PyTorch
(Stochastic) Gradient Descent and Neural Network

Lecture 12 for 14-763/18-763

Guannan Qu

Oct 09, 2024

```python
maxIter = 100

mymodel = MyLinearRegressionModel(1)

# this creates a optimizer, and we tell optimizer we are optimizing the parameters in mymodel
optimizer = torch.optim.SGD(mymodel.parameters(), lr = 1e-3)

for _ in range(maxIter):

    # pass input data to get the prediction outputs by the current model
    prediction = mymodel(x)

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

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

    # update parameters
    optimizer.step()
```

Forward Pass

Backward pass and compute gradient.
Note: VERY IMPORTANT to run optimizer.zero_grad() to reset grad to zero! Otherwise, the backward will be incorrect.

Run a gradient descent on the parameters using the computed gradient and the learning rate

```python
maxIter = 100

mymodel = MyLinearRegressionModel(1)

# this creates a optimizer, and we tell optimizer we are optimizing the parameters in mymodel
optimizer = torch.optim.SGD(mymodel.parameters(), lr = 1e-3)

for _ in range(maxIter):

    # pass input data to get the prediction outputs by the current model
    prediction = mymodel(x)

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

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

    # update parameters
    optimizer.step()
```

Model

Optimizer

Model Parameters

Loss

Optimizer

How does these variables interact with each other?

```python
maxIter = 100

mymodel = MyLinearRegressionModel(1)

# this creates a optimizer, and we tell optimizer we are optimizing the parameters in mymodel
optimizer = torch.optim.SGD(mymodel.parameters(), lr = 1e-3)

for _ in range(maxIter):

    # pass input data to get the prediction outputs by the current model
    prediction = mymodel(x)

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

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

    # update parameters
    optimizer.step()
```

[mymodel.w, mymodel.b]

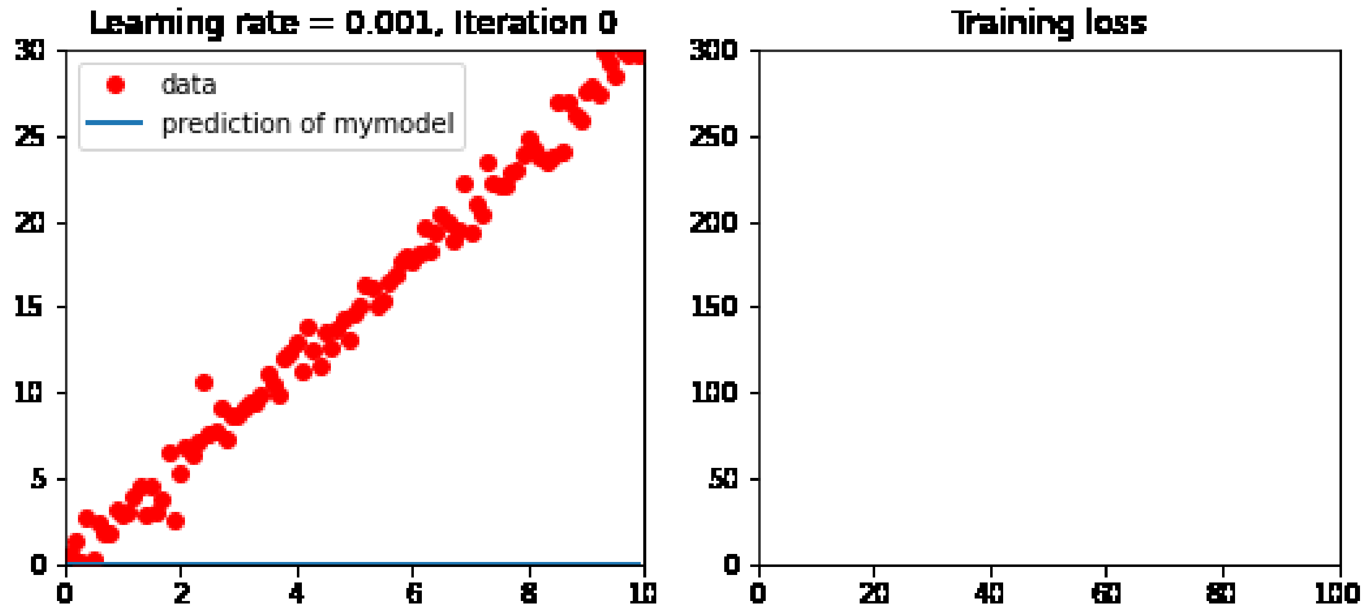
Parameters used to calculate forward forward

