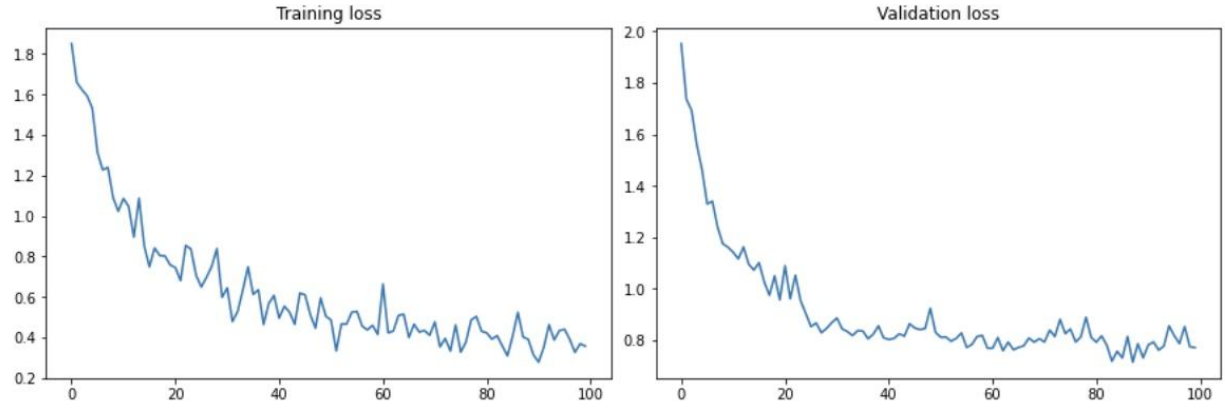

```
For tanh
Training loss is
0.6492397645397525

Validation loss is
0.6807226646093036
```



```
For Sigmoid
Training loss is
0.6314474813424521

Validation loss is
0.57229707947686
```




```
For Linear
Training loss is
1.2227473663112802

Validation loss is
2.5158840704083048
```



We can easily see that the training error is less than the validation error Both error decrease on the increase in the epochs but almost become constant Training and validation errors are both least on Relu Overall, Relu is performing the best among all four.

Code for plotting

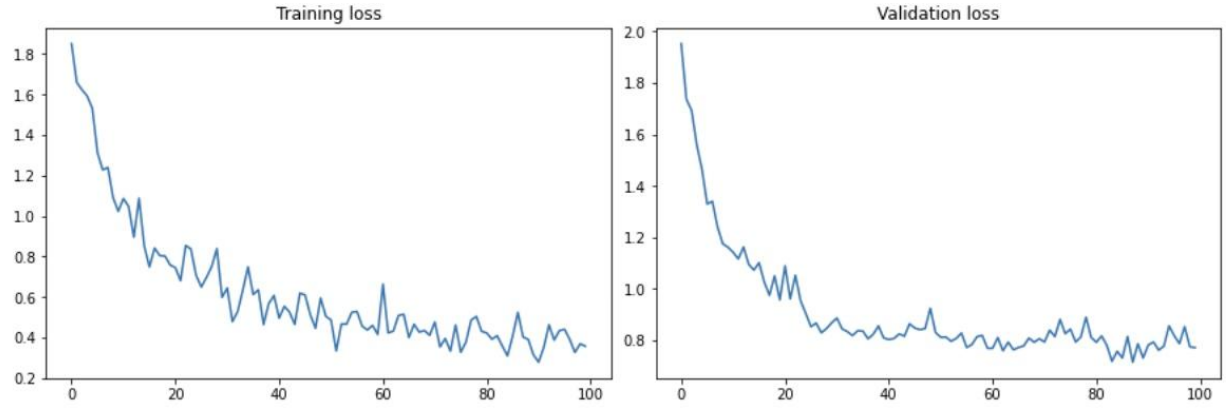
```
def fitter( epochs , batches , mlp , X_train , y_train , X_valid , y_valid ):   ## Function for partial fit
    train_samples = X_train.shape[0]
    features = np.unique(y_train)
    validation_loss =[]
    training_loss = []

    curr_epoch = 0
    while curr_epoch != epochs:
        Shuffled_X = np.random.permutation(X_train.shape[0]) ## Shuffling the dataset
        check_index = 0
        while True:
            i = Shuffled_X[check_index : check_index  + batches]
            mlp.partial_fit(X_train[i], y_train[i], classes=features)  ## partial fitting to get training scores
            check_index += batches                              ## and validation score after each iteration

            if check_index >= train_samples:            ## checking if partial fit done over all samples
                break

        validation_loss.append(log_loss(y_valid, mlp.predict_proba(X_valid)))
        training_loss.append( mlp.loss_)
        curr_epoch += 1

    # training_loss = mlp.loss_curve_
    print("Training loss is ")
    print( training_loss[len(training_loss)-1])
    # plt.plot(training_loss[20:])

    print("")
    print("Validation loss is ")
    print( validation_loss[len(validation_loss)-1])
    # plt.plot(validation_loss[20:])
    # plt.plot(training_loss)


    return training_loss , validation_loss
```

```
def plotter(training_loss , validation_loss , epochs=100):
    fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True, figsize=(12, 4))
    rng = np.arange(0, epochs)  #Cost function graph

    ax1.plot(rng, training_loss)
    ax2.plot(rng , validation_loss )

    ax1.set_title("Training loss");  ax2.set_title("Validation loss")
```

