Daniel, Eric, Tony

Data Science Lab: Lab 6

Submit:

A pdf of your notebook with solutions. A link to your colab notebook or also upload your .ipynb if not working on colab.

Goals of this Lab

Fully Connected Models and XOR

1. How to create data objects that pytorch can use
2. How to create a dataloader
3. How to define a basic fully connected single layer model
4. How to define a multi-layer fully connected model
5. How to add non-linear activation functions.
6. How to add layers in two different ways

We also see the importance of nonlinear activation functions directly, by experimenting with the simple 4-data-point XOR example that we saw in class.

We also see that neural networks are fundamentally different than, say, logistic regression or linear regression, in that they can get stuck in local minima. A simple way to say this is that a different random seed may lead to a different solution.

We explore how capacity (the size of the neural network) could affect this situation.

Note: Make sure that you fix a random seed so that you can replicate your solutions. This is always a good practice, and in particular important here, as we see below.

```
import torch
import numpy as np
import time
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
```

First we define a linear regressor.

This is the same as a fully connected layer. It will be a building block in making deeper neural networks with fully connected layers.

```
# We define our first class: LinearRegressor
#
class LinearRegressor(torch.nn.Module):
```

```
    def __init__(self, input_dim, output_dim):
        """
        Define the layer(s) needed for the linear model.
        """
        super().__init__()
        self.linear = torch.nn.Linear(input_dim, output_dim, bias = True) # just linear

    def forward(self, x):
        """
        Calculate the regression score (MSE).

        Input:
            x (float tensor N x d): input rows
        Output:
            y (float tensor N x 1): regression output
        """
        x = self.linear(x)
        return torch.flatten(x)


    # defining a separate predict function is useful for multi-class
    # classification as we will see later. Here it is
    # unnecessary.

    def predict(self, x):
        """
        Predict the regression label of the input vector.

        Input:
            x (float tensor N X d): input images
        Output:
            y (float tensor N x 1): regression output
        """
        x = self.linear(x)
        return torch.flatten(x)
```

Problem 1:

Now you will use torch.nn.Sequential (see https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html) to construct a two layer neural network, with two fully connected layers (no non-linearity yet). Thus, you will combine torch.nn.Sequential with torch.nn.Linear that you saw above.

Design your network so that the first layer has as many neurons as the input.

Note: you have only one line to fill in here.

```
class TwoLayerLinearRegressor(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        """
        Define a model that stacks two linear fully connected layers.
```

```
        """
        super().__init__()
        self.TLL = torch.nn.Sequential(torch.nn.Linear(input_dim, input_dim, bias=True), torch.nn.Linear(input_dim, output_dim, bias=True))

    def forward(self, x):
        """
        Calculate the regression score (MSE).

        Input:
            x (float tensor N x d): input rows
        Output:
            y (float tensor N x 1): regression output
        """
        x = self.TLL(x)
        return torch.flatten(x)

    def predict(self, x):
        """
        Predict the regression label of the input vector.

        Input:
            x (float tensor N X d): input images
        Output:
            y (float tensor N x 1): regression output
        """
        x = self.TLL(x)
        return torch.flatten(x)
```

Problem 2

Now you will create the same network, but using different syntax: you will not use torch.nn.Sequential. You need to fill in the two lines as noted by the comments.

```
class TwoLayerLinearRegressor2(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        """
        Define a model that stacks two linear fully connected layers.
        """
        super().__init__()
        self.fc1 = torch.nn.Linear(input_dim, input_dim, bias=True)
        self.fc2 = torch.nn.Linear(input_dim, output_dim, bias=True)

    def forward(self, x):
        """
        Calculate the regression score (MSE).

        Input:
            x (float tensor N x d): input rows
```

```
        Output:
            y (float tensor N x 1): regression output
        """

        x = self.fc1(x)
        x = self.fc2(x)
        return torch.flatten(x)

    def predict(self, x):
        """
        Predict the regression label of the input vector.

        Input:
            x (float tensor N X d): input images
        Output:
            y (float tensor N x 1): regression output
        """
        x = self.fc1(x)
        x = self.fc2(x)
        return torch.flatten(x)
```

Problem 3

Now you will define a 2 layer neural network with ReLU activation at the first layer. In other words:

Let x be the input. Then writing $z = Wx + c$, $h =$ReLU(z) is the first layer's neurons. Then the output is $y = w \cdot z + d$.