Backward computes gradient which is stored in parameters

Optimizer run a gradient descent on the parameters using the computed gradient and the learning rate

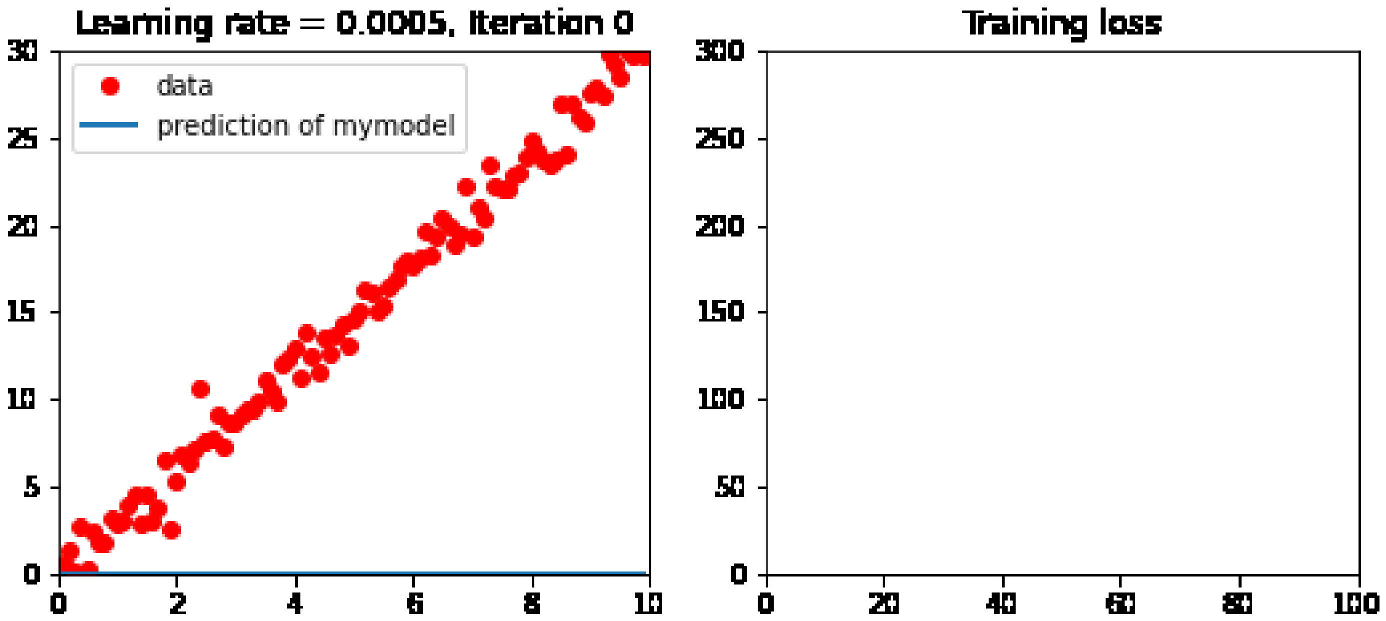
Visualizing Gradient Descent

Learning rate = 0.001



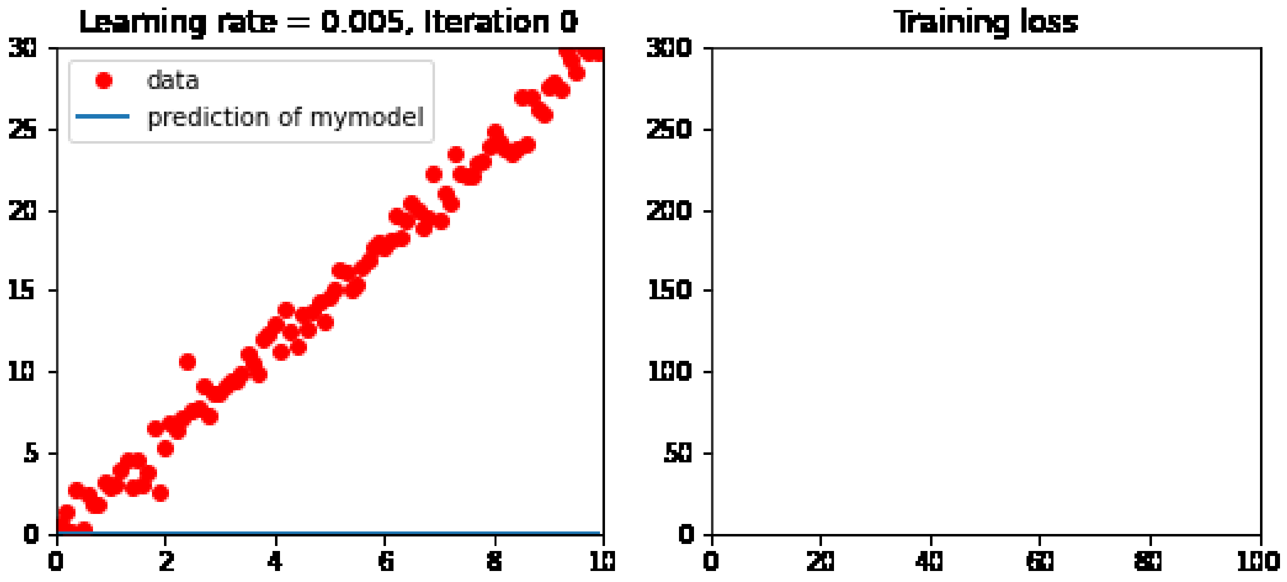
Visualizing Gradient Descent

Learning rate = 0.0005 (smaller than our first trial)



