DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Homework Assignment No. 11:

HW No. 11: Multilayer Perceptrons

submitted to:

Professor Joseph Picone
ECE 8527: Introduction to Pattern Recognition and Machine Learning
Temple University
College of Engineering
1947 North 12th Street
Philadelphia, Pennsylvania 19122

April 7th, 2023

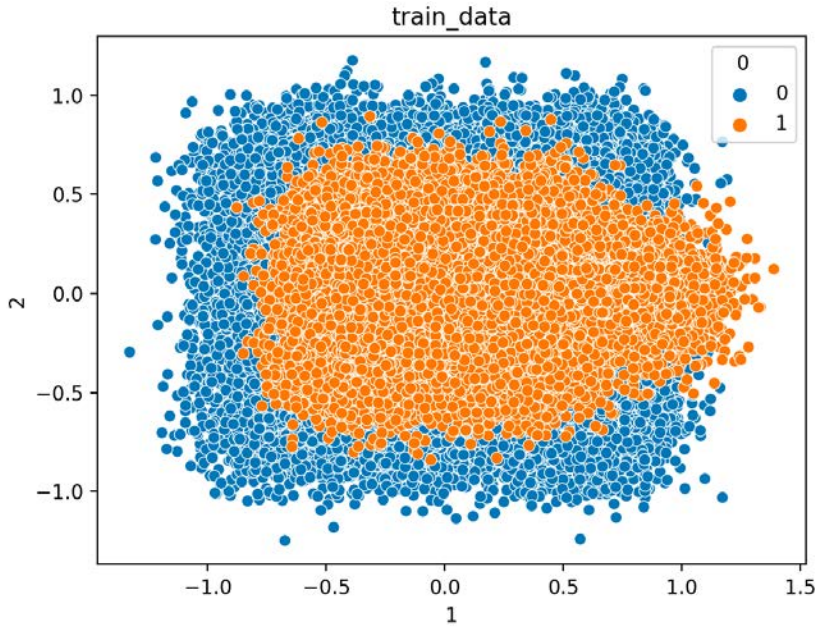
prepared by:

