Lecture_12_pytorch_sgd_neural_network_code

October 5, 2024

PyTorch: Stochastic Gradient Descent and Neural Networks

[6]:
```python
import torch  # now import the tensorflow module
from torch import nn
import numpy as np
```

0.1 1 Create Linear Regression Model

[7]:
```python
from torch import nn

class MyLinearRegressionModel(nn.Module):
    def __init__(self,d): # d is the dimension of the input
        super(MyLinearRegressionModel,self).__init__()   # call the init␣
 ↪function of super class
        # we usually create variables for all our model parameters (w and b in␣
 ↪our case) in __init__ and give them initial values.
        # need to create them as nn.Parameter so that the model knows it is an␣
 ↪parameter that needs to be trained
        self.w = nn.Parameter(torch.zeros(1,d, dtype=torch.float))
        self.b = nn.Parameter(torch.zeros(1,dtype=torch.float))
    def forward(self,x):
        # The main purpose of the forward function is to specify given input x,␣
 ↪how the output is calculated.
        return torch.inner(x,self.w) + self.b

# Let's check out our model

mymodel = MyLinearRegressionModel(1) # creating a model instance with input␣
 ↪dimension 1
print(mymodel.w)
print(mymodel.b)
```

```
Parameter containing:
tensor([[0.]], requires_grad=True)
Parameter containing:
tensor([0.], requires_grad=True)
```

0.2 2 Creating Dataset and DataLoader

The torch Dataset is similar to tf.data.Dataset. The general way to create a dataset is through subclassing the Dataset class and define the `__len__()` and `__getitem__()` method.

[8]:
```
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader

x = torch.arange(0,10,.1,dtype=torch.float)
x = x[:,None]
y = x*3+torch.randn(x.shape)

# Example of dataset
class MyDataset(Dataset):
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        return (self.x[idx],self.y[idx])
```

[9]:
```
mydataset = MyDataset(x,y)
for item in mydataset:
    print(item)
```

```
(tensor([0.]), tensor([1.2457]))
(tensor([0.1000]), tensor([0.1175]))
(tensor([0.2000]), tensor([0.9422]))
(tensor([0.3000]), tensor([3.0011]))
(tensor([0.4000]), tensor([-0.4568]))
(tensor([0.5000]), tensor([1.1579]))
(tensor([0.6000]), tensor([1.8405]))
(tensor([0.7000]), tensor([2.1727]))
(tensor([0.8000]), tensor([3.3919]))
(tensor([0.9000]), tensor([2.3869]))
(tensor([1.]), tensor([3.3897]))
(tensor([1.1000]), tensor([1.9761]))
(tensor([1.2000]), tensor([2.3848]))
(tensor([1.3000]), tensor([2.9465]))
(tensor([1.4000]), tensor([4.2778]))
(tensor([1.5000]), tensor([6.2860]))
(tensor([1.6000]), tensor([4.1121]))
(tensor([1.7000]), tensor([4.3621]))
(tensor([1.8000]), tensor([4.2256]))
(tensor([1.9000]), tensor([6.0039]))
```

```
(tensor([2.]), tensor([6.5152]))
(tensor([2.1000]), tensor([6.5941]))
(tensor([2.2000]), tensor([5.7994]))
(tensor([2.3000]), tensor([7.4983]))
(tensor([2.4000]), tensor([7.3677]))
(tensor([2.5000]), tensor([9.1190]))
(tensor([2.6000]), tensor([10.3882]))
(tensor([2.7000]), tensor([10.1740]))
(tensor([2.8000]), tensor([7.5970]))
(tensor([2.9000]), tensor([8.4861]))
(tensor([3.]), tensor([10.1763]))
(tensor([3.1000]), tensor([10.7329]))
(tensor([3.2000]), tensor([10.9123]))
(tensor([3.3000]), tensor([11.2244]))
(tensor([3.4000]), tensor([9.1344]))
(tensor([3.5000]), tensor([11.7839]))
(tensor([3.6000]), tensor([12.0660]))
(tensor([3.7000]), tensor([9.5300]))
(tensor([3.8000]), tensor([10.1838]))
(tensor([3.9000]), tensor([10.3585]))
(tensor([4.]), tensor([14.0892]))
(tensor([4.1000]), tensor([12.0775]))
(tensor([4.2000]), tensor([13.2745]))
(tensor([4.3000]), tensor([11.9290]))
(tensor([4.4000]), tensor([13.7796]))
(tensor([4.5000]), tensor([12.6947]))
(tensor([4.6000]), tensor([12.9104]))
(tensor([4.7000]), tensor([13.6831]))
(tensor([4.8000]), tensor([15.0263]))
(tensor([4.9000]), tensor([13.6702]))
(tensor([5.]), tensor([12.3192]))
(tensor([5.1000]), tensor([13.4018]))
(tensor([5.2000]), tensor([17.6889]))
(tensor([5.3000]), tensor([17.4949]))
(tensor([5.4000]), tensor([16.4059]))
(tensor([5.5000]), tensor([17.4463]))
(tensor([5.6000]), tensor([17.6586]))
(tensor([5.7000]), tensor([17.8863]))
(tensor([5.8000]), tensor([16.4887]))
(tensor([5.9000]), tensor([17.2960]))
(tensor([6.]), tensor([20.5103]))
(tensor([6.1000]), tensor([17.7755]))
(tensor([6.2000]), tensor([18.8127]))
(tensor([6.3000]), tensor([18.1241]))
(tensor([6.4000]), tensor([19.7516]))
(tensor([6.5000]), tensor([18.7486]))
(tensor([6.6000]), tensor([19.7492]))
(tensor([6.7000]), tensor([21.8706]))
```

```
(tensor([6.8000]), tensor([22.1195]))
(tensor([6.9000]), tensor([20.6584]))
(tensor([7.]), tensor([20.3775]))
(tensor([7.1000]), tensor([22.7021]))
(tensor([7.2000]), tensor([22.3048]))
(tensor([7.3000]), tensor([24.2129]))
(tensor([7.4000]), tensor([22.2383]))
(tensor([7.5000]), tensor([24.3921]))
(tensor([7.6000]), tensor([22.1790]))
(tensor([7.7000]), tensor([23.0269]))
(tensor([7.8000]), tensor([21.7757]))
(tensor([7.9000]), tensor([24.0109]))
(tensor([8.]), tensor([23.4366]))
(tensor([8.1000]), tensor([23.8894]))
(tensor([8.2000]), tensor([25.3090]))
(tensor([8.3000]), tensor([24.0570]))
(tensor([8.4000]), tensor([24.3680]))
(tensor([8.5000]), tensor([27.4692]))
(tensor([8.6000]), tensor([25.1863]))
(tensor([8.7000]), tensor([24.7462]))
(tensor([8.8000]), tensor([28.3490]))
(tensor([8.9000]), tensor([24.9879]))
(tensor([9.]), tensor([26.0128]))
(tensor([9.1000]), tensor([26.8871]))
(tensor([9.2000]), tensor([25.4319]))
(tensor([9.3000]), tensor([25.5913]))
(tensor([9.4000]), tensor([27.4982]))
(tensor([9.5000]), tensor([29.2889]))
(tensor([9.6000]), tensor([29.3427]))
(tensor([9.7000]), tensor([28.7557]))
(tensor([9.8000]), tensor([29.3267]))
(tensor([9.9000]), tensor([28.8826]))
```