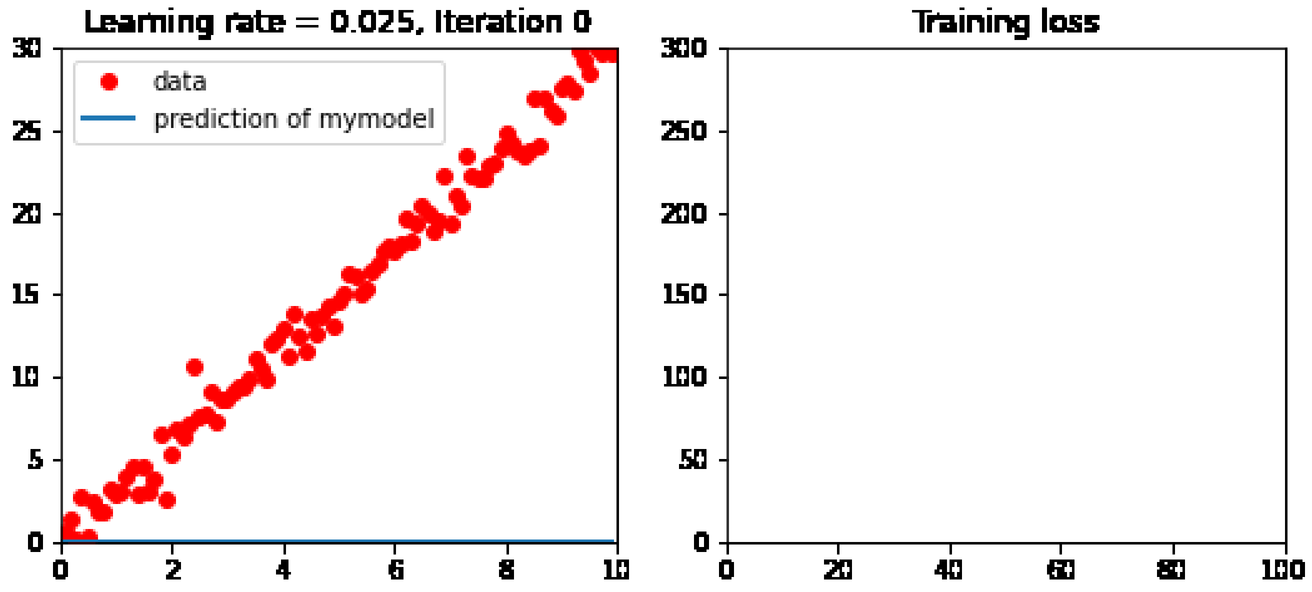
Visualizing Gradient Descent

Learning rate = 0.005 (larger than our first trial)



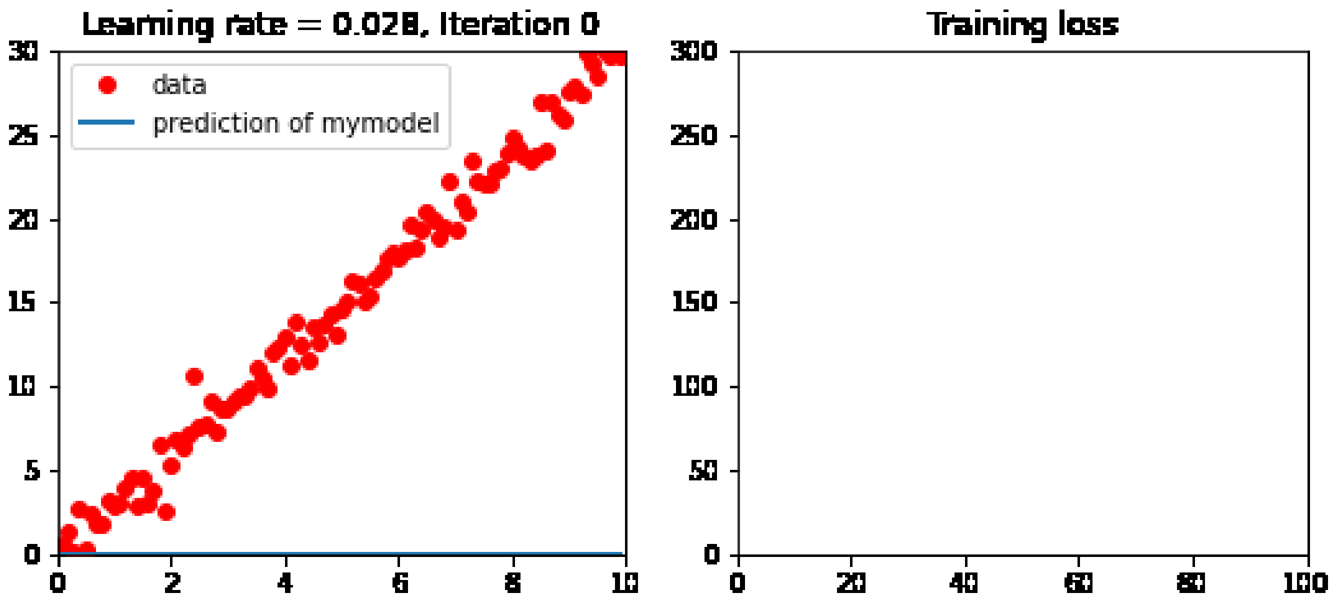
Visualizing Gradient Descent

Learning rate = 0.025 (much larger than our first trial)



