Lab 5 Questions

October 16, 2023

1 Data Science Lab: Lab 5

Due: Sunday October 22nd 2023

Submit:

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

2 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.

[ ]:
```
import torch
import torch.nn as nn
import numpy as np
import time
from tqdm.notebook import tqdm
```

2.1 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)
```

2.2 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 = nn.Sequential(nn.Linear(input_dim,input_dim),
                                 nn.Linear(input_dim,output_dim))# TO DO:␣
↪complete this equality, using torch.nn.Sequential

    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)
```

2.3 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 = nn.Linear(input_dim, input_dim)# TO DO
        self.fc2 = nn.Linear(input_dim, output_dim)# TO DO

    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)
```

2.4 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 = nn.Sequential(nn.Linear(input_dim, input_dim),
                                    nn.ReLU(),
                                    nn.Linear(input_dim, output_dim)
                                    )# TO DO

    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)
```

2.5 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 = nn.Linear(input_dim, input_dim)# TO DO
        self.relu = nn.ReLU() # TO DO
        self.fc2 = nn.Linear(input_dim, output_dim)# TO DO
```

```
    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)
```

2.6 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.

[ ]:
```
# 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)

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

        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)

            # 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)
```

[ ]:
```
# 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
```

2.7 Problem 6 (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:
 [ 0.36314795  5.60078547  4.40069214  0.30343396 -2.89207202 -5.35863321
 -3.92311866 -1.39700225 -2.44353516 -1.72176015  0.0314971   5.06068796
 -2.82758186 -3.18490464 -1.71864922]
These are the features:
 [[ 2.33664214  0.64912637 -0.66218391]
 [ 1.50075507  0.24257609  2.17130325]
 [ 0.46262391 -0.98139259  1.47833782]
 [-0.4676801   0.48493941  0.62802674]
 [ 0.39758312  0.30652109 -1.49156702]
```

```
[-1.37407248  0.43995766 -1.77230153]
[ 0.93726631  0.50200723 -2.17918887]
[-1.1554747  -0.77143594 -0.50648174]
[-0.21931039  0.7472172  -0.73850378]
[-0.15065868 -0.99546051 -1.28328099]
[ 1.06684282  0.36965284 -0.33284644]
[ 1.31793412 -1.60430995  1.06922194]
[-1.07522569 -0.23618969 -0.99427293]
[-0.37998457  0.26506811 -1.26992598]
[ 0.44413765 -0.75819435 -1.46049061]]
```

2.8 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)
```

```
  0%|          | 0/5000 [00:00<?, ?it/s]
```

2.9 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([[ 1.0000, -1.0000,  2.0000]])
linear.bias tensor([-8.1679e-09])
```

[ ]:
```
'\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)
```

```
  0%|          | 0/5000 [00:00<?, ?it/s]
```

[ ]:
```
# Let's see how well this model agrees with the training data
output_values = evaluate_model(model2,data_val,device)
list_output_values = np.array([value.item() for value in output_values])
print('Ground Truth:\n',Y)
print('Model Output:\n',list_output_values)
```

```
Ground Truth:
 [ 0.36314795  5.60078547  4.40069214  0.30343396 -2.89207202 -5.35863321
 -3.92311866 -1.39700225 -2.44353516 -1.72176015  0.0314971   5.06068796
 -2.82758186 -3.18490464 -1.71864922]
Model Output:
 [ 0.36319625  5.60078526  4.40066957  0.30352002 -2.89197922 -5.3584857
 -3.9230175  -1.39694047 -2.44341898 -1.7217201   0.03155695  5.06062174
 -2.82748795 -3.18479872 -1.71860921]
```

3 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]]
```

3.1 Problem 7

Train your linear regressor on these data. Now see how well you do, by evaluating your solution on the training data. Remember the values we got in class.

Your output here should be predicted values for each of the 4 points in our XOR data set.

[ ]:
```
# 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 = TwoLayerLinearRegressor(2, 1) # TO DO
model3.to(device)

train(model3, dataxor_train, dataxor_val, device)# TO DO -- give a command to␣
 ↪train your model
output_values = evaluate_model(model3, dataxor_train, device)# TO DO -- give a␣
 ↪command to evaluate your model
numpy_output_values = [x.item() for x in output_values]
print(f'Ground Truth values : {tensor_yxor}')# TO DO -- print the ground truth,␣
 ↪and then also print what your model predicts for the 4 points in the␣
 ↪training set.
print(f'Model Predictions values : {numpy_output_values}')
```

```
  0%|          | 0/5000 [00:00<?, ?it/s]
Ground Truth values : tensor([0., 1., 1., 0.])
Model Predictions values : [0.5004987120628357, 0.5004987120628357,
0.5004987120628357, 0.5004987120628357]
```

3.2 Problem 8

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.

If you did this correctly, the values you compute should equal (approximately) the values of the XOR function.

[ ]:
```
# Running Sequential Model
model4 = TwoLayerNonLinearRegressor(2,1)
```

```
model4.to(device)

train(model4, dataxor_train, dataxor_val, device)
output_values_non = evaluate_model(model4, dataxor_train, device)
numpy_output_values_non = [x.item() for x in output_values_non]
print(f'Ground Truth values : {tensor_yxor}')
print(f'Model Predictions values : {numpy_output_values_non}')
```

```
  0%|          | 0/5000 [00:00<?, ?it/s]
Ground Truth values : tensor([0., 1., 1., 0.])
Model Predictions values : [-1.401298464324817e-45, 1.0, 1.0,
-1.401298464324817e-45]
```

[ ]:
```
# Running Non-Sequential Model
model5 = TwoLayerNonLinearRegressor2(2,1)
model5.to(device)

train(model5, dataxor_train, dataxor_val, device)
output_values_non2 = evaluate_model(model5, dataxor_train, device)
numpy_output_values_non2 = [x.item() for x in output_values_non2]
print(f'Ground Truth values : {tensor_yxor}')
print(f'Model Predictions values : {numpy_output_values_non2}')
```

```
  0%|          | 0/5000 [00:00<?, ?it/s]
Ground Truth values : tensor([0., 1., 1., 0.])
Model Predictions values : [-3.5032461608120427e-44, 1.0, 1.0,
-3.5032461608120427e-44]
```

3.3 Problem 9

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.

[ ]:
```
for name, param in model5.named_parameters():
  print (name, param.data)
```

```
fc1.weight tensor([[1.0344, 1.0344],
        [0.8445, 0.8866]])
fc1.bias tensor([-0.0016, -0.9038])
fc2.weight tensor([[ 0.9682, -2.4194]])
fc2.bias tensor([-3.5032e-44])
```