[10]:
```
mydataloader = DataLoader(mydataset, batch_size = 4, shuffle = True)
for item in mydataloader:
    print(item)
```

```
[tensor([[9.4000],
        [1.9000],
        [0.3000],
        [3.8000]]), tensor([[27.4982],
        [ 6.0039],
        [ 3.0011],
        [10.1838]])]
[tensor([[1.4000],
        [4.5000],
        [0.4000],
        [1.1000]]), tensor([[ 4.2778],
```

```
         [12.6947],
         [-0.4568],
         [ 1.9761]])]
[tensor([[4.8000],
         [2.0000],
         [9.8000],
         [6.9000]]), tensor([[15.0263],
         [ 6.5152],
         [29.3267],
         [20.6584]])]
[tensor([[5.8000],
         [9.0000],
         [3.1000],
         [7.9000]]), tensor([[16.4887],
         [26.0128],
         [10.7329],
         [24.0109]])]
[tensor([[7.4000],
         [6.6000],
         [3.2000],
         [1.8000]]), tensor([[22.2383],
         [19.7492],
         [10.9123],
         [ 4.2256]])]
[tensor([[8.6000],
         [2.1000],
         [0.7000],
         [7.3000]]), tensor([[25.1863],
         [ 6.5941],
         [ 2.1727],
         [24.2129]])]
[tensor([[8.5000],
         [8.3000],
         [3.0000],
         [9.5000]]), tensor([[27.4692],
         [24.0570],
         [10.1763],
         [29.2889]])]
[tensor([[0.5000],
         [7.2000],
         [7.5000],
         [2.3000]]), tensor([[ 1.1579],
         [22.3048],
         [24.3921],
         [ 7.4983]])]
[tensor([[8.2000],
         [4.4000],
         [0.1000],
```

```
        [8.0000]]), tensor([[25.3090],
        [13.7796],
        [ 0.1175],
        [23.4366]])]
[tensor([[6.5000],
        [3.9000],
        [2.9000],
        [9.2000]]), tensor([[18.7486],
        [10.3585],
        [ 8.4861],
        [25.4319]])]
[tensor([[3.4000],
        [3.6000],
        [8.8000],
        [6.8000]]), tensor([[ 9.1344],
        [12.0660],
        [28.3490],
        [22.1195]])]
[tensor([[6.0000],
        [5.2000],
        [4.0000],
        [7.1000]]), tensor([[20.5103],
        [17.6889],
        [14.0892],
        [22.7021]])]
[tensor([[3.5000],
        [1.7000],
        [5.4000],
        [4.2000]]), tensor([[11.7839],
        [ 4.3621],
        [16.4059],
        [13.2745]])]
[tensor([[5.6000],
        [2.7000],
        [7.0000],
        [0.9000]]), tensor([[17.6586],
        [10.1740],
        [20.3775],
        [ 2.3869]])]
[tensor([[5.1000],
        [7.7000],
        [4.7000],
        [9.3000]]), tensor([[13.4018],
        [23.0269],
        [13.6831],
        [25.5913]])]
[tensor([[5.0000],
        [4.1000],
```

```
        [9.9000],
        [6.7000]]), tensor([[12.3192],
        [12.0775],
        [28.8826],
        [21.8706]])]
[tensor([[5.5000],
        [6.1000],
        [0.2000],
        [3.3000]]), tensor([[17.4463],
        [17.7755],
        [ 0.9422],
        [11.2244]])]
[tensor([[9.7000],
        [8.4000],
        [7.6000],
        [2.2000]]), tensor([[28.7557],
        [24.3680],
        [22.1790],
        [ 5.7994]])]
[tensor([[2.4000],
        [4.6000],
        [3.7000],
        [9.6000]]), tensor([[ 7.3677],
        [12.9104],
        [ 9.5300],
        [29.3427]])]
[tensor([[8.7000],
        [1.0000],
        [7.8000],
        [0.0000]]), tensor([[24.7462],
        [ 3.3897],
        [21.7757],
        [ 1.2457]])]
[tensor([[1.5000],
        [4.9000],
        [4.3000],
        [1.6000]]), tensor([[ 6.2860],
        [13.6702],
        [11.9290],
        [ 4.1121]])]
[tensor([[1.2000],
        [5.3000],
        [6.3000],
        [2.8000]]), tensor([[ 2.3848],
        [17.4949],
        [18.1241],
        [ 7.5970]])]
[tensor([[5.9000],
```

```
        [6.2000],
        [8.9000],
        [2.6000]]), tensor([[17.2960],
        [18.8127],
        [24.9879],
        [10.3882]])]
[tensor([[5.7000],
        [6.4000],
        [2.5000],
        [1.3000]]), tensor([[17.8863],
        [19.7516],
        [ 9.1190],
        [ 2.9465]])]
[tensor([[0.8000],
        [8.1000],
        [0.6000],
        [9.1000]]), tensor([[ 3.3919],
        [23.8894],
        [ 1.8405],
        [26.8871]])]
```

0.3 3 Stochastic Gradient Descent

The main difference between SGD and GD is that time, we use a batch of data of compute the gradient, as opposed to the full training dataset.

Typically, the SGD is implemented as a nested for-loop, where the outer loop go through a number of epochs, where the inner loop go through all batches in the data set, and the batches can be obtained from the DataLoader.

[11]:
```
import io
import imageio
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

mymodel = MyLinearRegressionModel(1) # creating a model instance with input␣
 ↪dimension 1
lr = 1e-3
batch_size = 2

optimizer = torch.optim.SGD(mymodel.parameters(), lr = lr) # this line creates␣
 ↪a optimizer, and we tell optimizer we are optimizing the parameters in␣
 ↪mymodel
mydataloader = DataLoader(mydataset, batch_size = batch_size, shuffle = True)


frames = []
```

```python
losses = []
losses_all = []
N_epochs = 6

gd_steps = 0
N_batches = len(mydataloader)
for epoch in range(N_epochs):
    batch_loss = []
    for batch_id, (x_batch, y_batch) in enumerate(mydataloader):
        gd_steps+=1
        # pass input data to get the prediction outputs by the current model
        prediction = mymodel(x_batch)

        # compare prediction and the actual output and compute the loss
        loss = torch.mean((prediction - y_batch)**2)

        # compute the gradient
        optimizer.zero_grad()
        loss.backward()

        # update parameters
        optimizer.step()


        # Generate visualization plots
        fig, ax = plt.subplots(nrows = 1, ncols = 3)
        canvas = FigureCanvas(fig)
        ax[0].plot(x,y,'ro')
        prediction_full = mymodel(x)
        ax[0].plot(x,prediction_full.detach(),linewidth = 2)
        ax[0].legend(['data','prediction of mymodel'],loc = 'upper left')
        ax[0].set_title(f"Batch size = {batch_size}, Learning rate = {lr},␣
 ↪Epoch #{epoch}, Batch #{batch_id}", fontsize = 20)
        ax[0].set_xlim((0,10))
        ax[0].set_ylim((0,30))
        losses_all.append(loss.detach().numpy())
        ax[1].plot(np.arange(gd_steps),np.array(losses_all).
 ↪squeeze(),linewidth=2 )
        ax[1].set_xlim((0,(N_epochs+1)*(N_batches)))
        ax[1].set_ylim((0,30))
        ax[1].set_title("Training loss", fontsize = 20)
        ax[1].set_xlabel("# of SGD Iterations", fontsize = 20)

        batch_loss.append(loss.detach().numpy())
        if epoch>0:
            ax[2].plot(np.arange(epoch),np.array(losses).squeeze(),linewidth=2 )
        else:
```

```
            ax[2].plot(np.arange(epoch+1),np.mean(np.array(batch_loss).
 ↪squeeze()),linewidth=2 )

        ax[2].set_xlim((0,N_epochs-1))
        ax[2].set_ylim((0,30))
        ax[2].set_title("Training loss", fontsize = 20)
        ax[2].set_xlabel("# of Epochs", fontsize = 20)

        fig.set_size_inches(27,9)
        canvas.draw()       # draw the canvas, cache the renderer

        image = np.frombuffer(canvas.tostring_rgb(), dtype='uint8')

        image = image.reshape(fig.canvas.get_width_height()[::-1] + (3,))

        frames.append(image)
        plt.close(fig)
    losses.append(np.mean(np.array(batch_loss)))

print("Saving GIF file")
with imageio.get_writer("SGD.gif", mode="I") as writer:
    for frame in frames:
        writer.append_data(frame)
```

```
Saving GIF file
```