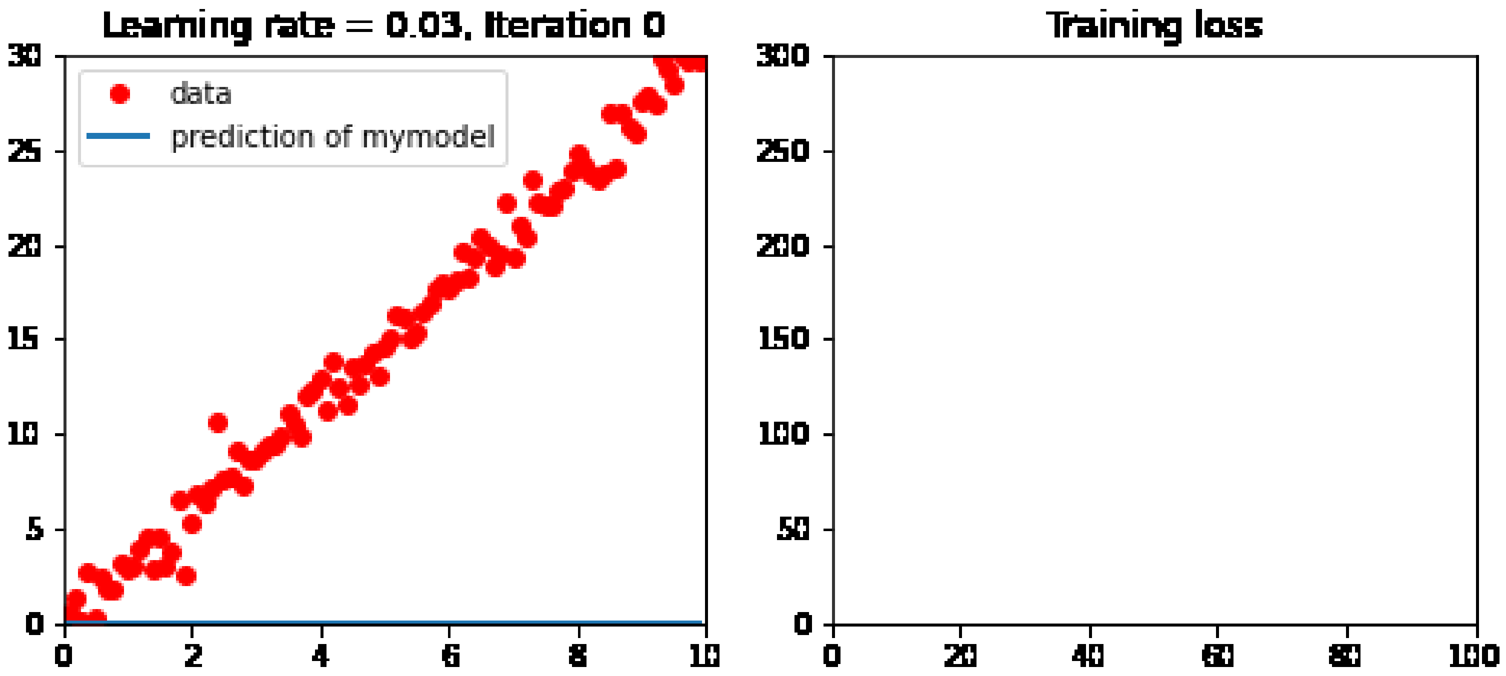
Visualizing Gradient Descent

Learning rate = 0.028 (much larger than our first trial)



Visualizing Gradient Descent

Learning rate = 0.03 (much larger than our first trial)



Lessons Learned on Learning Rate

- Learning rate too small:
  - Converges too slow and takes a lot of iterations
- Learning rate too large:
  - Exhibit unstable (oscillating) behaviors and may diverge