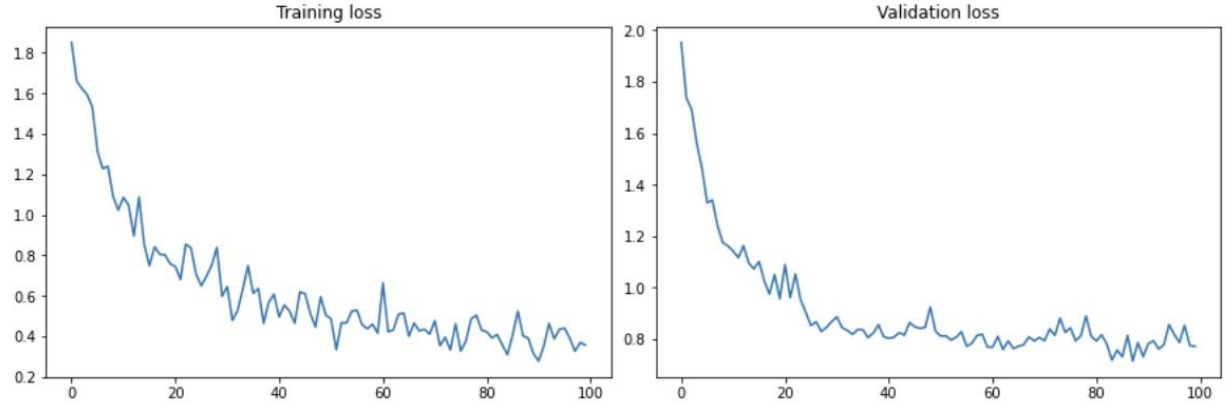
```
print("For Relu")
mlp_relu = MLPClassifier( hidden_layer_sizes=(256, 32),  max_iter=100 , batch_size = 200 , activation = 'relu' )
training_loss , validation_loss = fitter(100 , 200 , mlp_relu , X_train , y_train , X_valid , y_valid)
plotter(training_loss , validation_loss)
```

B)

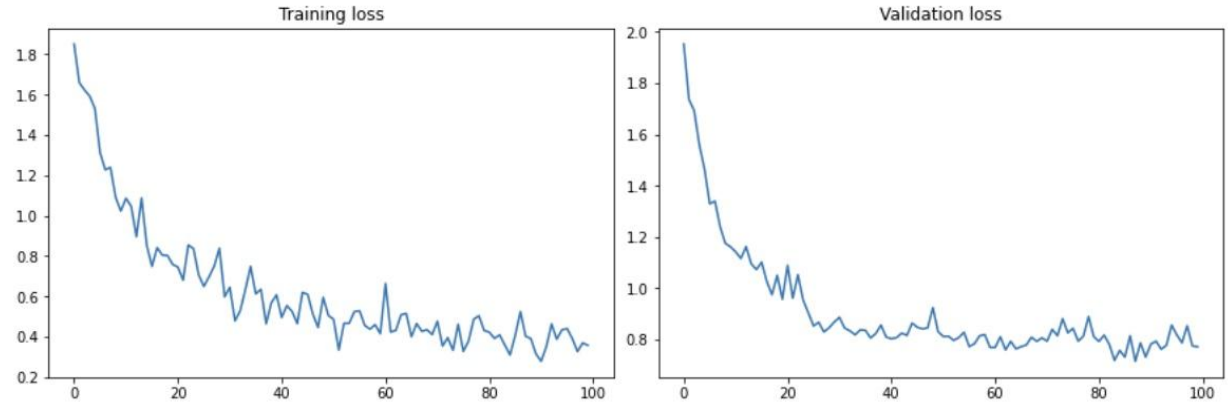
```
For Relu learning rate  = 0.1
Training loss is
2.3299924211800254

Validation loss is
2.3075881401266782
```




```
For Relu learning rate  = 0.01
Training loss is
2.3034475147436226

Validation loss is
2.3028027190449527
```



