Deep Learning Homework 1
Jyotit Kaushal, 1005245

Question 1: Given the code executed above, can you describe the different elements of the Machine Learning problem that we seem to be currently facing? At the moment you should be able to describe the task (T), dataset (D), inputs and outputs (I, O). The model (M) and loss (L) will be discussed later.

Based on what we have done above we can clearly see that we seem to be working on a sort of classification problem, where the task (T) is: to define and train a model that is able to correctly predict a particular category (i.e. the outputs) given two sets of related values (i.e. the inputs)- so it's a Classification Task. Subsequently the Dataset (D) is being stored in the dataset.xlsx file and seems to have 2 features and 1 column of labels that are either 0 or 1. The inputs (I, O) are as mentioned earlier for inputs it is two columns stacked together each having values between -1 and 1, together they correspond to a point on the graph which either has 0 or 1 as the outputs (O)- an output of 0 corresponds to a green dot and that of a 1 corresponds to a red cross on the graph you can see above.

Question 2: Could we use a simple Logistic Regression model to solve this Machine Learning task?

Nope we can't use a simple Logistic Regression model in this case because of the simple fact that it simply wouldn't fit the data that we have too well. A Logistic Regression Model owing to how it's defined assumes a linear relationship between the input features and the log-odds of the target variable. As a result, it is essentially only able to model linear decision boundaries and is hence limited to a dataset that is linearly separable in the first place which the current dataset we are working on isn't. Hence, we won't be able to use the Logistic Regression Model to solve this ML task but instead probably need to rely on neural networks instead.

Question 3: Show the code for your CustomDataset object, after you have correctly figured out how to replace the different None variables.

```
class CustomDataset(torch.utils.data.Dataset):
   def __init__(self):
       self.dataframe = pd.read_excel('dataset.xlsx')
       self.dataset_length= len(self.dataframe)

       self.input_features= self.dataframe.iloc[:, :2]
       self.output_label= self.dataframe.iloc[:, 2]

   def __len__(self):
       return len(self.dataframe)

   def __getitem__(self, idx):
```

```
        # Select columns corresponding to the different inputs and outputs from the
dataframe we just created.
        # And convert to PyTorch tensors with the right dtype
        x1 = torch.tensor(self.dataframe.iloc[idx, 0], dtype=torch.float64)
        x2 = torch.tensor(self.dataframe.iloc[idx, 1], dtype=torch.float64)
        y = torch.tensor(self.dataframe.iloc[idx, 2], dtype=torch.float64)
        # Assemble all input features in a single inputs tensor with 2 columns and rows
for each sample in the dataset.
        inputs = torch.stack((x1, x2), dim=0)
        y= y
        return inputs, y
```

Question 4: What information about the dataset is the *len* method supposed to return?

The len function is supposed to return the size of the dataset, which is essentially the number of rows/entries in our dataset that we are currently working with. In this case, the dataset we are working with has 1024 entries as can be seen by using the len method.

Question 5: :D

Question 6: Can you figure out what to put in place of the None variables in the cell below? Show your code in your report.

```
# Define batch size
batch_size = 128
# Create DataLoader object
pt_dataloader = torch.utils.data.DataLoader(pt_dataset, batch_size= batch_size,
shuffle=True)
```

Question 7: Is this a valid activation function to use? Is it technically possible to have a trainable parameter in an activation function?

Yes the above activation function although weird does seem to be valid since what we are doing is essentially still working with a ReLU function just that in this case to spell it out, if the output of a neuron x is positive- then we are looking at the fist ReLU being activated and the second term will of course be reduced to a*0. In this case, we encounter a value of x that is negative- we have the first ReLU term that returns a 0 (as input <0) whereas the second term gives us the magnitude of x scaled by our trainable parameter alpha. Giving us -(alpha*x). We can say it is valid because we have a clear continuous distribution of output values from the ouput function and is also differentiable so we can aptly do gradient-based optimization on it as well. As for the second question, I would say yes- it is technically possible to have a trainable parameter in an activation function although it's not seen traditionally. There was an introduction to it in Google's

Swish paper that demonstrates how we can have another trainable parameter as part of the activation function.

Question 8: In the code below, we will define a *WeirdActivation* object, implementing our activation function. As before, there are a few *None* variables that probably need to be replaced. Show your code for the *WeirdActivation* object in your report. You should probably use the *torch.relu(x)* function in your implementation.

```
class WeirdActivation(torch.nn.Module):
   def __init__(self, a, device):
       super().__init__()
       self.a = nn.Parameter(torch.tensor([a], dtype = torch.float64, device = device,
requires_grad = True))

   def forward(self, x):
       relu= nn.ReLU()
       x= relu(x)-self.a*(relu(-x))
       return x
```

Question 9: How would you describe this *WeirdActivation* function? Does it resemble another activation function we have discussed in class?