0.4 4. Simple Neural Nework with PyTorch

0.5 4.1 Equivalent Way to Build Linear Regression Model with Built-in Layers

In our previous linear model, we declared all the parameters (`w` and `b`) in the model and manually coded all the mathematical operations.

PyTorch provides many built-in layers such that you don't need to do the above yourself. For linear regression, you can just use `nn.Linear(input_dim, output_dim)` to create a linear function.

[ ]:
```
import torch
from torch import nn
from torch.utils.data import Dataset,DataLoader


class MyLinearRegressionModel_withBuiltinLayers(nn.Module):
    def __init__(self,d):
        super(MyLinearRegressionModel_withBuiltinLayers,self).__init__()
        self.linear_layer = nn.Linear(d,1) # define a linear layer and store it␣
 ↪as an attribute
    def forward(self,x):
        return self.linear_layer(x) # use the linear layer to give the output
```

You can print a model instance, which will show what are the layers inside.

```
[ ]: mymodel_lr = MyLinearRegressionModel_withBuiltinLayers(1)

     print(mymodel_lr)
```

```
MyLinearRegressionModel_withBuiltinLayers(
  (linear_layer): Linear(in_features=1, out_features=1, bias=True)
)
```

You can also show all the parameters within the `linear_layer`. It has a `weight` parameter and `bias` parameter, which is the same as the `w` and the `b` parameter we defined before.

```
[ ]: for name,param in mymodel_lr.state_dict().items():
         print(name, param)
```

```
linear_layer.weight tensor([[-0.9548]])
linear_layer.bias tensor([0.8108])
```

0.6 4.2 Build Neural Network Model

With the built-in layers, it is now convenient to create a neural network, which is a connection of linear layers with nonlinear activation functions. To create neural networks in PyTorch, we need to ‘’connect” `nn.Linear()` and nonlinear activation functions (`nn.ReLU()` if using ReLU as activation) together. We can do this conveniently using the `nn.Sequential()`.

```
[ ]: from torch import nn

     # Example: using Sequential in Pytorch

     class myMultiLayerPerceptron(nn.Module):
         def __init__(self,input_dim,output_dim):
             super().__init__()
             self.sequential = nn.Sequential(  # here we stack multiple layers␣
      ↪together
                 nn.Linear(input_dim,20),
                 nn.ReLU(),
                 nn.Linear(20,20),
                 nn.ReLU(),
                 nn.Linear(20,20),
                 nn.ReLU(),
                 nn.Linear(20,20),
                 nn.ReLU(),
                 nn.Linear(20,output_dim)
             )
         def forward(self,x):
             y = self.sequential(x)
             return y
```

```python
# Let's check out our model

mymodel = myMultiLayerPerceptron(1,1) # creating a model instance with input␣
 ↪dimension 1 and output dimension 1

print(mymodel)
```

```
myMultiLayerPerceptron(
  (sequential): Sequential(
    (0): Linear(in_features=1, out_features=20, bias=True)
    (1): ReLU()
    (2): Linear(in_features=20, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=20, bias=True)
    (5): ReLU()
    (6): Linear(in_features=20, out_features=20, bias=True)
    (7): ReLU()
    (8): Linear(in_features=20, out_features=1, bias=True)
  )
)
```

