M7-HW2

October 30, 2023

1 Problem 2

1.1 Problem Description

In this problem you will train a neural network to classify points with features x_0 and x_1 belonging to one of three classes, indicated by the label y. The structure of your neural network is up to you, but you must describe the structure of your network, training parameters, and report an accuracy for your fitted model on the provided data.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

- Visualization of provided data
- Visualization of trained model with provided data
- Trained model accuracy
- Discussion of model structure and training parameters

Imports and Utility Functions:

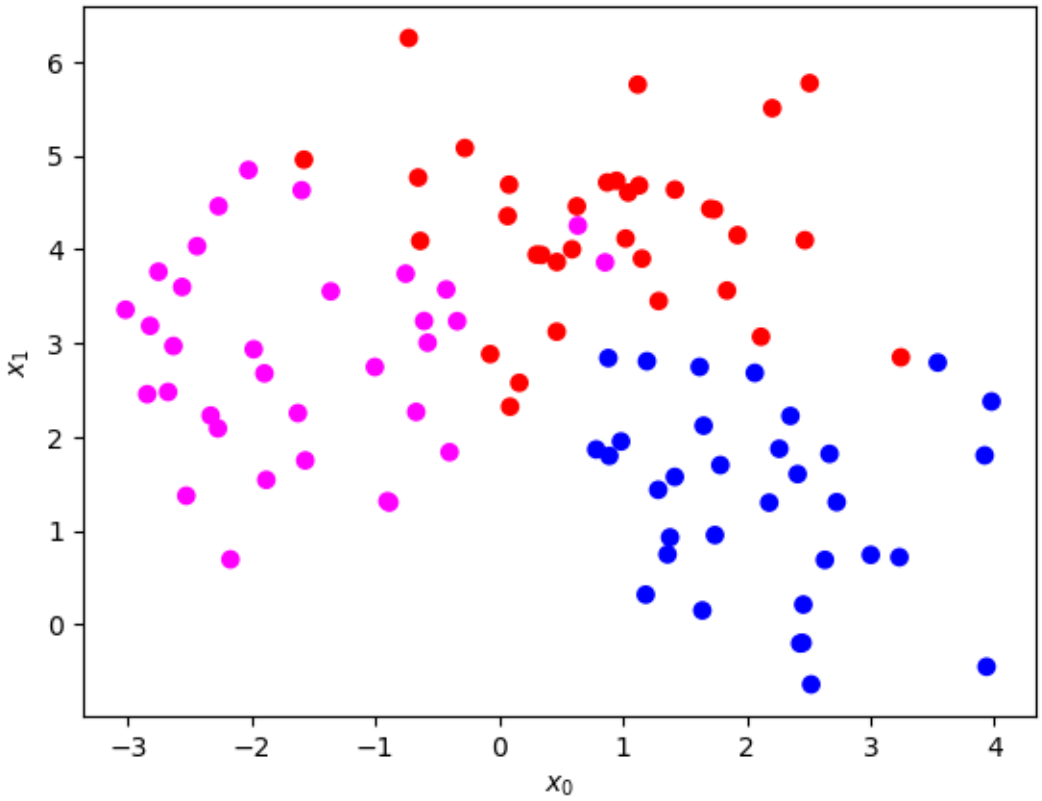
[ ]:
```
import torch
import torch.nn as nn
from torch import optim, nn
import torch.nn.functional as F
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def dataGen():
    # random_state = 0 set so generated samples are identical
    x, y = datasets.make_blobs(n_samples = 100, n_features = 2, centers = 3,␣
 ↪random_state = 0)
    return x, y

def visualizeModel(model):
    # Get data
    x, y = dataGen()
```

```
    # Number of data points in meshgrid
    n = 100
    # Set up evaluation grid
    x0 = torch.linspace(min(x[:,0]), max(x[:,0]),n)
    x1 = torch.linspace(min(x[:,1]), max(x[:,1]),n)
    X0, X1 = torch.meshgrid(x0, x1, indexing = 'ij')
    Xgrid = torch.vstack((X0.flatten(),X1.flatten())).T
    Ypred = torch.argmax(model(Xgrid), dim = 1)
    # Plot data
    plt.scatter(x[:,0], x[:,1], c = y, cmap =␣
 ↪ListedColormap(['red','blue','magenta']))
    # Plot model
    plt.contourf(Xgrid[:,0].reshape(n,n), Xgrid[:,1].reshape(n,n), Ypred.
 ↪reshape(n,n), cmap = ListedColormap(['red', 'blue', 'magenta']), alpha = 0.
 ↪15)
    plt.xlabel('$x_0$')
    plt.ylabel('$x_1$')
    plt.show()
```

1.2 Generate and visualize the data

Use the `dataGen()` function to generate the x and y data, then visualize with a 2D scatter plot, coloring points according to their labels.