What we are looking at from this WeirdActivation function is that while the output of a neuron x is positive- then we are looking at the fist ReLU being activated and the second term will of course be reduced to a*0. In this case, we encounter a value of x that is negative- we have the first ReLU term that returns a 0 (as input <0) whereas the second term gives us the magnitude of x scaled by our trainable parameter alpha. Giving us -(alpha*x). What this basically ends up being which is confirmed when we look at the graphs below is that this function is basically changing the normal ReLU function that reduces all negative values into 0 we are now able to define and perhaps train a scalar alpha/a that allows us to still get some information from the output when the input is negative. The higher the value of this scaler- the more of the negative value we can see pass through and in case where a is 1, we see it's ReLU but where -x gives -x on the other side. This function suspiciously looks like the Leaky ReLU function we looked at in class.

Question 10: Would it be a good idea to use our WeirdActivation layer as the final activation layer for our model? Or would it be preferable to use a Sigmoid?

No, it wouldn't make as much sense to use our WeirdActivation layer which we have identified as now to be something that is quite similar to a standard ReLU function or a Leaky ReLU function in what we have seen it work/do. This is because we have identified the task given to us as a classification task based on the dataset we are working with. Given that, it doesn’t make as much sense to use a final activation function that gives you values that go up until infinity and for which it’s going to be really hard to get a threshold that allows us to make that classification

we are looking for. Sigmoid activation function on the other hand gives us a direct probability of which class it thinks the current data point belongs to and by simply setting the threshold as 0.5 or something we can get a clear and concise classification. Therefore, it is preferable to use a sigmoid function which we have done ultimately.

Question 11: Following your answer for Question 10, show your final code for the *NeuralNetwork* class in your report.

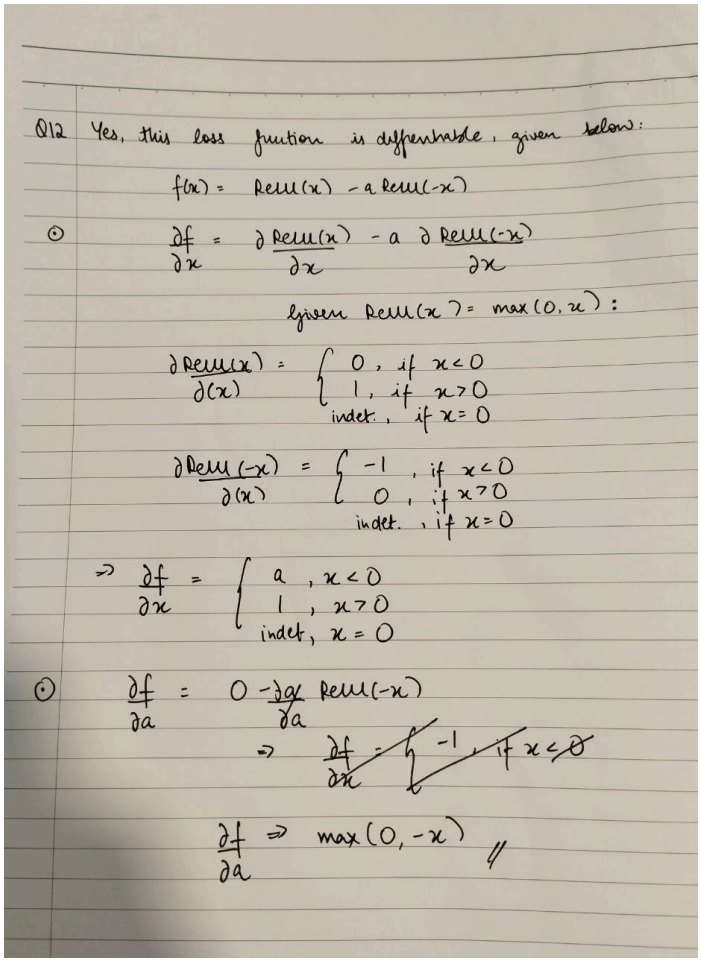
```
class NeuralNetwork(nn.Module):
   def __init__(self, n_x, n_h, n_y, a):
       # Super init
       super(NeuralNetwork, self).__init__()
       self.n_x, self.n_h, self.n_y, self.a= n_x, n_h, n_y, a

       # Layers to use
       self.fc1 = nn.Linear(n_x, n_h, dtype = torch.float64)
       self.weird_activation = WeirdActivation(a, device = device)
       self.fc2 = nn.Linear(n_h, n_y, dtype = torch.float64)
       self.final_activation = nn.Sigmoid()

       # Loss and Accuracy metrics
       self.loss = nn.BCELoss()
       self.accuracy = BinaryAccuracy()

   def forward(self, x):
       # All operations for forward, in order
       out1 = self.fc1(x)
       out2 = self.weird_activation(out1)
       out3 = self.fc2(out2)
       out4 = self.final_activation(out3)
       return out4
```