Create this neural network using the torch.nn.Sequential command. Conceptually, it may help to realize that this neural network is: a fully connected layer followed by a ReLU, followed by a fully connected layer.

Also see: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

Note: You have only one line to fill in here.

```
class TwoLayerNonLinearRegressor(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        """
        Define a model that has a linear layer, a ReLU layer and another linear layer.
        """
        super().__init__()
        self.linear = torch.nn.Sequential(torch.nn.Linear(input_dim, input_dim, bias=True), torch.nn.ReLU(), torch.nn.Linear(input_dim, output_dim, bias=True))

    def forward(self, x):
        """
        Calculate the regression score (MSE).

        Input:
            x (float tensor N x d): input rows
        Output:
            y (float tensor N x 1): regression output
        """
        x = self.linear(x)
        return torch.flatten(x)
```

```
    def predict(self, x):
        """
        Predict the regression label of the input vector.

        Input:
            x (float tensor N X d): input images
        Output:
            y (float tensor N x 1): regression output
        """
        x = self.linear(x)
        return torch.flatten(x)
```

Problem 4

Do this one more time, but now without torch.nn.Sequential.

You have three lines to fill in here.

```
# We now do this again, without using nn.sequential
# in order to illustrate different syntax.

class TwoLayerNonLinearRegressor2(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        """
        Define a model that has a linear layer, a ReLU layer and another linear layer.
        """
        super().__init__()
        # self.fc1 = torch.nn.Linear(input_dim, input_dim, bias=True)
        # self.relu = torch.nn.ReLU()
        # self.fc2 = torch.nn.Linear(input_dim, output_dim, bias=True)
        self.fc1 = torch.nn.Linear(input_dim, 15, bias=True)
        self.relu = torch.nn.ReLU()
        self.fc2 = torch.nn.Linear(15, output_dim, bias=True)

    def forward(self, x):
        """
        Calculate the regression score (MSE).

        Input:
            x (float tensor N x d): input rows
        Output:
            y (float tensor N x 1): regression output
        """

        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return torch.flatten(x)

    def predict(self, x):
```

```
        """
        Predict the regression label of the input vector.

        Input:
            x (float tensor N X d): input images
        Output:
            y (float tensor N x 1): regression output
        """
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return torch.flatten(x)
```

Problem 5 (Nothing to turn in)

Read the documentation https://pytorch.org/docs/stable/optim.html to see what are the options pytorch provides for an optimizer, and what the parameters are.

Problem 6

Explain what the code

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

and

```
model.to(device)
x = x.to(device)
y = y.to(device)
```

does. This does not have to be a completely rigorous definition -- just explain enough to make clear that you understand what the code is doing and why it's important.

ANSWER:

The first code cell sets the device to use a GPU if one is available, otherwise use a CPU.

The second code cell moves the model, input data, and target to the device that was set in the first code cell so that training can be done on the selected device.

Problem 7

Modify the code below for the training loop, so that in addition to training, you output a curve of the training error over each epoch.

```
# Now we define a function for training
# Note each of the arguments that it takes

def train(model, data_train, data_val, device, lr=0.01, epochs=5000):
    """
    Train the model.

    Input:
      model (torch.nn.Module): the model to train
      data_train (torch.utils.data.Dataloader): yields batches of data
      data_val (torch.utils.data.Dataloader): use this to validate your model
      device (torch.device): which device to use to perform computation

      (optional) lr: learning rate hyperparameter
      (optional) epochs: number of passes over dataloader
    """

    # Setup the loss function to use: mean squared error
    loss_function = torch.nn.MSELoss(reduction = 'sum')

    # Setup the optimizer -- just generic ADAM
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    loss_vals = []

    # Wrap in a progress bar.
    for epoch in tqdm(range(epochs)):
        # Set the model to training mode.
        model.train()

        running_loss = 0.0

        for x, y in data_train:
            x = x.to(device)
            y = y.to(device)

            # Forward pass through the network
            output = model(x)

            # Compute loss
            loss = loss_function(output, y)
            running_loss += loss.item()

            # update model weights.
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # Set the model to eval mode and compute accuracy.
        model.eval()

        accuracys_val = list()

        for x, y in data_val:
            x = x.to(device)
            y = y.to(device)
```

```
            y_pred = model.predict(x)

        loss_vals.append(running_loss/len(data_train))

    plt.plot(loss_vals)
    plt.show()
```

```
# We write a function that takes a model, evaluate on the validation
# data set and returns the predictions

def evaluate_model(model,data_val,device):
  model.eval()
  output_vals = list()
  accuracys_val = list()
  for x, y in data_val:
            x = x.to(device)
            y = y.to(device)

            y_pred = model.predict(x)
            output_vals.append(y_pred)
            # accuracy_val = (y_pred == y).float().mean().item()
            # accuracys_val.append(accuracy_val)

  # accuracy = torch.FloatTensor(accuracys_val).mean().item()
  return output_vals
```