[ ]:
```
x, y = dataGen()
def getArray(index):
    arr = np.zeros(3)
    arr[index] = 1
    return arr
y_new = np.array([getArray(i) for i in y])

plt.scatter(x[:,0], x[:,1], c = y, cmap =␣
 ↪ListedColormap(['red','blue','magenta']))
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.show()
```



1.3 Create and train a neural network using PyTorch

Choice of structure and training parameters are entirely up to you, however you will need to provide reasoning for your choices. An accuracy of 0.9 or more is reasonable.

Hint: think about the number out nodes in your output layer and choice of output layer activation function for this multi-class classification problem.

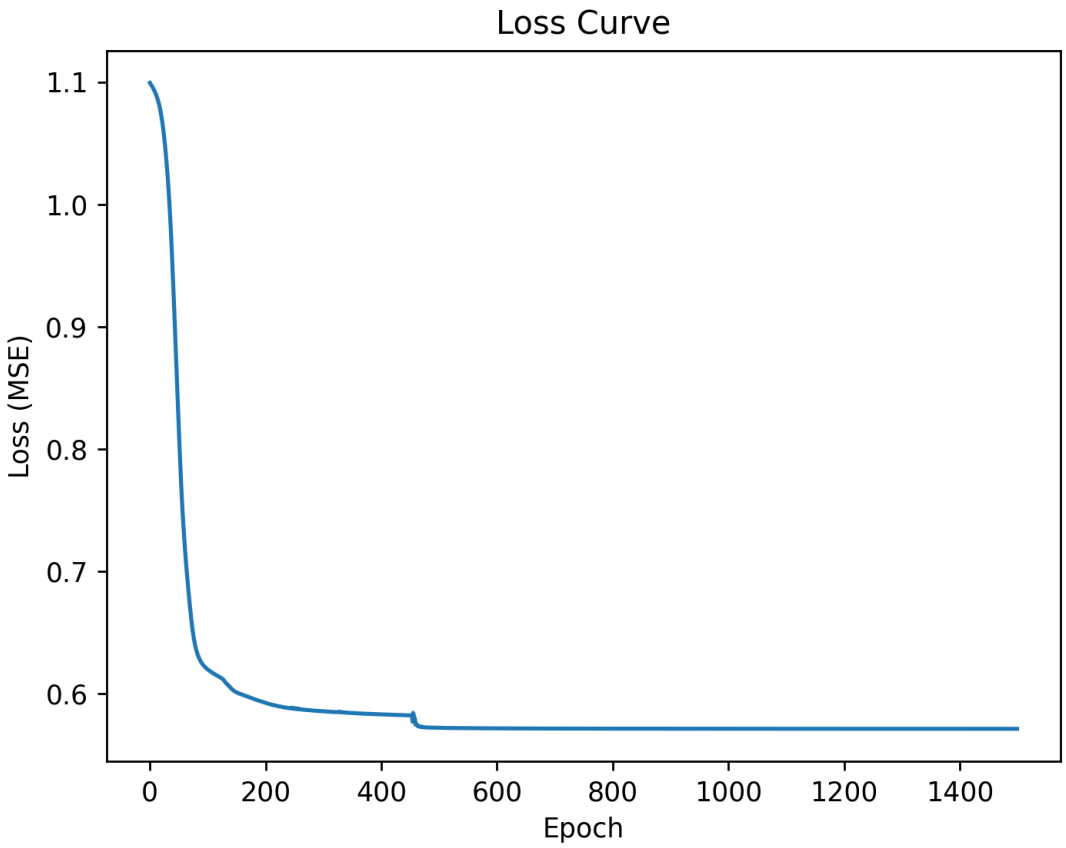
[ ]:
```
class NNet(nn.Module):
    def __init__(self, N_hidden=6, N_in=2, N_out=3):
        super().__init__()
        self.seq = nn.Sequential(
            nn.Linear(N_in, N_hidden),
            nn.LeakyReLU(),
            nn.Linear(N_hidden, N_hidden),
            nn.Tanh(),
            nn.Linear(N_hidden, N_hidden),
            nn.LeakyReLU(),
            nn.Linear(N_hidden, N_hidden),
            nn.LeakyReLU(),
            nn.Linear(N_hidden, N_out),
            nn.Sigmoid()
```

```
        )

    def forward(self,x):
        return self.seq(x)

x = torch.Tensor(x)
y = torch.Tensor(y.reshape(-1,1))
y_new = torch.Tensor(y_new)

model = NNet(N_hidden = 10)
loss_curve = []
accuracy = []

# Training parameters: Learning rate, number of epochs, loss function
# (These can be tuned)
lr = 0.005
epochs = 1500
loss_fcn = nn.CrossEntropyLoss()

# Set up optimizer to optimize the model's parameters using Adam with the␣
 ↪selected learning rate
opt = optim.Adam(params = model.parameters(), lr=lr)

# Training loop
for epoch in range(epochs):
    out = model(x) # Evaluate the model

    loss = loss_fcn(out,y_new)

    loss_curve.append(loss.item())
    acc = out.max(axis=1).indices == y.squeeze(1)

    # Print loss progress info 25 times during training
    if epoch % int(epochs / 25) == 0:
        print(f"Epoch {epoch} of {epochs}... \tAverage loss: {loss.item()}␣
 ↪\tAccuracy: {acc.float().sum() / y.squeeze(1).shape[0]}")

    # Move the model parameters 1 step closer to their optima:
    opt.zero_grad()
    loss.backward()