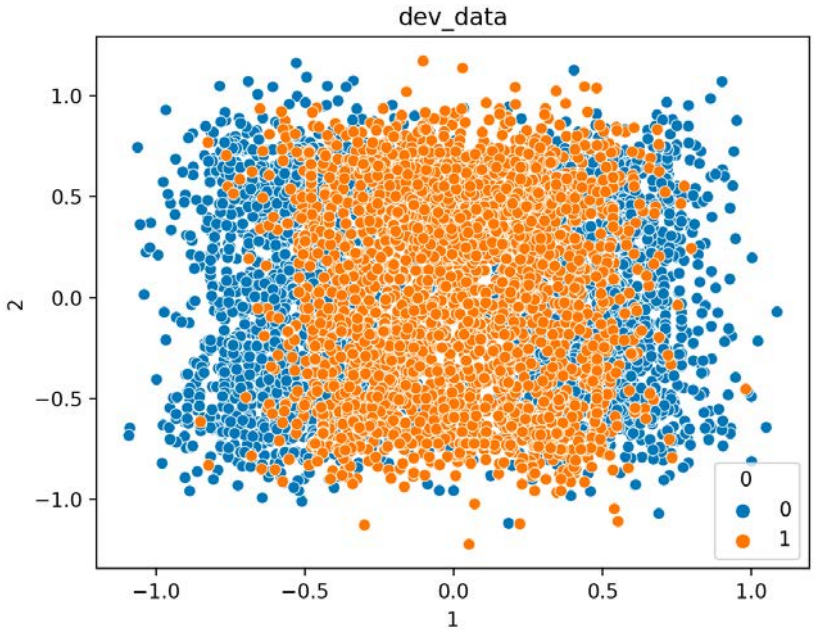
Gavin Koma
Email: gavintkoma@temple.edu

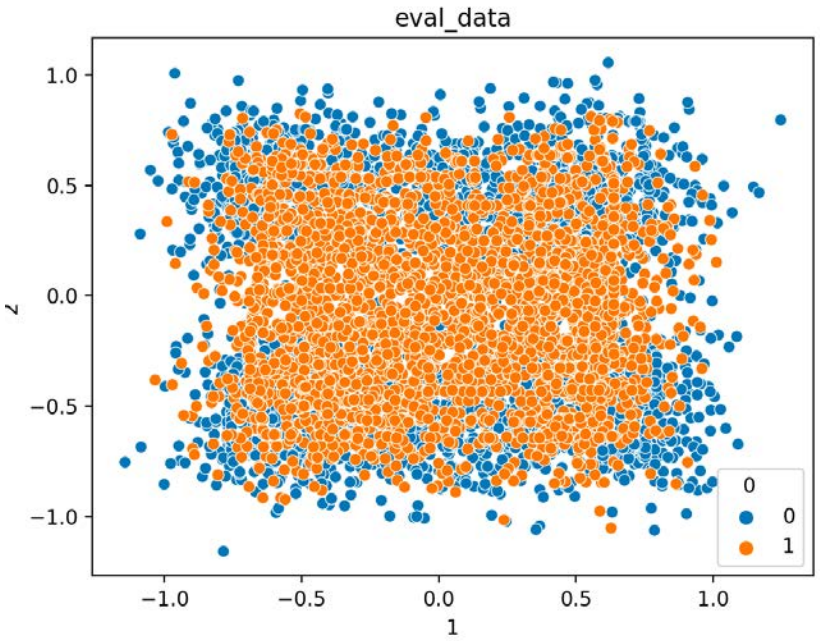
A. TASK 1

The main Task of this assignment requires us to use Dataset 10 and to implement either a single multilayer perceptron OR an equivalent neural network and compare the performance to the previous analyses that we have performed on Dataset 10.

As always, I want to plot the data for this set just to ensure that no degree of corruption has occurred while previously working with it.







These graphs look good, and I have omitted the code to plot them due to simplicity. I utilized seaborn's scatterplot in order to plot these and specified hue in the label column. The next goal is to optimize and prep our data to be passed to pytorch's functions.

We first arrange the data in the proper x & y train/dev/eval and transform the values into FloatTensors. A FloatTensor is a data structure that represents a multi-dimensional array of floating-point numbers. FloatTensor data is used for representing numerical data that is commonly used in training and evaluating ML models. In a forum, I was also able to read that this type of data is preferred over other data types because floating-point arithmetic allows for notably higher accuracy and precision.

```
#%%prep the datasets for training
x_train = train_data.iloc[:,1:3]
y_train = train_data.iloc[:,0]
x_dev = dev_data.iloc[:,1:3]
y_dev = dev_data.iloc[:,0]
x_eval = eval_data.iloc[:,1:3]
y_eval = eval_data.iloc[:,0]

x_train = torch.Tensor(x_train.values) #convert to an array
x_dev = torch.Tensor(x_dev.values)
x_eval = torch.Tensor(x_eval.values) #array

x_train = torch.FloatTensor(x_train)
y_train = torch.FloatTensor(y_train)
x_dev = torch.FloatTensor(x_dev)
y_dev = torch.FloatTensor(y_dev)
x_eval = torch.FloatTensor(x_eval)
y_eval = torch.FloatTensor(y_eval)
```

Prior to building our model, we need to include a method for performing gradient descent which will allow us to optimize our neural network. As seen below, we set requires_grad = True in order to ensure that our gradients are calculated automatically. On another note, we have to specify our function to ensure that our gradient can be differentiated. Thus, we create a vector z which is a function of y which is also a function x. Doing so, will allow us to perform backpropagation automatically and we can obtain the gradient by calculated the partial derivative of z/x.

```
#%%implementation of SLP
x = torch.ones(1,requires_grad=True)
print(x.grad) #returns None initially
y=x+2
z=y*y*2
z.backward()
print(x.grad) #this is partial z/x


class Feedforward(torch.nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Feedforward, self).__init__()
        self.input_size = input_size
        self.hidden_size  = hidden_size
        self.fc1 = torch.nn.Linear(self.input_size,
                                   self.hidden_size)
        self.relu = torch.nn.ReLU()
        self.fc2 = torch.nn.Linear(self.hidden_size, 1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        hidden = self.fc1(x)
        relu = self.relu(hidden)
        output = self.fc2(relu)
        output = self.sigmoid(output)
        return output
```

The next step is to define our model. I was unsure of how to begin with creating this model, so I followed the steps given in the medium.com article provided given to us. I know that no model is perfectly designed for some type of data and each model has its own benefits and negatives. I am unsure of how to decide these models for myself and instead chose to follow the one given to us while changing the optimization parameters as we have learned about some of them.

Here, the feedforward model has hidden layers between the input and the output layer and after every hidden layer is an activation function that is applied to ensure non-linearity. One important thing to note is that the final output activation function is sigmoidal. Sigmoidal is a great function to use here when we are computing binary categories like in Dataset 10.

```
#%%implement the model w/ tensors
model = Feedforward(2,100)
criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr = 0.01)
```

We then define the model with input dimension 2 and hidden dimension 100. The person who wrote the article defined hidden layers to be 10 but I chose to increase these. The number of dimensions in any feedforward network is typically chosen based on the complexity of the problem being solved and the corresponding amount of data that is available. A general sentiment is that adding additional dimensions to a feedforward neural network can increase its representational power and thereby enable it to learn more complex functions.