[ ]:
```python
for name,param in mymodel.state_dict().items():
    print(name,param)
```

```
sequential.0.weight tensor([[-0.3058],
        [-0.8540],
        [-0.0853],
        [ 0.7444],
        [-0.9963],
        [ 0.5363],
        [-0.8776],
        [ 0.3161],
        [ 0.7581],
        [-0.7360],
        [ 0.3863],
        [-0.7745],
        [ 0.1953],
        [-0.9288],
        [ 0.2089],
        [-0.6512],
        [-0.8162],
        [ 0.3498],
        [ 0.3428],
        [ 0.5378]])
sequential.0.bias tensor([-0.9314,  0.7788,  0.0453, -0.7418, -0.5442,  0.6527,
0.4688,  0.7857,
        -0.1453,  0.0692,  0.2235, -0.0609,  0.3079,  0.8309, -0.7672,  0.6019,
```

```
        -0.0597, -0.2741,  0.5761,  0.5241])
sequential.2.weight tensor([[ 1.9191e-01,  5.6867e-03,  1.3620e-01,  1.1376e-01,
-9.6147e-02,
          9.8625e-02,  7.5702e-02,  4.0915e-02, -1.6518e-01,  1.1429e-01,
         -1.8421e-01, -1.9916e-01, -2.3071e-02,  2.1642e-01, -6.7595e-02,
          5.2481e-02, -1.6609e-01, -1.5764e-01,  1.2740e-01,  1.5464e-01],
        [ 1.1961e-01,  1.4661e-02, -1.3006e-01,  9.0424e-02, -2.1995e-01,
         -1.2044e-02, -1.9587e-01, -9.8699e-02,  7.6424e-02,  7.7618e-03,
          8.5139e-02,  1.3060e-01, -1.6035e-01, -1.9439e-01, -2.2419e-02,
          1.3700e-01, -1.2481e-01, -1.2958e-01,  9.7386e-02,  8.7938e-02],
        [-1.1357e-01,  1.9497e-01,  2.1165e-01, -1.4686e-01,  1.5342e-01,
         -2.1191e-02, -1.1344e-01,  1.8365e-02, -1.0079e-01, -1.0308e-02,
         -5.6763e-02,  1.0994e-01,  1.8245e-01,  3.6436e-02, -1.8524e-01,
          1.5238e-01, -4.8559e-02,  9.0425e-02, -6.5051e-02,  1.5471e-01],
        [ 1.9415e-01, -6.6969e-02, -1.4559e-01, -7.9033e-02,  1.7294e-01,
         -1.1231e-01, -5.2746e-02, -5.3814e-02,  2.0702e-01,  1.2005e-01,
         -5.1117e-02,  3.0610e-02,  1.8194e-02, -1.4372e-01,  7.8048e-03,
          1.0142e-02,  2.3640e-02, -1.6641e-01, -1.5090e-01,  4.0743e-02],
        [ 7.8691e-02,  1.7885e-01,  1.2236e-01,  1.2346e-01,  5.8927e-02,
         -9.1852e-02, -2.2255e-01, -1.5256e-01,  1.8848e-01, -9.4315e-02,
         -8.8562e-02,  5.9533e-02,  1.5741e-01, -1.1972e-02, -6.0016e-02,
         -1.5898e-01,  4.8084e-02, -3.0885e-02, -6.3576e-02,  8.1465e-02],
        [ 2.2171e-01,  2.1654e-02,  1.6414e-01, -1.3928e-01,  6.9190e-02,
         -1.7664e-01, -7.5303e-02,  1.1568e-01, -1.5362e-01, -1.0044e-01,
         -1.9771e-01, -1.6679e-02, -1.7510e-01,  1.8491e-01,  6.4341e-02,
          1.9931e-01, -1.1443e-01,  8.5660e-02,  7.4657e-02, -5.0080e-02],
        [-1.1380e-01,  1.5878e-01, -5.2743e-03, -2.2201e-01, -1.0080e-01,
         -4.1863e-02,  1.5374e-01,  9.0688e-02, -1.3664e-01, -1.9481e-01,
          4.6127e-02,  1.5404e-01, -3.2956e-02, -4.6967e-02, -1.9858e-01,
         -3.8873e-02, -9.2804e-02, -2.0552e-01,  1.5094e-01, -1.3244e-01],
        [ 6.6007e-02, -1.6689e-01, -5.4067e-02, -2.0854e-01, -2.0865e-02,
         -1.3338e-01,  7.5254e-02, -1.5681e-01, -2.7144e-02, -4.7951e-02,
         -1.9064e-01,  9.0783e-02,  5.3411e-02,  6.1323e-02, -7.4286e-02,
         -4.4378e-02, -1.4506e-01,  9.6563e-02, -5.9605e-02,  1.0311e-02],
        [-9.3005e-02, -1.2205e-01, -5.7783e-02,  4.2518e-02,  1.4180e-02,
          4.7801e-02, -8.7778e-02,  7.5702e-02, -6.8394e-02, -6.5523e-02,
         -8.2122e-02, -1.4245e-02,  5.9519e-02,  7.4425e-02, -3.2761e-02,
          3.7927e-02,  1.1747e-01, -5.8839e-02, -1.1085e-01,  7.1992e-02],
        [ 3.0214e-02,  1.9807e-01,  1.7403e-01,  1.9654e-01,  1.2443e-02,
          1.7851e-01, -1.6653e-01, -4.7778e-03, -1.0772e-01, -3.5422e-02,
         -1.6564e-01,  8.9819e-02,  9.5779e-02,  1.0495e-01,  1.6255e-01,
          9.3894e-02, -3.0805e-02,  1.7827e-01,  1.0943e-02,  1.0202e-01],
        [ 1.0837e-01, -1.3197e-01, -1.2405e-01,  3.4787e-02, -1.0295e-01,
          7.8697e-02, -1.2649e-01,  2.2079e-01,  1.5915e-01,  2.5149e-02,
          1.9105e-01, -1.8830e-01, -4.6635e-02, -9.2367e-02,  2.0099e-01,
         -1.8862e-01, -1.1178e-02,  7.0899e-02,  1.6164e-02,  1.4991e-01],
        [-2.1562e-01, -1.4211e-01,  3.9175e-02,  3.4480e-03, -3.8805e-02,
         -2.1193e-02,  2.0261e-01,  1.2881e-01, -1.3818e-01,  9.6850e-02,
```

```
          1.2356e-01, -1.6617e-01,  2.9431e-02,  1.7097e-01,  2.1018e-02,
         -1.1406e-01, -1.7176e-01,  6.3938e-02, -2.1700e-01, -3.4168e-02],
        [ 1.0305e-01,  7.1974e-02,  1.7926e-01, -1.0807e-01, -1.8010e-01,
          7.8156e-02,  7.7960e-02, -1.7887e-01,  2.1744e-01, -1.5786e-01,
         -9.6197e-02, -1.5766e-01,  1.0039e-01,  1.7145e-01, -1.6024e-01,
          1.0452e-01, -9.8607e-03,  1.2370e-01, -9.0341e-02,  2.0681e-01],
        [-3.4414e-02,  4.2031e-02,  1.4340e-01,  1.2947e-01, -3.5601e-02,
          5.4370e-02, -1.2560e-01, -3.8243e-02, -1.2221e-01, -2.0495e-01,
         -1.9190e-01,  1.3532e-01, -2.0739e-01,  2.0571e-01, -1.4493e-01,
         -1.6347e-01,  1.4085e-01,  1.4128e-01, -3.8738e-02, -5.5905e-03],
        [ 9.7996e-02, -9.0367e-02,  1.3108e-01, -5.0768e-03, -1.0228e-01,
         -5.0550e-02,  5.9998e-02,  1.1742e-01,  2.3876e-02, -1.5225e-01,
         -1.3563e-01, -1.0545e-01, -1.4107e-01,  1.8589e-01, -7.8186e-05,
          8.1263e-02, -7.8986e-02,  6.1075e-02,  1.8517e-01,  2.0356e-02],
        [-1.9604e-01,  1.2420e-01,  1.5083e-01,  1.2950e-01, -1.0832e-01,
         -7.4789e-02,  1.0003e-01, -2.1659e-01,  1.2607e-01, -1.5388e-01,
          7.3955e-02,  2.0575e-01,  4.8987e-03,  1.9208e-01, -2.1015e-01,
          1.0503e-01,  3.2832e-02,  4.6586e-03, -1.8004e-01,  1.5837e-01],
        [ 1.6002e-01, -1.0169e-01, -1.6333e-01,  2.0429e-01, -2.9806e-02,
         -1.0240e-01, -1.9517e-01,  1.7168e-01, -9.4411e-02,  8.2709e-02,
         -9.5470e-02, -1.8628e-01, -4.1396e-02,  1.4047e-01, -9.8430e-02,
          2.1197e-01,  1.4795e-02,  5.6604e-02, -4.1569e-02,  1.6669e-02],
        [-1.1766e-02, -1.0670e-01,  1.3130e-01,  8.8405e-02,  2.0607e-01,
         -2.5251e-02, -1.7052e-01,  1.6100e-01, -9.1788e-02,  2.5497e-02,
          1.5090e-01,  1.1472e-01, -6.9282e-02, -8.3204e-02, -1.3479e-01,
          1.7997e-01, -1.1683e-01,  1.0269e-01, -4.5118e-03,  5.7891e-02],
        [ 8.2056e-02, -9.7197e-02,  6.1668e-02, -1.4625e-01, -8.1076e-02,
         -1.0350e-01,  5.5008e-02, -1.8393e-01, -2.1174e-01,  8.8058e-02,
          1.9541e-01,  1.4984e-01, -1.3199e-01,  1.8431e-01, -1.0089e-01,
         -1.7940e-01, -3.6352e-02,  1.0495e-01, -8.6435e-02, -4.4591e-02],
        [ 2.2297e-02, -7.9682e-02,  1.3495e-01, -8.6891e-02, -2.1674e-01,
         -1.3871e-01,  8.9895e-02,  3.3899e-02, -1.5638e-01,  1.4247e-01,
         -7.8337e-02, -8.6195e-03, -2.1294e-01, -3.8736e-03, -1.6634e-01,
         -4.7271e-02, -5.6967e-02, -1.3925e-01,  1.4928e-01, -6.6750e-04]])
sequential.2.bias tensor([ 0.0384,  0.1675, -0.1484, -0.0664, -0.1466,  0.0709,
-0.0954,  0.2016,
        -0.0727,  0.0040, -0.1135, -0.1329,  0.0582,  0.0319,  0.1820, -0.1976,
        -0.2087,  0.0053, -0.0869,  0.0370])
sequential.4.weight tensor([[-0.0328,  0.1017,  0.1603, -0.0207,  0.2069,
-0.0477,  0.0170,  0.0825,
          0.0510, -0.1392,  0.1079, -0.1453,  0.1957, -0.1726,  0.2127, -0.0127,
          0.1974,  0.0394,  0.0503, -0.1452],
        [-0.2218,  0.1814, -0.0859, -0.0749,  0.2216,  0.1843, -0.1145,  0.0385,
         -0.1652,  0.0128,  0.0589, -0.0546,  0.1152, -0.1504,  0.0213,  0.0394,
          0.1061, -0.1547,  0.1374,  0.2166],
        [-0.1574, -0.1520,  0.1723, -0.2084, -0.2121, -0.1029, -0.2030,  0.1860,
         -0.1296, -0.0092, -0.0423, -0.1936, -0.1219,  0.1467,  0.1615, -0.0578,
         -0.1715, -0.0551,  0.0101, -0.0693],
```

```
[-0.1913,  0.1772,  0.1487,  0.0674, -0.0189,  0.1860,  0.0428,  0.2160,
 -0.0461,  0.0595, -0.1887,  0.0176, -0.0971,  0.0344, -0.1047, -0.1362,
 -0.1696,  0.1213,  0.1608,  0.0663],
[ 0.0855, -0.1604,  0.0291,  0.0384,  0.1478, -0.2233, -0.1891, -0.1456,