Question 12: Is this function differentiable? Can you explicitly describe the partial derivatives of this function $ f $ with respect to $ x $ and $ a $?



Question 13: Why are we using *backward()* on the Binary Cross Entropy loss calculated and stored in the variable *loss_value* instead of the Binary Accuracy value calculated and stored in the variable *binary_accuracy_value*? What would happen if we changed *loss_value.backward()* and replaced it with *binary_accuracy_value.backward()* instead?

The answer of this question simply has to do with the fact as to what backward propagation does in the first palace and what is the purpose of having it as part of a neural model. Since with backward propagation we are trying to essentially teach the model on how bad a particular prediction is and it does so by basically gradient descent. In the case of the binary accuracy value, since it's always going to be either a 0 or a constant (in this case 1) since it's binary we will either have a 1 gradient or indeterminate gradient if it classifies the entry as 0. Therefore, if we use the binary_accuracy_value and run the backward() propagation on it, in the end we will have a model that is not learning anything and we will do our task. Therefore, we do it on the loss_value instead, which allows for smooth gradient calculations hence helping in guiding the model to perform better. The binary_accuracy_value model is simply a final measure for us to see what's the performance/accuracy of the model at a certain epoch or point in training.

Question 14: When running the cell above, it seems the model is not capable of achieving a great accuracy. If anything, it seems to remain stuck at an accuracy of 60% or so. Did we make a mistake in one of our hyperparameters (e.g. the learning rate)? Or is the model simply incapable of figuring out this task (maybe because of the weird activation function we have decided to use)?

The reason for it actually becomes quite obvious when you look at the way the Adam optimizer is set up in the block of code above. The learning rate for it has been set to 1e3 which as several papers have shown is way too high and what ends up happening is that we are always making changes to the weights of the model that are way too high and hence doing an improper gradient descent that doesn't allow us to converge at not only a global minima but even any local minima too.

Question 15: Assuming you figured out what was wrong in Question 14, show in your report how you resolved the problem in the code above.
Your model should be able to produce a final training accuracy above 98%.

As defined in the problem above with the learning rate, with changing the learning rate to 4e-2 (this was done by doing a little bit of testing on my own end) that involved setting the learning rates from a range of 1e-2 to 1e-3, while with 1e-3 I saw that the training rate was really reduced so 15 epochs weren't enough to get the desired 98%+ accuracy. Eventually I ended up using 4e-2 and with a torch seed of 17 set at the start of the notebook that helped us get a 98.44% accuracy in the 15th epoch with a training loss of 0.0875.

```
optimizer = torch.optim.Adam(model.parameters(),
                             lr = 4e-2, # original 1e3
                             betas = (0.9, 0.999),
                             eps = 1e-08)
optimizer.zero_grad()
```

Final output:

```
Epoch [1/15], Training Loss: 0.6531, Training Accuracy: 0.6484
Epoch [2/15], Training Loss: 0.5135, Training Accuracy: 0.8828
Epoch [3/15], Training Loss: 0.3180, Training Accuracy: 0.9922
Epoch [4/15], Training Loss: 0.2240, Training Accuracy: 0.9766
Epoch [5/15], Training Loss: 0.1726, Training Accuracy: 0.9766
Epoch [6/15], Training Loss: 0.1626, Training Accuracy: 0.9297
Epoch [7/15], Training Loss: 0.1065, Training Accuracy: 0.9688
Epoch [8/15], Training Loss: 0.1192, Training Accuracy: 0.9531
Epoch [9/15], Training Loss: 0.0754, Training Accuracy: 0.9922
Epoch [10/15], Training Loss: 0.0931, Training Accuracy: 0.9766
Epoch [11/15], Training Loss: 0.1008, Training Accuracy: 0.9922
Epoch [12/15], Training Loss: 0.0921, Training Accuracy: 0.9766
Epoch [13/15], Training Loss: 0.0660, Training Accuracy: 0.9922
Epoch [14/15], Training Loss: 0.0908, Training Accuracy: 0.9609
Epoch [15/15], Training Loss: 0.0875, Training Accuracy: 0.9844
```

Question 16: What does the code *print(model.weird_activation.a.item())* above show? Shouldn't the value of *a* be $ 0.2 $ as initially set upon the creation of our neural network?