I also changed the criterion and optimizer values multiple times. Leaving the learning rate at 0.01 seemed to be the best option for training our network however, the author of the article was using BCE (binary cross entropy) for loss. I altered this to Mean Squared Error Loss (MSE) and received notably lower loss and notably higher accuracy for my model.

Now, we can assess how our model is performing prior to training by setting our model into .eval mode. This will allow us to view how well our model is performing before and after training with our provided dataset. During training, I chose to implement training for 50 epochs total. Any more than that and the model began to plateau in accuracy and I wanted to avoid overfitting and refrain from performing superfluous training epochs.

```
#%%evaluate before training
model.eval()
y_pred = model(x_eval)
before_train = criterion(y_pred.squeeze(),y_eval)
print('test loss before training',before_train.item())

#%%evluate after training
model.train()
epoch = 50

epoch_val = []
loss_val = []

for epoch in range(epoch):
    optimizer.zero_grad()

    #forward pass
    y_pred = model(x_train)

    #loss computation
    loss = criterion(y_pred.squeeze(),y_train)
    epoch_val.append(epoch)
    loss_val.append(loss.detach())
    print('Epoch: {} \n Train Loss: {} \n'.format(epoch,loss.item()))

    #backward pass
    loss.backward()
    optimizer.step()

plt.figure()
sns.scatterplot(x = epoch_val,y=loss_val).set_title("Loss vs. Epoch #")
plt.xlabel("Epoch #")
plt.ylabel("Loss Value")
```

Finally, we have to evaluate the model and print our error rate. Doing so, we change our model from .train() to .eval() and calculate the predictions of our eval data and *not* the dev data. The values for performance regarding the /train, /dev, and /eval data are compiled in a table below

```
#%%evaluation
model.eval()
y_pred = model(x_eval)
after_train = criterion(y_pred.squeeze(),y_eval)
print('Test loss after training: ',after_train.item())

#%%accuracy
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

accuracy_train = accuracy(x_train,y_train)
accuracy_eval = accuracy(y_pred,y_eval)

print("Error of Train: ", str(1-accuracy_train))
print("Error of Eval: ", str(1-accuracy_eval))
```

B. TASK 2

The final request of this assignment was to compile all the data into a table for all algorithms that have been utilized for Dataset 10. This is the following table exclusively containing error rates:

DS	System	Training Data	Train	Dev	Eval
#10	KNN *baseline	/train	0.3210	0.1848	0.6090
	RNF *baseline	/train	0.0100	0.3940	0.3260
	KNN *imld	/train	0.0763	0.3883	0.3344
	RNF *imld	/train	0.0215	0.3974	0.3328
	FeedForward Neural Net	/train	0.5618	0.5418	0.5000
	RBF SVM	/train	0.0880	0.3951	0.3260
	PCA *baseline	/train	0.4699	0.4551	0.4871
	PCA Conventional + Bootstrapping	/train	0.4698	0.4699	0.4700
	PCA Conventional + CV	/train	0.4699	0.4491	0.5244
	PCA Bootstrap	/train	0.4501	0.4711	0.4201
	PCA Bagging	/train	0.0898	0.1339	0.1699
	PCA Boosting	/train	0.4722	0.4534	0.4883
	CI-PCA	/train	0.5301	0.5270	0.4892
	QDA	/train	0.8392	0.5524	0.6589
	LDA	/train	0.5301	0.5534	0.4893
	CI-PCA	/train + /dev	0.5282	0.5074	0.5092
	QDA	/train + /dev	0.8503	0.8247	0.6530
	LDA	/train + /dev	0.5301	0.5534	0.4893

C. CONCLUSION

The highest scores that we see after filling out this table with a variety of algorithms of data techniques are found in our random forest algorithm, RBF SVM, and PCA with bagging. All of the other algorithms seem to function at about the same error rate (~50%). It is always important to recall that these algorithms do not necessarily perform better than one another and that there is no single algorithm that is "the best." Instead, we just know that from our research, the three previously mentioned algorithms work the best with this data set. This can be due to data overlapping or our chosen parameters.

Overall, I think that this homework assignment really helped to solidify the concept that no algorithm is the 'best.' Instead, it was very insightful for me to realize that there are a ton of options to choose from when picking an algorithm to work with and the importance of keeping my options open when working with data.

The implementation of the feedforward neural network was also helpful because I suppose I never consciously realized that the choices of layers and activation functions are random. The personalization of these algorithms and the understanding of activation functions are what allow us to optimize these models to work with our data.