  0.1805, -0.0175,  0.1889,  0.1100, -0.0893, -0.0035,  0.1031,  0.2184,
 -0.0573,  0.1902,  0.0081, -0.1270],
[ 0.1583,  0.1602,  0.0832, -0.0032, -0.1449, -0.1024,  0.0182,  0.0240,
  0.0540, -0.1887, -0.0623,  0.1981,  0.1683, -0.1115, -0.1328,  0.0199,
 -0.1936,  0.2201,  0.0275,  0.1270],
[-0.0519, -0.1825, -0.0541,  0.1171, -0.1152,  0.1677,  0.1934,  0.1753,
  0.0746, -0.0289, -0.0202, -0.1638, -0.1402,  0.1011, -0.0621,  0.2100,
  0.1464, -0.1507,  0.2064, -0.0327],
[-0.0250,  0.0023,  0.1713,  0.1669,  0.1357, -0.1703, -0.1492, -0.1289,
 -0.1950,  0.1060,  0.0792, -0.0079, -0.0271,  0.1957, -0.1403, -0.0616,
  0.0500,  0.0314, -0.1541,  0.1449],
[-0.0342, -0.1706, -0.1337, -0.1728,  0.0083,  0.1147,  0.1744, -0.0255,
  0.2196, -0.1641,  0.0725, -0.0871, -0.0757,  0.0107, -0.1525,  0.0589,
 -0.0432, -0.0469,  0.1867, -0.2013],
[ 0.1617,  0.1569, -0.0442, -0.2072, -0.0987,  0.0264, -0.2205,  0.1215,
  0.1639, -0.1715,  0.1950, -0.1018, -0.1390,  0.0927, -0.0971,  0.0058,
  0.0886, -0.0898, -0.0637, -0.0732],
[ 0.1248, -0.1141, -0.1505,  0.0157, -0.0937,  0.0031, -0.1796,  0.2227,
  0.1586,  0.1266, -0.0866,  0.1903,  0.0503, -0.0611,  0.1860,  0.0711,
 -0.0078, -0.2180,  0.2016,  0.1407],
[-0.1856,  0.0774,  0.0329,  0.2182, -0.1940, -0.0757,  0.1795,  0.0838,
 -0.0772, -0.0099,  0.2199,  0.1410,  0.0708, -0.1759, -0.1186,  0.1735,
  0.0574,  0.2022,  0.2153, -0.1686],
[-0.1992, -0.0117,  0.1680, -0.0696, -0.0476, -0.1062,  0.0273,  0.1564,
 -0.1673,  0.2165, -0.2112, -0.0862, -0.1365,  0.0865,  0.0200,  0.1846,
 -0.0836,  0.1739, -0.0900, -0.0181],
[ 0.0187,  0.1842, -0.2163, -0.0038,  0.1808, -0.1690, -0.1122,  0.0961,
 -0.2063, -0.1676,  0.1769, -0.1765, -0.1157,  0.0310,  0.0022, -0.1335,
  0.1332,  0.1748, -0.2232, -0.0357],
[-0.1757, -0.1816,  0.0089, -0.1091, -0.0463, -0.2137,  0.1781,  0.1587,
  0.2005, -0.1550, -0.1186,  0.1130,  0.0165, -0.0687, -0.0061, -0.1467,
 -0.0029,  0.1770,  0.0698, -0.2067],
[ 0.0646,  0.1986,  0.1727,  0.1587,  0.0691, -0.2221,  0.2211, -0.1106,
  0.1462,  0.0598, -0.0579, -0.1196,  0.0143,  0.0753,  0.1730,  0.2141,
 -0.0035,  0.0280,  0.0016,  0.1193],
[ 0.0546,  0.0852, -0.1504, -0.0608,  0.2044,  0.1368, -0.1793, -0.1402,
 -0.0853, -0.1815,  0.0154,  0.0399, -0.0308, -0.0645,  0.0406,  0.0798,
 -0.2136, -0.1598,  0.1188,  0.0295],
[ 0.0418, -0.1365, -0.2085, -0.1861,  0.0454, -0.0655,  0.1152,  0.1810,
 -0.1261, -0.0828, -0.1887,  0.0970,  0.0412, -0.0333,  0.1157,  0.1541,
 -0.1968,  0.1226, -0.0740, -0.0092],
[-0.0726, -0.1721,  0.2043,  0.1599, -0.1584,  0.0096,  0.0438,  0.1168,
 -0.1073,  0.0057, -0.0354,  0.1541, -0.2009,  0.1883, -0.1522, -0.0558,
  0.0811, -0.0281, -0.1294, -0.1081],
```

```
        [-0.0574, -0.0768, -0.0124,  0.0116, -0.0808, -0.1476, -0.1579, -0.0798,
          0.2233, -0.2223,  0.0231, -0.1881, -0.1066, -0.0709, -0.1777,  0.2097,
         -0.1415,  0.1556,  0.1995, -0.1774]])
sequential.4.bias tensor([-0.2052,  0.1219, -0.0381,  0.0570, -0.0791, -0.0512,
0.2217,  0.1534,
         0.1287,  0.0299, -0.1493,  0.2015, -0.0402, -0.1203, -0.1370, -0.1700,
        -0.1530, -0.1508,  0.0076, -0.2029])
sequential.6.weight tensor([[ 0.1407, -0.1317, -0.1480, -0.0112,  0.0804,
0.0183,  0.0019,  0.0959,
          0.1853,  0.2048, -0.0921,  0.0713, -0.0500, -0.1992,  0.0937,  0.0872,
          0.0577,  0.1913, -0.2132, -0.0914],
        [ 0.0351,  0.0904,  0.1354,  0.1774, -0.1952,  0.0890,  0.1787, -0.1625,
          0.1438, -0.0318,  0.0824,  0.0227,  0.1613, -0.1177, -0.1136, -0.0780,
         -0.0401,  0.1830, -0.0517, -0.0735],
        [-0.0967,  0.2012,  0.2116, -0.0869,  0.0300,  0.0103,  0.0804, -0.2122,
         -0.0528,  0.1860, -0.1196, -0.2211, -0.0817, -0.0223,  0.0777,  0.2127,
          0.1102,  0.1062,  0.0765,  0.1420],
        [ 0.1853, -0.1773,  0.2008, -0.0795, -0.0448,  0.1012, -0.0889,  0.1282,
          0.1711, -0.0640, -0.0573, -0.1727,  0.1842,  0.0035, -0.1056,  0.1667,
         -0.1410, -0.0273, -0.1211,  0.1816],
        [-0.1226,  0.0653,  0.1971,  0.0672, -0.0834,  0.1627,  0.0349,  0.0614,
          0.1440,  0.1561,  0.1390,  0.1705,  0.0080,  0.0396,  0.1892, -0.1712,
          0.1208, -0.1791,  0.1827, -0.0189],
        [-0.1856,  0.1715, -0.1257,  0.0092,  0.0313,  0.1958,  0.1442,  0.1923,
          0.0361,  0.1847, -0.0169, -0.1937, -0.1836,  0.0075, -0.0305,  0.2178,
          0.0108,  0.1276, -0.0579,  0.1803],
        [-0.1330, -0.0311,  0.1751, -0.2078, -0.0581,  0.1069, -0.1868, -0.0276,
         -0.2176, -0.1213,  0.1516, -0.1003, -0.1161, -0.1116,  0.0313, -0.1281,
         -0.1008,  0.1847,  0.0572,  0.1567],
        [ 0.1497, -0.0450, -0.1508,  0.0230,  0.0331, -0.1326,  0.1962, -0.1548,
          0.1095, -0.0055, -0.1197,  0.1817, -0.0384, -0.0085, -0.2171,  0.1180,
         -0.1820, -0.1499, -0.0532,  0.1394],
        [-0.1754,  0.1199, -0.0408,  0.1384,  0.2232,  0.0337,  0.1664,  0.1642,
         -0.1094,  0.1046, -0.2112,  0.0033,  0.0442,  0.1583,  0.1690,  0.0959,
         -0.1876,  0.2040,  0.2105,  0.1605],
        [ 0.1081, -0.0719,  0.1751,  0.2058, -0.0782, -0.1241, -0.1922,  0.1261,
         -0.1976, -0.0626, -0.0594, -0.0187, -0.1095,  0.1102,  0.1152, -0.0998,
         -0.1859,  0.0078, -0.0403, -0.2008],
        [ 0.1827,  0.1599, -0.0313,  0.1229, -0.1790,  0.0510,  0.1732, -0.1475,
          0.1279,  0.0836,  0.0918, -0.0480,  0.1722,  0.0486, -0.0373, -0.1334,
         -0.1410,  0.0416, -0.0856,  0.2038],
        [-0.0450,  0.0990, -0.0495,  0.1628, -0.1717, -0.0893, -0.2225,  0.1727,
         -0.1829,  0.1304, -0.1546, -0.1672, -0.0755,  0.1211, -0.0466, -0.1024,
         -0.0110,  0.1702, -0.2212, -0.0373],
        [-0.0875, -0.1230, -0.0374, -0.0807,  0.2035,  0.0557, -0.0752, -0.1493,
         -0.0249,  0.2047,  0.1348, -0.1211,  0.1985, -0.0589,  0.0559,  0.1896,
         -0.1002, -0.0483, -0.1018, -0.1713],
        [-0.1034, -0.0152, -0.0105,  0.2108, -0.2235, -0.0299, -0.2048,  0.0599,
```

```
         0.1045, -0.0520,  0.1817,  0.1589, -0.1114, -0.1020, -0.1028,  0.0247,
        -0.1563, -0.0521,  0.0401,  0.0343],
        [-0.1159,  0.1330,  0.0882, -0.0100, -0.0798,  0.2134, -0.0734, -0.1006,
         0.1097, -0.0550,  0.2202,  0.2014, -0.0679, -0.1237, -0.0005,  0.0473,
        -0.0707, -0.1293,  0.1959, -0.1668],
        [-0.1478, -0.1879, -0.2029, -0.1381, -0.1593,  0.0335, -0.0186,  0.0127,
         0.0773, -0.1281, -0.0378,  0.1720,  0.1358,  0.2111,  0.0593, -0.2037,
        -0.1460, -0.1793, -0.0409,  0.0003],
        [ 0.0235, -0.1737, -0.0527,  0.1044, -0.1195, -0.0127,  0.2031, -0.0513,
        -0.1771,  0.1209,  0.1008,  0.0617, -0.1901,  0.1313, -0.1139, -0.0538,
         0.0583,  0.0507,  0.0428,  0.0258],
        [-0.0270, -0.0450, -0.1169, -0.1976, -0.1532, -0.2109, -0.2096, -0.1385,
        -0.0734, -0.1911,  0.0769, -0.0559,  0.2158,  0.1231, -0.1212, -0.1460,
        -0.1346,  0.0703, -0.1615, -0.1064],
        [ 0.0848, -0.1975, -0.1721,  0.0912, -0.1996,  0.1598, -0.1096, -0.1403,
         0.1672, -0.0673,  0.1118,  0.1677, -0.0415, -0.1287,  0.1270, -0.0877,
         0.1027, -0.1508, -0.1404, -0.1150],
        [-0.1466, -0.2127,  0.1986, -0.1811, -0.0616,  0.0386,  0.0518, -0.1947,
        -0.0429,  0.0912, -0.1557, -0.1351,  0.2166, -0.1711, -0.1866,  0.1986,
        -0.0659,  0.1153,  0.1025,  0.0606]])
sequential.6.bias tensor([-0.0674,  0.1514, -0.0894,  0.0924,  0.0653,  0.0093,
-0.1218,  0.1894,
        -0.1824,  0.1709,  0.1847,  0.0383,  0.0506, -0.0894,  0.0120, -0.1381,
        -0.1559, -0.1866,  0.0026, -0.0388])
sequential.8.weight tensor([[ 0.1530, -0.1454, -0.0048,  0.0304, -0.1437,
0.1478,  0.0091,  0.0004,
         -0.1213, -0.0522,  0.1388, -0.0894, -0.1294, -0.0099,  0.1128,  0.0589,
         -0.0781,  0.2152,  0.2192,  0.1193]])
sequential.8.bias tensor([0.2182])
```