- How to find a good learning rate:
  - Find a small enough learning rate that does not diverge
  - Increase learning rate and plot the training loss curve
    - If the loss curve appears to be converging and “stable”, can further increase
    - If the loss curve appears to be unstable and shows signs of divergence, decrease learning rate

Up Next: Stochastic gradient descent

```python
maxIter = 100

mymodel = MyLinearRegressionModel(1)

# this creates a optimizer, and we tell optimizer we are optimizing the parameters in mymodel
optimizer = torch.optim.SGD(mymodel.parameters(), lr = 1e-3)

for _ in range(maxIter):

    # pass input data to get the prediction outputs by the current model
    prediction = mymodel(x)

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

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

    # update parameters
    optimizer.step()
```

We used the entire training dataset to do forward pass!

- This is fine because our dataset is small (~100 samples)
- But would be too computationally heaving for large datasets
- Think about x, y contains 1 million records - take a long time to conduct a single forward step!

```python
maxIter = 100

mymodel = MyLinearRegressionModel(1)

# this creates a optimizer, and we tell optimizer we are optimizing the parameters in mymodel
optimizer = torch.optim.SGD(mymodel.parameters(), lr = 1e-3)

for _ in range(maxIter):

    # pass input data to get the prediction outputs by the current model
    prediction = mymodel(x)

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

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

    # update parameters
    optimizer.step()
```

Stochastic Gradient Descent: Let's replace these with a small batch of the training set!

Stochastic Gradient Descent

- Current approach (Gradient Descent):
  - Use the full training dataset to do forward, backward, and gradient descent
  - May be computationally difficult if the full training dataset is too large

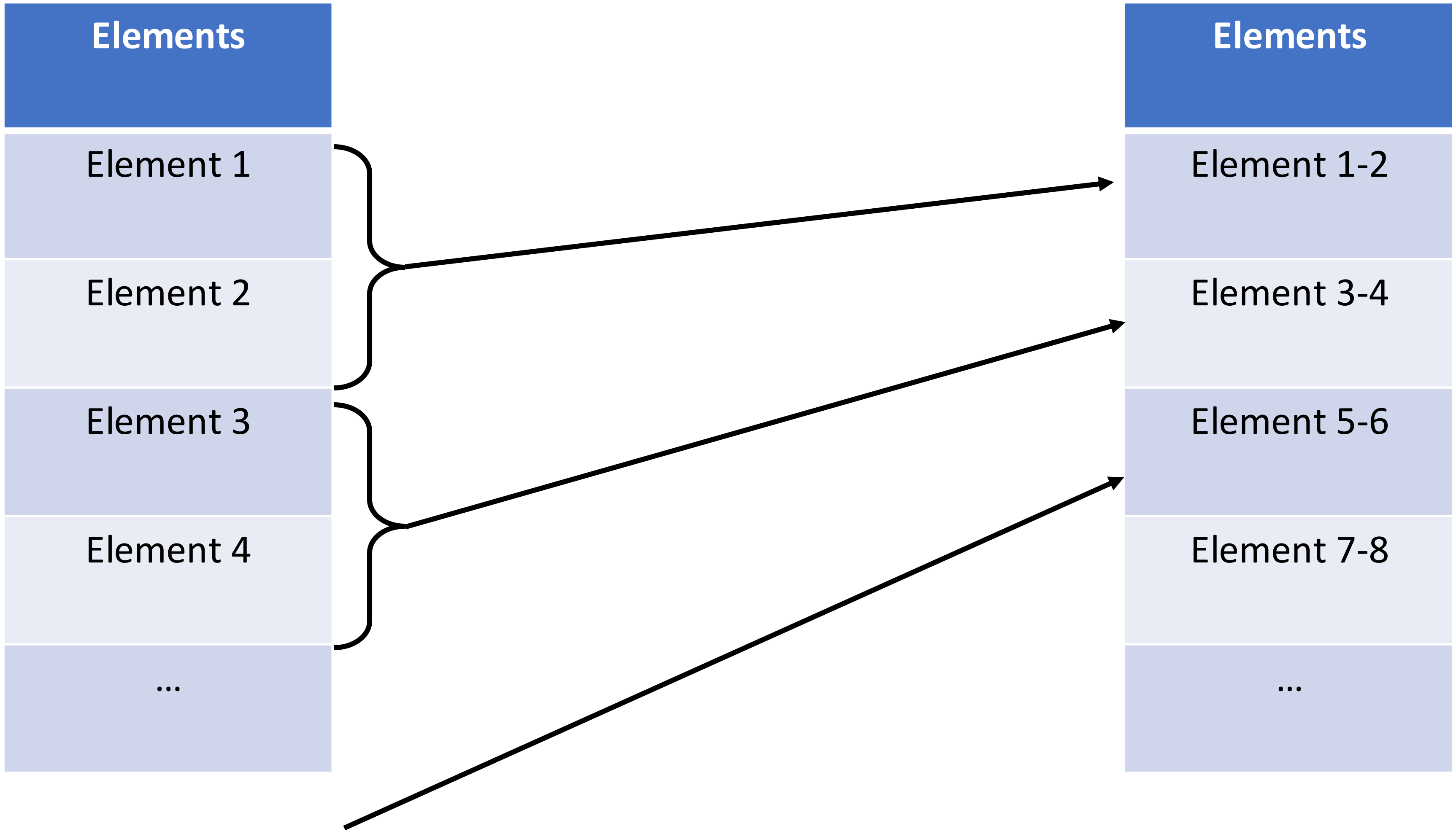
- Stochastic Gradient Descent:
  - Every time only using a random ”batch” of samples to do forward, backward and gradient descent