Problem 8 (Nothing to turn in)

Read the documentation and try to understand what a dataloader is. You can start here https://pytorch.org/docs/stable/data.html but there are many tutorials out there as well.

```
# Creating the data: Linear Regression on Linear Data
from torch.utils.data import TensorDataset, DataLoader
N = 15
X = np.random.randn(N,3)
beta = np.array([1,-1,2])
Y = np.dot(X,beta)
tensor_x = torch.Tensor(X) # transform to torch tensor
tensor_y = torch.Tensor(Y)
print('These are the labels:\n',Y)
print('These are the features:\n',X)

m = 1 # Batch size
data = TensorDataset(tensor_x,tensor_y) # create your datset
data_train = DataLoader(data,batch_size = m, shuffle = True) # create your dataloader with training data
data_val = DataLoader(data) # create your dataloader with validation data, here same as training
```

```
These are the labels:
 [-0.29679968  2.74937078 -2.5452594   5.57195048 -1.71819402  2.76167236
  1.3637214   1.10479131  1.53233299 -3.11569524  1.59814225 -1.37358099
  2.68819189  1.71582501  0.63109856]
```

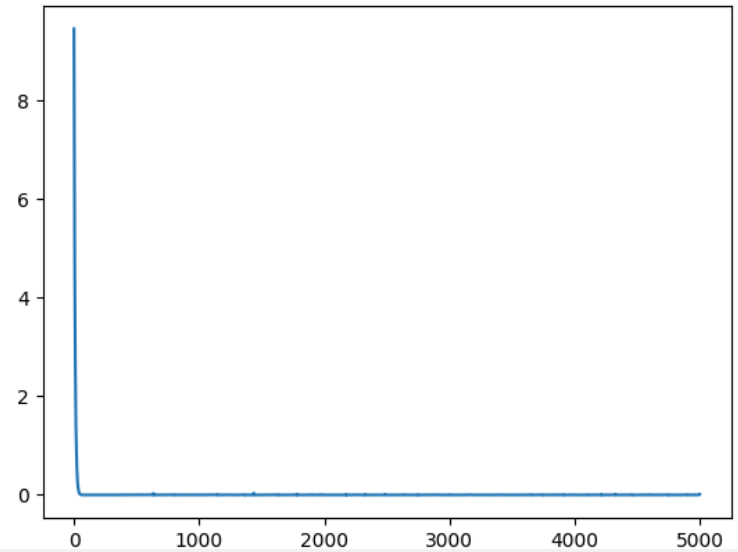
```
These are the features:
 [[ 1.56769355e-03  5.66878192e-01  1.34255409e-01]
 [ 4.94636467e-01 -1.91083812e+00  1.71948093e-01]
 [-1.39707489e+00 -1.57635058e+00 -1.36226755e+00]
 [ 1.19616988e+00 -2.64618192e+00  8.64799339e-01]
 [-1.46220706e+00  1.24925845e+00  4.96635745e-01]
 [-5.19153510e-01 -8.47419942e-01  1.21670297e+00]
 [ 1.13876120e-01 -2.29081578e+00 -5.20485248e-01]
 [ 1.40598151e+00  4.74843785e-01  8.68267899e-02]
 [ 1.79791764e-01 -1.32934552e+00  1.15978489e-02]
 [-1.07728078e-01  8.39748325e-01 -1.08410942e+00]
 [-6.03379506e-01 -1.18630921e+00  5.07606277e-01]
 [ 1.73552520e+00 -1.33428579e-01 -1.62126738e+00]
 [-3.24991860e-02 -9.80298194e-01  8.70196443e-01]
 [-6.79247950e-01  1.27265798e+00  1.83386547e+00]
 [-1.45827494e+00  3.28708533e-01  1.20904102e+00]]
```