    opt.step()

plt.figure(dpi=250)
plt.plot(loss_curve)
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.title('Loss Curve')
```

```
plt.show()
```

```
Epoch 0 of 1500…      Average loss: 1.0993950366973877        Accuracy:
0.3400000035762787
Epoch 60 of 1500…     Average loss: 0.7226464748382568        Accuracy:
0.9200000166893005
Epoch 120 of 1500…    Average loss: 0.6139636635780334        Accuracy:
0.9399999976158142
Epoch 180 of 1500…    Average loss: 0.5958366990089417        Accuracy:
0.9700000286102295
Epoch 240 of 1500…    Average loss: 0.5884358286857605        Accuracy:
0.9700000286102295
Epoch 300 of 1500…    Average loss: 0.5857990980148315        Accuracy:
0.9700000286102295
Epoch 360 of 1500…    Average loss: 0.5840986967086792        Accuracy:
0.9700000286102295
Epoch 420 of 1500…    Average loss: 0.5829318165779114        Accuracy:
0.9700000286102295
Epoch 480 of 1500…    Average loss: 0.5725138783454895        Accuracy:
0.9800000190734863
Epoch 540 of 1500…    Average loss: 0.5720308423042297        Accuracy:
0.9800000190734863
Epoch 600 of 1500…    Average loss: 0.5718442797660828        Accuracy:
0.9800000190734863
Epoch 660 of 1500…    Average loss: 0.5717346668243408        Accuracy:
0.9800000190734863
Epoch 720 of 1500…    Average loss: 0.5716642737388611        Accuracy:
0.9800000190734863
Epoch 780 of 1500…    Average loss: 0.5716164112091064        Accuracy:
0.9800000190734863
Epoch 840 of 1500…    Average loss: 0.5715824365615845        Accuracy:
0.9800000190734863
Epoch 900 of 1500…    Average loss: 0.5715575814247131        Accuracy:
0.9800000190734863
Epoch 960 of 1500…    Average loss: 0.5715387463569641        Accuracy:
0.9800000190734863
Epoch 1020 of 1500…   Average loss: 0.5715242624282837        Accuracy:
0.9800000190734863
Epoch 1080 of 1500…   Average loss: 0.5715128183364868        Accuracy:
0.9800000190734863
Epoch 1140 of 1500…   Average loss: 0.5715035796165466        Accuracy:
0.9800000190734863
Epoch 1200 of 1500…   Average loss: 0.5714960694313049        Accuracy:
0.9800000190734863
Epoch 1260 of 1500…   Average loss: 0.5714898705482483        Accuracy:
0.9800000190734863
Epoch 1320 of 1500…   Average loss: 0.5714847445487976        Accuracy:
0.9800000190734863
```