Nope it's actually printing -1.69 this is because as given earlier we set the alpha as a trainable parameter which was basically what we had as the weird activation function so it makes sense that after training the model on our own accord the value of ‘a’ was changed from what it was initialized as.

Question 17: Assuming you figured out what is wrong in Question 15, you should be able to obtain a final 98% training accuracy score. Is this score good enough to guarantee that the model will be able to generalize? Or would it be preferable to use a different metric instead? (You might want to look at the files provided for this homework for a hint...)

Nope, it’s actually not good enough to guarantee that the model will be able to generalize and in the case of a neural network this doesn’t even have anything to do with the score at all. At the training time all the accuracy score tells us is how well the model is predicting the labels on the data that it’s training on. This means that it could be that it’s doing quite well on the training dataset but has now overfitted to the data (i.e. learning all the intricacies of it) and hence is not generalizable at all. But in the case of neural networks the thing is that we do still expect quite a high training score in general since they are so good at capturing all the intricacies of the data so it’s not uncommon especially with using 10 neurons that we see a 98% training accuracy for a relatively simple dataset. Therefore, I do still think that it would be generalizable just that we can't simply base the training accuracy score.
Therefore, instead of using the training accuracy score to check generalizability we should probably run the model on a second/different unseen set of data and see how well the trained model scores on that.

Question 18: Having figured out how to prove/disprove generalization in Question 17, we leave the rest of the notebook cells for you to play with the code and figure out how to prove that your model is indeed capable of generalization (or not). Show your code, your results and how it matches the reasoning you described in Question 17. Is your model truly capable of solving this classification task in the end?

Like I said earlier we will now running the trained model in evaluation mode on the new dataset that has been given to us in the unseen_data.xlsx. To do this we simply create testing_inputs, and ground_truth_labels tensors using the dataset and then run the model on that.
We then calculate the binary_accuracy_value like we did when training the model.

```
import pandas as pd

test_data = pd.read_excel("unseen_dataset.xlsx")
```

```
testing_inputs = torch.tensor(test_data[['x1', 'x2']].values,
dtype=torch.float64).to(device)
ground_truth_labels= torch.tensor(test_data[['class']].values,
dtype=torch.float64).to(device)



model.eval()
with torch.no_grad():
   test_predictions = model(testing_inputs)

binary_accuracy_value = model.accuracy(test_predictions, ground_truth_labels)

print(f'Testing Accuracy: {binary_accuracy_value.item():.4f}')
```

We get the results as: (again with seed 17 like said before so you can check for yourself). The 95% accuracy does confirm my hypothesis in the previous question. So yes I would say my model is truly capable of solving this classification task in the end.

Testing Accuracy: 0.9531

Question 19: Looking back at the dataset and the samples distribution, do you feel *feature engineering* could have been useful here? If so, which synthetic features would you craft for this dataset? Bonus points might be given if you show code demonstrating some feature engineering.

The dataset and the sample distribution does make me actually feel like there might be something we can do with feature engineering.This is because from the graph we can quickly see that there does seem to be some sort of curve on which the labels are based upon that has a square function somewhere in there. As a result I feel like adding some features that are squares of our two original features might give us some performance benefits.