These are the labels: [-0.29679968 2.74937078 -2.5452594 5.57195048 -1.71819402 2.76167236 1.3637214 1.10479131 1.53233299 -3.11569524 1.59814225 -1.37358099 2.68819189 1.71582501 0.63109856] These are the features: [[1.56769355e-03 5.66878192e-01 1.34255409e-01] [4.94636467e-01 -1.91083812e+00 1.71948093e-01] [-1.39707489e+00 -1.57635058e+00 -1.36226755e+00] [1.19616988e+00 -2.64618192e+00 8.64799339e-01] [-1.46220706e+00 1.24925845e+00 4.96635745e-01] [-5.19153510e-01 -8.47419942e-01 1.21670297e+00] [1.13876120e-01 -2.29081578e+00 -5.20485248e-01] [1.40598151e+00 4.74843785e-01 8.68267899e-02] [1.79791764e-01 -1.32934552e+00 1.15978489e-02] [-1.07728078e-01 8.39748325e-01 -1.08410942e+00] [-6.03379506e-01 -1.18630921e+00 5.07606277e-01] [1.73552520e+00 -1.33428579e-01 -1.62126738e+00] [-3.24991860e-02 -9.80298194e-01 8.70196443e-01] [-6.79247950e-01 1.27265798e+00 1.83386547e+00] [-1.45827494e+00 3.28708533e-01 1.20904102e+00]]

Now we train and evaluate the linear model.

```
# Define the model we wish to use, and train it.
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = LinearRegressor(3, 1)
model.to(device)

train(model, data_train, data_val, device)
```

```
100%  5000/5000 [01:34<00:00, 45.22it/s]
```



Here is some code for getting the parameters of the model.

```
# Now let's get the model parameters.
# We can see that we have succeeded in learning beta: [1,-1,2]
for name, param in model.named_parameters():
  print (name, param.data)

# If we wanted to, we could also only print the ones that we update (may be useful for more complex models)
"""
for name, param in model.named_parameters():
    if param.requires_grad:
        print (name, param.data)
"""
```

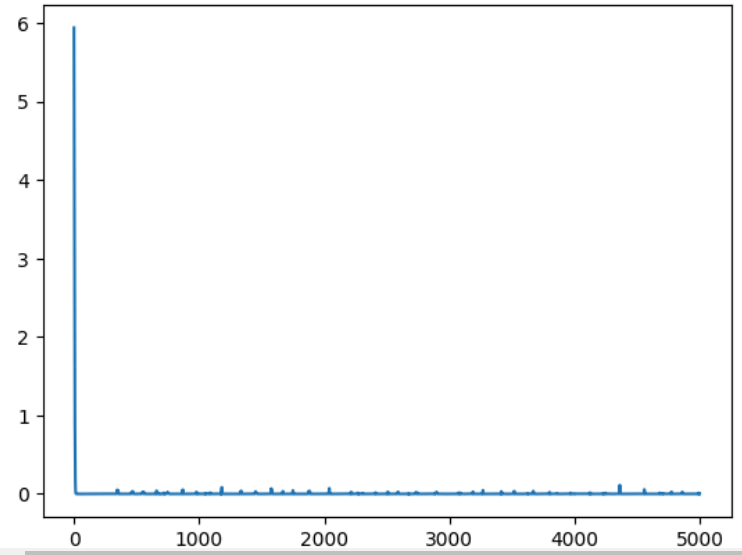
```
linear.weight tensor([[ 1.0726, -0.9719,  2.0314]])
linear.bias tensor([0.0502])
'\nfor name, param in model.named_parameters():\n    if param.requires_grad:\n        print (name, param.data)\n'
```

linear.weight tensor([[1.0726, -0.9719, 2.0314]]) linear.bias tensor([0.0502]) \nfor name, param in model.named_parameters():\n if param.requires_grad:\n print (name, param.data)\n

```
# Now let's move to our second model: the two layer linear regressor.
# We again define the model using the class we created.
# Then we train the model, as above.

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
model2 = TwoLayerLinearRegressor(3, 1)
model2.to(device)

train(model2, data_train, data_val, device)
```

```
100%                                                5000/5000  [01:51<00:00,  49.96it/s]
```




```
# Let's see how well this model agrees with the training data
output_values = evaluate_model(model2,data_val,device)
print('Ground Truth:\n',Y)
print('Model Output:\n',output_values)
```

```
>), tensor([-1.3960], grad_fn=<ViewBackward0>), tensor([2.6869], grad_fn=<ViewBackward0>), tensor([1.7226], grad_fn=<ViewBackward0>), tensor([0.6358], grad_fn=<ViewBackward0>)]
```