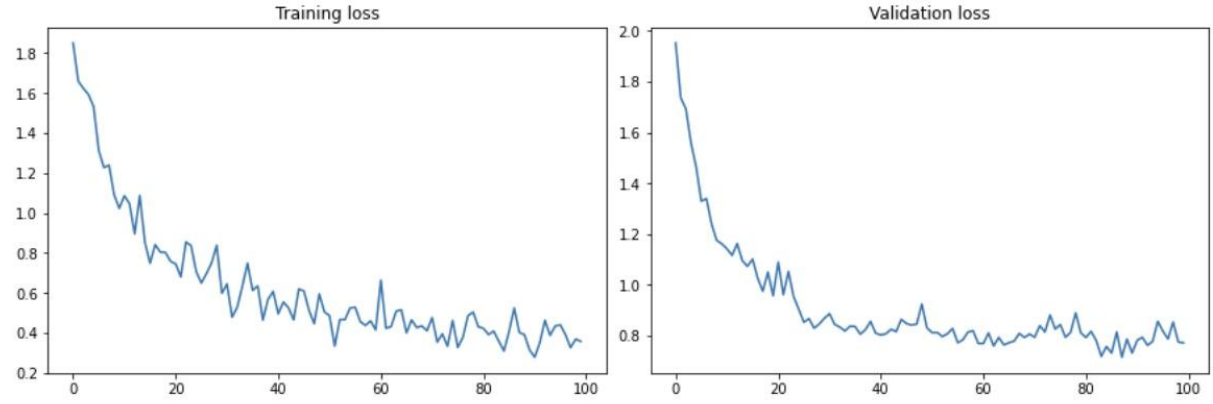
```
For Relu learning rate  = 0.001
Training loss is
0.25428129366570756

Validation loss is
0.8403054104521971
```



We are getting the best result with a learning rate = 0.001, as in other cases, we are underfitting.

C)

```
For Relu with less neurons
Training loss is
0.7952417314228731

Validation loss is
0.960231474638995
```



Decreaseing the number of neurons per layers increases the loss

On changing the hidden layer parameter to (220, 20), the loss increased.
So when the number of neurons decreases, the loss increases.
Training loss increased from 0.35 to 0.79
Validation loss increased from 0.77 to 0.96

D)

```
mlp_gs = MLPClassifier(max_iter=400 )
parameter_space = {
    'hidden_layer_sizes': [(256,32) , (220, 20)],
    'activation': ['relu' ] ,
    'learning_rate_init':[0.001 , 0.005 , 0.05]

}
clf = GridSearchCV(mlp_gs, parameter_space, n_jobs=-1, cv=5 , verbose=2)
clf.fit(X_train, y_train) # X is train samples and y is the corresponding labels
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.005; total time=  16.7s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.005; total time=  22.3s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.005; total time=  28.8s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.005; total time=  25.5s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.001; total time= 1.0min
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.05; total time=  16.9s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.05; total time=  15.3s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.05; total time=  14.3s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.05; total time=  22.5s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.05; total time=  21.7s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.001; total time= 1.5min
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.005; total time= 1.7min
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.001; total time=  58.6s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.001; total time= 2.5min
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.005; total time=  14.7s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.001; total time= 2.7min
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.005; total time=  15.2s
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.005; total time=  16.3s
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.005; total time=  30.9s
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.005; total time=  13.7s
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.05; total time=  11.5s
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.05; total time=   9.3s
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.05; total time=   9.9s
[CV] END activation=relu, hidden_layer_sizes=(256, 32), learning_rate_init=0.001; total time= 3.2min
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.05; total time=   7.2s
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.05; total time=   7.4s
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.001; total time= 2.0min
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.001; total time= 2.0min
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.001; total time= 2.8min
[CV] END activation=relu, hidden_layer_sizes=(220, 20), learning_rate_init=0.001; total time= 3.0min

GridSearchCV(cv=5, estimator=MLPClassifier(max_iter=400), n_jobs=-1,
             param_grid={'activation': ['relu'],
                         'hidden_layer_sizes': [(256, 32), (220, 20)],
                         'learning_rate_init': [0.001, 0.005, 0.05]},
             verbose=2)
```

```
print(clf.best_params_)
```

```
{'activation': 'relu', 'hidden_layer_sizes': (256, 32), 'learning_rate_init': 0.001}
```

On applying grid search, we find that with relu, the best parameters are hidden layer size (256, 32) and learning rate = 0.001.
The learning rate can't be 0.005 as then we will overfit.
If it is 0.05, then we will underfit.
If the number of neurons decreases then the loss increases, so (256, 32) is a better parameter of hidden_layer than (220, 20)
And we already know that Relu is better than other activation functions.

So our best model is Relu with hidden layer size (256, 32) and learning rate = 0.001