We do this with the code below, as you can see we have basically slightly modified the CustomDataset class into CustomDatasetfeateng to scale to the additional number of features we are synthetically generating. We then change our n_x parameters accordingly and we get the following that has been changed and run:

```
class CustomDatasetfeateng(torch.utils.data.Dataset):
   def __init__(self):
       self.dataframe = pd.read_excel('dataset.xlsx')
       self.dataset_length= len(self.dataframe)

       self.input_features= self.dataframe.iloc[:, :2]
       self.output_label= self.dataframe.iloc[:, 2]
```

```
   def __len__(self):
       return len(self.dataframe)

   def __getitem__(self, idx):
       # Select columns corresponding to the different inputs and outputs from the
dataframe we just created.
       # And convert to PyTorch tensors with the right dtype
       x1 = torch.tensor(self.dataframe.iloc[idx, 0], dtype=torch.float64)
       x1_squared= x1.pow(2)

       x2 = torch.tensor(self.dataframe.iloc[idx, 1], dtype=torch.float64)
       x2_squared= x2.pow(2)

       y = torch.tensor(self.dataframe.iloc[idx, 2], dtype=torch.float64)
       # Assemble all input features in a single inputs tensor with 2 columns and rows
for each sample in the dataset.
       inputs = torch.stack((x1, x2, x1_squared, x2_squared), dim=0)
       y= y
       return inputs, y
```

```
# Create Neural Network model
n_x = 4
n_h = 10 # originally 10
n_y = 1
a = 0.2 # originally 0.2
batch_size= 128
model_feateng = NeuralNetwork(n_x, n_h, n_y, a).to(device)
pt_dataset_feateng= CustomDatasetfeateng()
pt_dataloader_feateng= torch.utils.data.DataLoader(pt_dataset_feateng, batch_size=
batch_size, shuffle=True)



# Gradient descent parameters: optimizers, repetitions, etc.
num_epochs = 15 # originally 15
optimizer = torch.optim.Adam(model_feateng.parameters(),
                             lr = 4e-2, # originall 1e3
                             betas = (0.9, 0.999),
                             eps = 1e-08)
optimizer.zero_grad()

for epoch in range(num_epochs):
   for batch in pt_dataloader_feateng:
```

```
        # Unpack the mini-batch data
        inputs_batch, outputs_batch = batch
        outputs_re = outputs_batch.to(device).reshape(-1, 1)
        inputs_re = inputs_batch.to(device)

        # Forward pass
        pred = model_feateng(inputs_re)
        loss_value = model_feateng.loss(pred, outputs_re)
        # Compute binary accuracy
        binary_accuracy_value = model_feateng.accuracy(pred, outputs_re)

        # Backward pass and optimization
        loss_value.backward()
        # binary_accuracy_value.backward()
        optimizer.step()
        optimizer.zero_grad()

    # Print loss and accuracy
    print(f'Epoch [{epoch+1}/{num_epochs}], Training Loss: {loss_value.item():.4f},
Training Accuracy: {binary_accuracy_value.item():.4f}')
```

Results we get are as follows:

```
Epoch [1/15], Training Loss: 0.6373, Training Accuracy: 0.6016
Epoch [2/15], Training Loss: 0.4265, Training Accuracy: 0.9062
Epoch [3/15], Training Loss: 0.2607, Training Accuracy: 0.9531
Epoch [4/15], Training Loss: 0.1217, Training Accuracy: 0.9922
Epoch [5/15], Training Loss: 0.1272, Training Accuracy: 0.9766
Epoch [6/15], Training Loss: 0.0747, Training Accuracy: 0.9922
Epoch [7/15], Training Loss: 0.0862, Training Accuracy: 1.0000
Epoch [8/15], Training Loss: 0.0870, Training Accuracy: 0.9609
Epoch [9/15], Training Loss: 0.0651, Training Accuracy: 0.9844
Epoch [10/15], Training Loss: 0.0532, Training Accuracy: 0.9844
Epoch [11/15], Training Loss: 0.0533, Training Accuracy: 0.9922
Epoch [12/15], Training Loss: 0.0601, Training Accuracy: 0.9922
Epoch [13/15], Training Loss: 0.0393, Training Accuracy: 0.9922
Epoch [14/15], Training Loss: 0.0318, Training Accuracy: 0.9922
Epoch [15/15], Training Loss: 0.0679, Training Accuracy: 0.9609
```

As we can see the model is much quicker to converge and yes although we see the final accuracy in the 15th epoch is a bit less than our original it's simply a bit to do with the learning rate but we can easily work past that by simply doing a check over some of the previous epochs and choosing the value with the best accuracy and less training loss. It's clear to see **the training loss was also much quicker to go down** despite using the same learning rate and optimizer. So although doing this feature engineering might not have been a big win it did still have it's merits.

Next, we have the following code to also test it's generalizability:

```
import pandas as pd

test_data = pd.read_excel("unseen_dataset.xlsx")

og_inputs = torch.tensor(test_data[['x1', 'x2']].values,
dtype=torch.float64).to(device)
og_inputs_squared= og_inputs**2
testing_inputs = torch.tensor(torch.cat((og_inputs, og_inputs_squared), axis=1),
dtype=torch.float64).to(device)


ground_truth_labels= torch.tensor(test_data[['class']].values,
dtype=torch.float64).to(device)


model_feateng.eval()
with torch.no_grad():
   test_predictions = model_feateng(testing_inputs)

binary_accuracy_feateng = model_feateng.accuracy(test_predictions,
ground_truth_labels)

print(f'Testing Accuracy: {binary_accuracy_feateng.item():.4f}')
```