Ground Truth: [-0.29679968 2.74937078 -2.5452594 5.57195048 -1.71819402 2.76167236 1.3637214 1.10479131 1.53233299 -3.11569524 1.59814225 -1.37358099 2.68819189 1.71582501 0.63109856] Model Output: [tensor([-0.3030], grad_fn=), tensor([2.7421], grad_fn=), tensor([-2.5576], grad_fn=), tensor([5.5673], grad_fn=), tensor([-1.7181], grad_fn=), tensor([2.7639], grad_fn=), tensor([1.3529], grad_fn=), tensor([1.0945], grad_fn=), tensor([1.5248], grad_fn=), tensor([-3.1297], grad_fn=), tensor([1.5960], grad_fn=), tensor([-1.3960], grad_fn=), tensor([2.6869], grad_fn=), tensor([1.7226], grad_fn=), tensor([0.6358], grad_fn=)]

The XOR Data Set

We see that linear layers do not suffice.

```
"""
Here we create the simple XOR data set an a numpy array.
Then we make X and Y into tensor objects that torch uses,
and we package it into a Dataset object called data.
Then we create a DataLoader.
"""

Xxor = np.array([[0,0],[0,1],[1,0],[1,1]])
Yxor = np.array([0,1,1,0])
tensor_xxor = torch.Tensor(Xxor) # transform to torch tensor
tensor_yxor = torch.Tensor(Yxor)
print('These are the labels:\n',Yxor)
print('These are the features:\n',Xxor)

dataxor = TensorDataset(tensor_xxor,tensor_yxor) # create your datset
dataxor_train = DataLoader(dataxor) # create your dataloader with training data
dataxor_val = DataLoader(dataxor) # create your dataloader with validation data, here same as training
```

```
These are the labels:
 [0 1 1 0]
These are the features:
 [[0 0]
 [0 1]
 [1 0]
 [1 1]]
```

These are the labels: [0 1 1 0] These are the features: [[0 0] [0 1] [1 0] [1 1]]

Problem 9

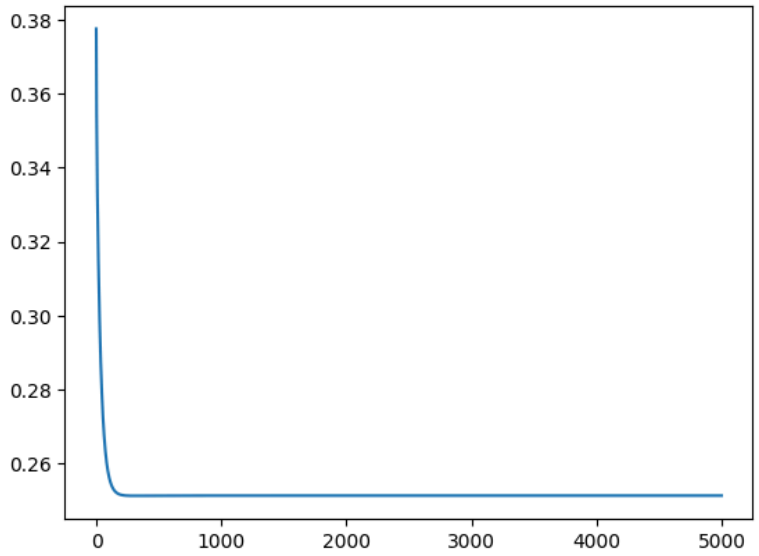
Train your linear regressor on these data. Now see how well you do, by evaluating your solution on the training data.