0.6.1 Which one is correct?

In pytorch, you need to specify the input and output dimension for each layer. The input dimension of each layer must match the output dimension for the previous layer. Which one of the following code is correct?

[ ]:
```python
class myMultiLayerPerceptron_1(nn.Module):
    def __init__(self,input_dim,output_dim):
        super().__init__()
        self.sequential = nn.Sequential(  # here we stack multiple layers␣
 ↪together
            nn.Linear(input_dim,20),
            nn.ReLU(),
            nn.Linear(20,30),
            nn.ReLU(),
            nn.Linear(20,30),
            nn.ReLU(),
            nn.Linear(20,30),
```

```
            nn.ReLU(),
            nn.Linear(20,output_dim)
        )
    def forward(self,x):
        y = self.sequential(x)
        return y

class myMultiLayerPerceptron_2(nn.Module):
    def __init__(self,input_dim,output_dim):
        super().__init__()
        self.sequential = nn.Sequential(  # here we stack multiple layers␣
 ↪together
            nn.Linear(input_dim,30),
            nn.ReLU(),
            nn.Linear(30,20),
            nn.ReLU(),
            nn.Linear(20,30),
            nn.ReLU(),
            nn.Linear(30,20),
            nn.ReLU(),
            nn.Linear(20,output_dim)
        )
    def forward(self,x):
        y = self.sequential(x)
        return y
```