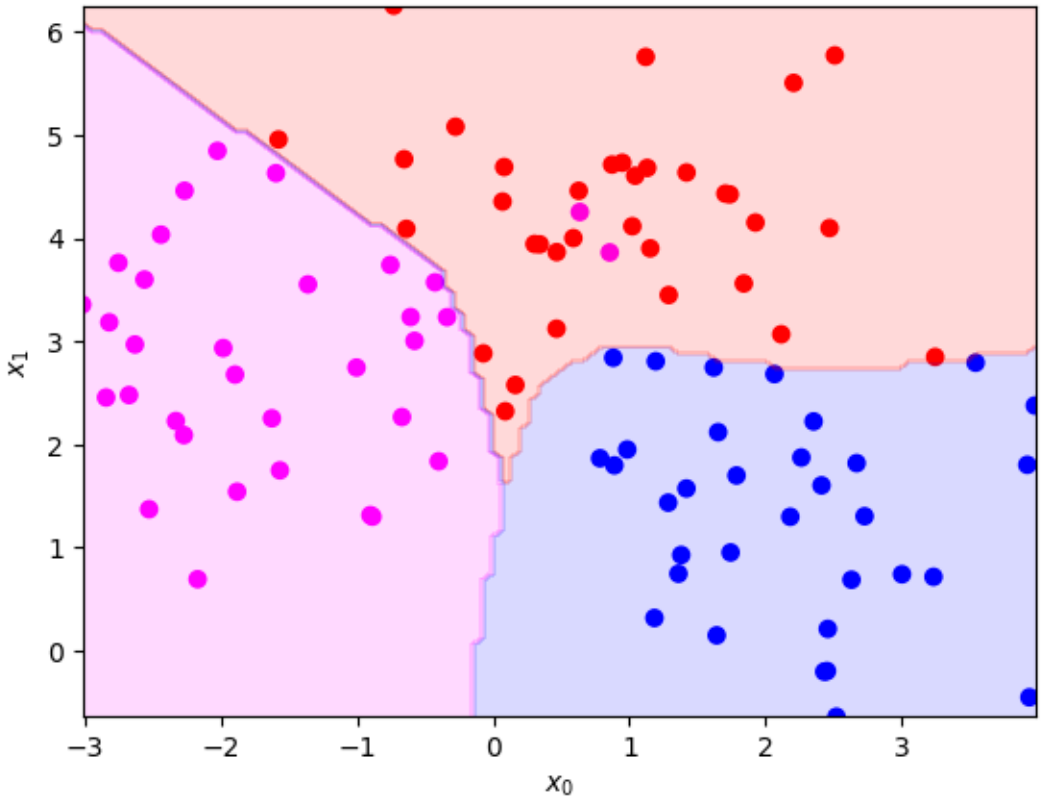
```
Epoch 1380 of 1500…   Average loss: 0.5714803338050842        Accuracy:
0.9800000190734863
Epoch 1440 of 1500…   Average loss: 0.5714766979217529        Accuracy:
0.9800000190734863
```



1.4 Visualize your trained model

Use the provided `visualizeModel()` function by passing in your trained model to see your models predicted function compared to the provided data

[ ]:
```
visualizeModel(model)
```



1.5 Discussion

Report the accuracy of your trained model on the generated data. Discuss the structure of your network, including the number and size of hidden layers, choice of activation function, loss function, optimizer, learning rate, number of training epochs.

number and size of hidden layers: The number of hidden layers was chosen to minimize the number of nodes and layers while still producing consistently smooth and accurate results. This was iterated on until a final number of layers and nodes per layer was decided on. The activation functions were chosen to produce smooth output results while also eliminating unnecessary nodes. For this reason the LeakyReLU function was used for most of the inner layers with the second layer having an activation function of tanh. This was done to smooth out the final results a bit more as the LeakyReLU function produced very jagged dividing lines. Cross entropy loss was used for the loss function because it measures the performance od probability classification models and is useful for multi-class classification problems such as this one. The Adam optimizer was used as the optimizer because it provides a varying gradient to speed up or slow down training when necessary. This provides faster and more accurate training results. The learning rate used was determined after trying out a few in the range of 0.05 - 0.0005. 0.005 was found to speed up training enough while also producing accurate and not over trained results. A similar method was used for determining the number of epochs while making sure that training had slowed down enough but not allowed to go on for too long and risk over fitting.