Print your output. Do you get the right values?

```
# Now we train a linear classifier on these data.
# We know (and can verify) that this will fail because no linear classifier can succeed

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model3 = LinearRegressor(2, 1)
model3.to(device)

# TO DO -- give a command to train your model
train(model3, dataxor_train, dataxor_val, device)

# TO DO -- give a command to evaluate your model
output_values = evaluate_model(model3, dataxor_val, device)

# TO DO -- print the ground truth, and then also print what your model predicts for the 4 points in the training set.
```

```
print('Ground Truth:\n', Yxor)
print('Model Output:\n', output_values)
```

```
100%                                                  5000/5000  [00:25<00:00,  215.43it/s]
```




```
Ground Truth:
 [0 1 1 0]
Model Output:
```

Ground Truth: [0 1 1 0] Model Output: [tensor([0.5012], grad_fn=), tensor([0.5006], grad_fn=), tensor([0.5005], grad_fn=), tensor([0.4999], grad_fn=)]

Problem 10

Now repeat this, but using both versions of your non-linear two-layer model. Thus: train both versions of your non-linear two layer models, and evaluate them on the data.

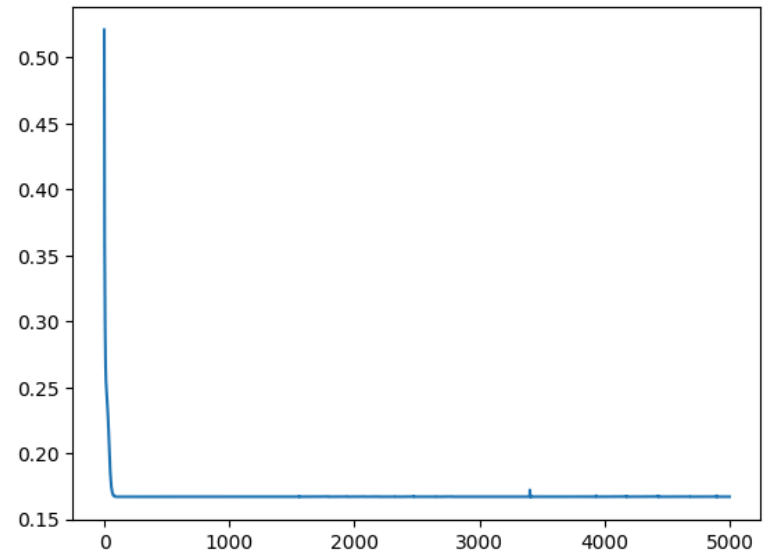
Remember the values we got in class. We saw that there is a solution -- i.e., a setting for the parameters of the nonlinear model -- so that the network outputs the XOR outputs.

If you got the right values, try a different random seed. If you didn't get the right values, try changing the random seed to see if you get something different.

```
torch.manual_seed(2)
np.random.seed(2)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model4 = TwoLayerNonLinearRegressor(2, 1)
model4.to(device)
```

```
train(model4, dataxor_train, dataxor_val, device)

output_values = evaluate_model(model4, dataxor_val, device)

print('Ground Truth:\n', Yxor)
print('Model Output:\n', output_values)
```

```
100%                                                  5000/5000  [00:32<00:00,  176.06it/s]
```

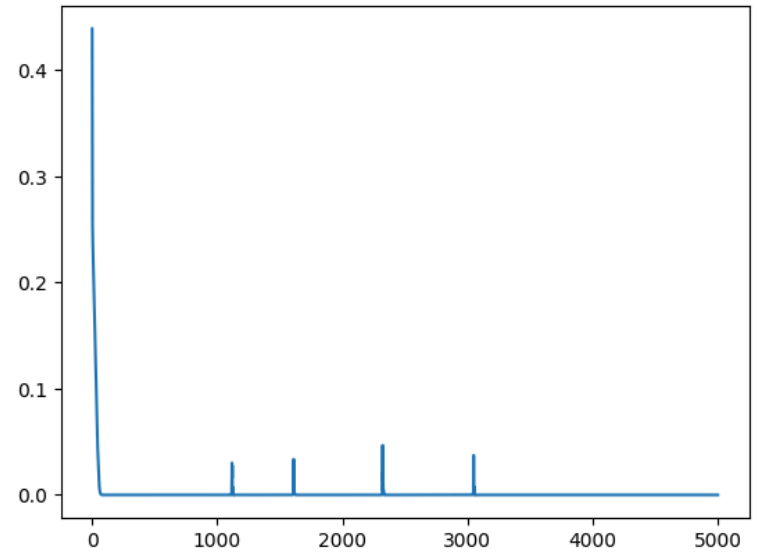



```
Ground Truth:
 [0 1 1 0]
Model Output:
```