[ ]:
```
mymlp_1 = myMultiLayerPerceptron_1(1,1)
mymlp_1(torch.tensor([[1.0]]))
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

/Users/coolq/Library/CloudStorage/Box-Box/Teaching/Tool Chain/Toolchain 2023␣
 ↪Fall/notebooks/Lecture_16_pytorch_neural_networks.ipynb Cell 16 line 2


      <a href='vscode-notebook-cell:/Users/coolq/Library/CloudStorage/Box-Box/
 ↪Teaching/Tool%20Chain/Toolchain%202023%20Fall/notebooks/
 ↪Lecture_16_pytorch_neural_networks.ipynb#X21sZmlsZQ%3D%3D?line=0'>1</a>␣
 ↪mymlp_1 = myMultiLayerPerceptron_1(1,1)

----> <a href='vscode-notebook-cell:/Users/coolq/Library/CloudStorage/Box-Box/
 ↪Teaching/Tool%20Chain/Toolchain%202023%20Fall/notebooks/
 ↪Lecture_16_pytorch_neural_networks.ipynb#X21sZmlsZQ%3D%3D?line=1'>2</a>␣
 ↪mymlp_1(torch.tensor([[1.0]]))
```

```
File ~/opt/anaconda3/envs/sparktest2/lib/python3.9/site-packages/torch/nn/
↪modules/module.py:1102, in Module._call_impl(self, *input, **kwargs)

  1098 # If we don't have any hooks, we want to skip the rest of the logic in

  1099 # this function, and just call forward.

  1100 if not (self._backward_hooks or self._forward_hooks or self.
↪_forward_pre_hooks or _global_backward_hooks

  1101         or _global_forward_hooks or _global_forward_pre_hooks):

-> 1102     return forward_call(*input, **kwargs)

  1103 # Do not call functions when jit is used

  1104 full_backward_hooks, non_full_backward_hooks = [], []



/Users/coolq/Library/CloudStorage/Box-Box/Teaching/Tool Chain/Toolchain 2023␣
↪Fall/notebooks/Lecture_16_pytorch_neural_networks.ipynb Cell 16 line 1


     <a href='vscode-notebook-cell:/Users/coolq/Library/CloudStorage/Box-Box/
↪Teaching/Tool%20Chain/Toolchain%202023%20Fall/notebooks/
↪Lecture_16_pytorch_neural_networks.ipynb#X21sZmlsZQ%3D%3D?line=14'>15</a> def␣
↪forward(self,x):

---> <a href='vscode-notebook-cell:/Users/coolq/Library/CloudStorage/Box-Box/
↪Teaching/Tool%20Chain/Toolchain%202023%20Fall/notebooks/
↪Lecture_16_pytorch_neural_networks.ipynb#X21sZmlsZQ%3D%3D?line=15'>16</a>    ␣
↪y = self.sequential(x)

     <a href='vscode-notebook-cell:/Users/coolq/Library/CloudStorage/Box-Box/
↪Teaching/Tool%20Chain/Toolchain%202023%20Fall/notebooks/
↪Lecture_16_pytorch_neural_networks.ipynb#X21sZmlsZQ%3D%3D?line=16'>17</a>    ␣
↪return y



File ~/opt/anaconda3/envs/sparktest2/lib/python3.9/site-packages/torch/nn/
↪modules/module.py:1102, in Module._call_impl(self, *input, **kwargs)

  1098 # If we don't have any hooks, we want to skip the rest of the logic in

  1099 # this function, and just call forward.

  1100 if not (self._backward_hooks or self._forward_hooks or self.
↪_forward_pre_hooks or _global_backward_hooks
```

```
   1101         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1102     return forward_call(*input, **kwargs)
   1103 # Do not call functions when jit is used
   1104 full_backward_hooks, non_full_backward_hooks = [], []


File ~/opt/anaconda3/envs/sparktest2/lib/python3.9/site-packages/torch/nn/
 ↪modules/container.py:141, in Sequential.forward(self, input)
    139 def forward(self, input):
    140     for module in self:
--> 141         input = module(input)
    142     return input


File ~/opt/anaconda3/envs/sparktest2/lib/python3.9/site-packages/torch/nn/
 ↪modules/module.py:1102, in Module._call_impl(self, *input, **kwargs)
   1098 # If we don't have any hooks, we want to skip the rest of the logic in
   1099 # this function, and just call forward.
   1100 if not (self._backward_hooks or self._forward_hooks or self.
 ↪_forward_pre_hooks or _global_backward_hooks
   1101         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1102     return forward_call(*input, **kwargs)
   1103 # Do not call functions when jit is used
   1104 full_backward_hooks, non_full_backward_hooks = [], []


File ~/opt/anaconda3/envs/sparktest2/lib/python3.9/site-packages/torch/nn/
 ↪modules/linear.py:103, in Linear.forward(self, input)
    102 def forward(self, input: Tensor) -> Tensor:
```

```
--> 103     return F.linear(input, self.weight, self.bias)



File ~/opt/anaconda3/envs/sparktest2/lib/python3.9/site-packages/torch/nn/
↪functional.py:1848, in linear(input, weight, bias)

   1846 if has_torch_function_variadic(input, weight, bias):

   1847     return handle_torch_function(linear, (input, weight, bias), input,␣
↪weight, bias=bias)

-> 1848 return torch._C._nn.linear(input, weight, bias)



RuntimeError: mat1 and mat2 shapes cannot be multiplied (1x30 and 20x30)
```

[ ]:
```python
mymlp_2 = myMultiLayerPerceptron_2(1,1)
mymlp_2(torch.tensor([[1.0]]))
```

```
tensor([[-0.0547]], grad_fn=<AddmmBackward0>)
```

0.7 4.3 Prepare Training Data

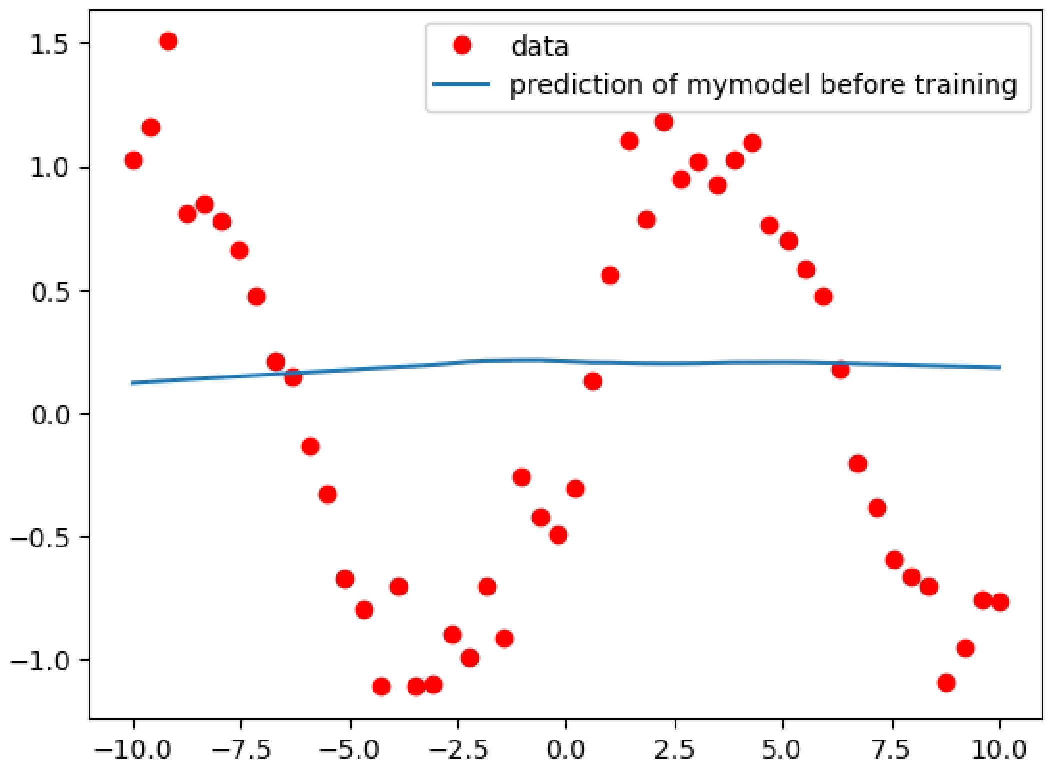
Since neural networks are nonlinear ML models, we are going to create some nonlinear training data.

[ ]:
```python
# Now let's create some simple synthetic data
import numpy as np
import matplotlib.pyplot as plt

N_samples = 50
x = torch.linspace(-10,10,N_samples,dtype=torch.float)
x = x[:,None]
y = torch.sin(0.5*x) + np.random.randn(N_samples,1)*0.2


prediction = mymodel(x).detach().numpy()
plt.plot(x,y,'ro')
plt.plot(x,prediction)
plt.legend(['data','prediction of mymodel before training'])
```

```
<matplotlib.legend.Legend at 0x7ff6d31e3070>
```



Since we will be running SGD, we need to do batching. Therefore, let's convert the training data tensors into Dataset objects.

[ ]:
```
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader

class MyDataset(Dataset):
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        return (self.x[idx],self.y[idx])
```

[ ]:
```
mydataset = MyDataset(x,y) # generate a Dataset based on x,y

# Randomly split dataset into train and validate dataset
dataset_len = len(mydataset)
train_dataset_len = round(dataset_len*0.8)
```

```
validate_dataset_len = dataset_len - train_dataset_len
train_dataset,validate_dataset = torch.utils.data.
 ↪random_split(mydataset,[train_dataset_len, validate_dataset_len])
```

0.8 4.4 Training Neural Network via Stochastic Gradient Descent

Overall, the training loop is almost identical as before, which is a nested for-loop. In each training iteration in the training loop, recall we have the following steps.

- First the prediction is computed based on the input variable of a small batch of the training data set (obtained from DataLoader).
- Then, the prediction, together with the true output, is used to compute the loss.
- Then, we run the optimizer.zero_grad(), which clears the gradient computed from the previous loop.
- Then, we run loss.backward() which calculates the gradient of the loss w.r.t. the parameters
- Finally, optimizer.step() conducts a gradient step

One difference this time is that at the end of each epoch, we also calculate the validation loss using the validation dataset. This is a common practice in deep learning.

The code below also saves the traning process as a GIF file so that you can visualize the training process.

[ ]:
```
# Now let's do the training!
import io
import imageio
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

mymodel = myMultiLayerPerceptron(1,1) # creating a model instance with input␣
 ↪dimension 1

# Three hyper parameters for training
lr = .04
batch_size = 10
N_epochs = 160

# Create dataloaders for training and validation
train_dataloader = DataLoader(train_dataset, batch_size = batch_size, shuffle =␣
 ↪True)
validate_dataloader = DataLoader(validate_dataset,batch_size =␣
 ↪batch_size,shuffle = True)

# Create optimizer
optimizer = torch.optim.SGD(mymodel.parameters(), lr = lr) # this line creates␣
 ↪a optimizer, and we tell optimizer we are optimizing the parameters in␣
 ↪mymodel
```

```
frames = [] # This variable stores all images to be saved to the GIF file

losses = [] # training losses of each epoch
validate_losses = [] # validation losses of each epoch
losses_all = [] # training losses of each SGD iteration

gd_steps = 0
N_batches = len(train_dataloader)

for epoch in range(N_epochs):
    batch_loss = []
    for batch_id, (x_batch, y_batch) in enumerate(train_dataloader):
        gd_steps+=1
        # pass input data to get the prediction outputs by the current model
        prediction = mymodel(x_batch)

        # compare prediction and the actual output and compute the loss
        loss = torch.mean((prediction - y_batch)**2)

        # compute the gradient
        optimizer.zero_grad()
        loss.backward()

        # update parameters
        optimizer.step()


        # Generate visualization plots
        fig, ax = plt.subplots(nrows = 1, ncols = 3)
        canvas = FigureCanvas(fig)
        ax[0].plot(x,y,'ro')
        prediction_full = mymodel(x)
        ax[0].plot(x,prediction_full.detach(),linewidth = 2)
        ax[0].legend(['data','prediction of mymodel'],loc = 'upper left')
        ax[0].set_title(f"Batch size = {batch_size}, Learning rate = {lr},␣
↪Epoch #{epoch}, Batch #{batch_id}", fontsize = 20)
        ax[0].set_xlim((-10,10))
        ax[0].set_ylim((-2,2))
        losses_all.append(loss.detach().numpy())
        ax[1].plot(np.arange(gd_steps),np.array(losses_all).
↪squeeze(),linewidth=2 )
        ax[1].set_xlim((0,(N_epochs+1)*(N_batches)))
        ax[1].set_ylim((0,2))
        ax[1].set_title("Train loss per iteration", fontsize = 20)
        ax[1].set_xlabel("# of SGD Iterations", fontsize = 20)
```

```
        batch_loss.append(loss.detach().numpy())
        if epoch>0:
            ax[2].plot(np.arange(epoch),np.array(losses).squeeze(),linewidth=2,␣
↪label = 'train loss' )
            ax[2].plot(np.arange(epoch),np.array(validate_losses).
↪squeeze(),linewidth=2, label = 'validate loss')
            ax[2].legend(fontsize = 20)

        ax[2].set_xlim((0,N_epochs-1))
        ax[2].set_ylim((0,2))
        ax[2].set_title("Train/validate loss per epoch", fontsize = 20)
        ax[2].set_xlabel("# of Epochs", fontsize = 20)
        fig.set_size_inches(27,9)
        canvas.draw()       # draw the canvas, cache the renderer

        image = np.frombuffer(canvas.tostring_rgb(), dtype='uint8')

        image = image.reshape(fig.canvas.get_width_height()[::-1] + (3,))

        frames.append(image)
        plt.close(fig)

    # Calculate Validation Loss
    validate_batch_loss = []
    for x_batch, y_batch in validate_dataloader:
        # pass input data to get the prediction outputs by the current model
        prediction = mymodel(x_batch)

        # compare prediction and the actual output and compute the loss
        loss = torch.mean((prediction - y_batch)**2)
        validate_batch_loss.append(loss.detach())

    validate_losses.append( np.mean(np.array(validate_batch_loss)))
    losses.append(np.mean(np.array(batch_loss)))

print("Saving GIF file")
with imageio.get_writer("MLPSGD.gif", mode="I") as writer:
    for frame in frames:
        writer.append_data(frame)
```

```
Saving GIF file
```

[ ]:
```
# let's see how the model looks like!



prediction = mymodel(x).detach().numpy()
plt.plot(x,y,'ro')
```

```
plt.plot(x,prediction,linewidth = 4)
plt.legend(['data','prediction of mymodel after training'])
```

```
<matplotlib.legend.Legend at 0x7ff6d335a370>
```