Ground Truth: [0 1 1 0] Model Output: [tensor([0.3336], grad_fn=), tensor([0.3336], grad_fn=), tensor([1.0004], grad_fn=), tensor([0.3336], grad_fn=)]

```
torch.manual_seed(222)
np.random.seed(222)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model5 = TwoLayerNonLinearRegressor2(2, 1)
model5.to(device)

train(model5, dataxor_train, dataxor_val, device)

output_values = evaluate_model(model5, dataxor_val, device)

print('Ground Truth:\n', Yxor)
print('Model Output:\n', output_values)
```

```
100%  5000/5000 [00:31<00:00, 172.94it/s]
```




```
Ground Truth:
 [0 1 1 0]
Model Output:
```

Problem 11

If you did the above correctly, you will see that the values depend quite a bit on the random seed. In fact, you probably noticed that most random seeds **don't seem to work**.

What is happening here is that we are getting stuck in a local minimum, and our gradient-based method used in the training loop cannot escape from it.

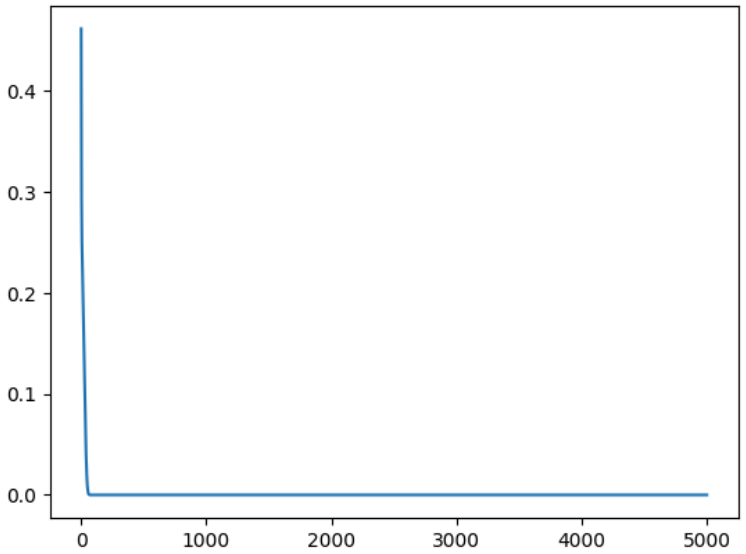
Modify the definition of your non-linear two layer network so that the first layer of neurons (the hidden layer) has more than 2 (say, 10 or 15) neurons. Note that this means your network has more parameters.

Then run it again with different seeds on the XOR training data. Is it easier (i.e., easier to find a random seed that works)?

```
for i in range(3):
  torch.manual_seed(i)
  np.random.seed(i)
  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

  model6 = TwoLayerNonLinearRegressor2(2, 1)
  model6.to(device)

  train(model6, dataxor_train, dataxor_val, device)

  output_values = evaluate_model(model6, dataxor_val, device)
```

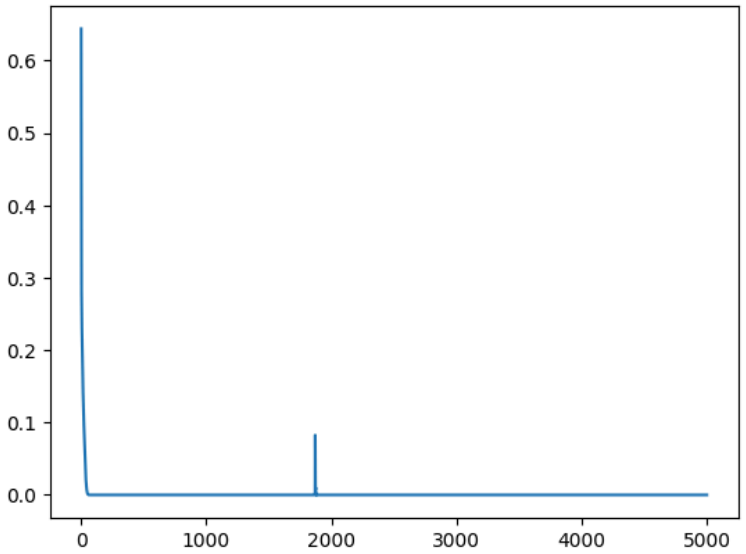
```
print('Ground Truth:\n', Yxor)
print('Model Output:\n', output_values)
```

100% 5000/5000 [00:32<00:00, 169.80it/s]



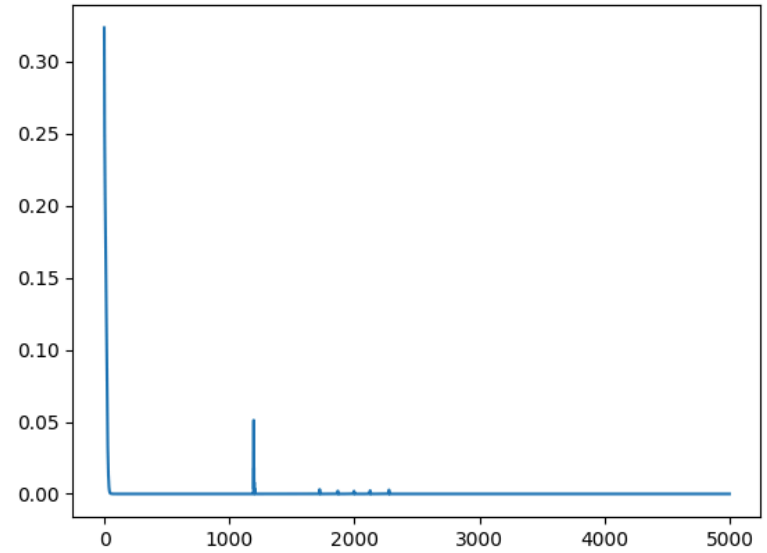

```
Ground Truth:
 [0 1 1 0]
Model Output:
 [tensor([0.], grad_fn=<ViewBackward0>), tensor([1.], grad_fn=<ViewBackward0>), tensor([1.], grad_fn=<ViewBackward0>), tensor([0.], grad_fn=<ViewBackward0>)]
```

100% 5000/5000 [00:32<00:00, 166.23it/s]




```
Ground Truth:
 [0 1 1 0]
Model Output:
 [tensor([0.], grad_fn=<ViewBackward0>), tensor([1.], grad_fn=<ViewBackward0>), tensor([1.], grad_fn=<ViewBackward0>), tensor([0.], grad_fn=<ViewBackward0>)]
```

100% 5000/5000 [00:32<00:00, 166.43it/s]




```
Ground Truth:
 [0 1 1 0]
Model Output:
```

Graph 1: Ground Truth: [0 1 1 0] Model Output: [tensor([0.], grad_fn=), tensor([1.], grad_fn=), tensor([1.], grad_fn=), tensor([0.], grad_fn=)]

Graph 2: Ground Truth: [0 1 1 0] Model Output: [tensor([0.], grad_fn=), tensor([1.], grad_fn=), tensor([1.], grad_fn=), tensor([0.], grad_fn=)]

Graph 3: Ground Truth: [0 1 1 0] Model Output: [tensor([6.4460e-44], grad_fn=), tensor([1.], grad_fn=), tensor([1.], grad_fn=), tensor([6.4460e-44], grad_fn=)]

Much easier to find a seed that works with hidden layer that has 15 neurons.

Problem 12

Print the parameters of one of your non-linear models. Thus, you should print: 4 weights plus 2 bias values for the first layer, and then 2 weights plus 1 bias value for the second: 9 parameters in total.

```
# Now let's get the model parameters.
for name, param in model4.named_parameters():
  print (name, param.data)

# If we wanted to, we could also only print the ones that we update (may be useful for more complex models)
"""
for name, param in model.named_parameters():
    if param.requires_grad:
        print (name, param.data)
"""
```

```
linear.0.weight tensor([[ 1.1160, -0.9760],
        [-0.0351, -0.1309]])
linear.0.bias tensor([-0.2352, -0.0309])
linear.2.weight tensor([[ 0.7570, -0.4065]])
linear.2.bias tensor([0.3336])
```

linear.0.weight tensor([[1.1160, -0.9760], [-0.0351, -0.1309]]) linear.0.bias tensor([-0.2352, -0.0309]) linear.2.weight tensor([[0.7570, -0.4065]]) linear.2.bias tensor([0.3336]) \nfor name, param in model.named_parameters():\n if param.requires_grad:\n print (name, param.data)\n

```
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
Filler for PDF
```

Filler for PDF Filler for PDF