Stochastic Gradient Descent

Batch size = 2

Original Dataset

Batched Dataset



Stochastic Gradient Descent

How to do batching in PyTorch?

Use the torch.utils.data.Dataset and Dataloader API

Batch size = 2

Batched Dataset

Elements
Element 1-2
Element 3-4
Element 5-6
Element 7-8
…

One epoch means go through all batches one time.

Use the 1st batch to do a GD step

Use the 2nd batch to do a GD step

…

…

Dataset/DataLoader in PyTorch

```
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])
```

The general way to create Dataset is to subclassing Dataset and define two methods:

- __len__() which returns the total number of elements
- __getitem__() which returns a element of a given index

Initialization, where we save x and y as the attribute of dataset.

Return the total number of elements

Return a particular element with index idx

Dataset/DataLoader in PyTorch

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

✓ 0.1s

```
Output exceeds the size limit. Open the full output data in a text editor
(tensor([0.]), tensor([0.4506]))
(tensor([0.1000]), tensor([-2.1006]))
(tensor([0.2000]), tensor([1.0836]))
(tensor([0.3000]), tensor([0.0449]))
```

Dataset/DataLoader in PyTorch

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

✓ 0.5s

```
Output exceeds the size limit. Open the full output data in a text editor
[tensor([[7.5000],
        [2.8000],
        [4.8000],
        [0.6000]]), tensor([[23.5160],
        [ 8.1548],
        [16.2094],
        [ 2.7893]])]
[tensor([[2.6000],
        [9.6000],
        [6.4000],
        [2.3000]]), tensor([[ 6.9438],
        [30.0177],
        [18.2395],
        [ 8.1607]])]
```

Stochastic Gradient Descent

How to implement Stochastic Gradient Descent in PyTorch?

One epoch means go through all batches one time.

Use the 1st batch to do a GD step

Use the 2nd batch to do a GD step

…

…

Batch size = 2

Batched Dataset

Elements
Element 1-2
Element 3-4
Element 5-6
Element 7-8
…

SGD

Outer for loop is for the epochs

Inner for loop is go through all batches of data set

```
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()
```

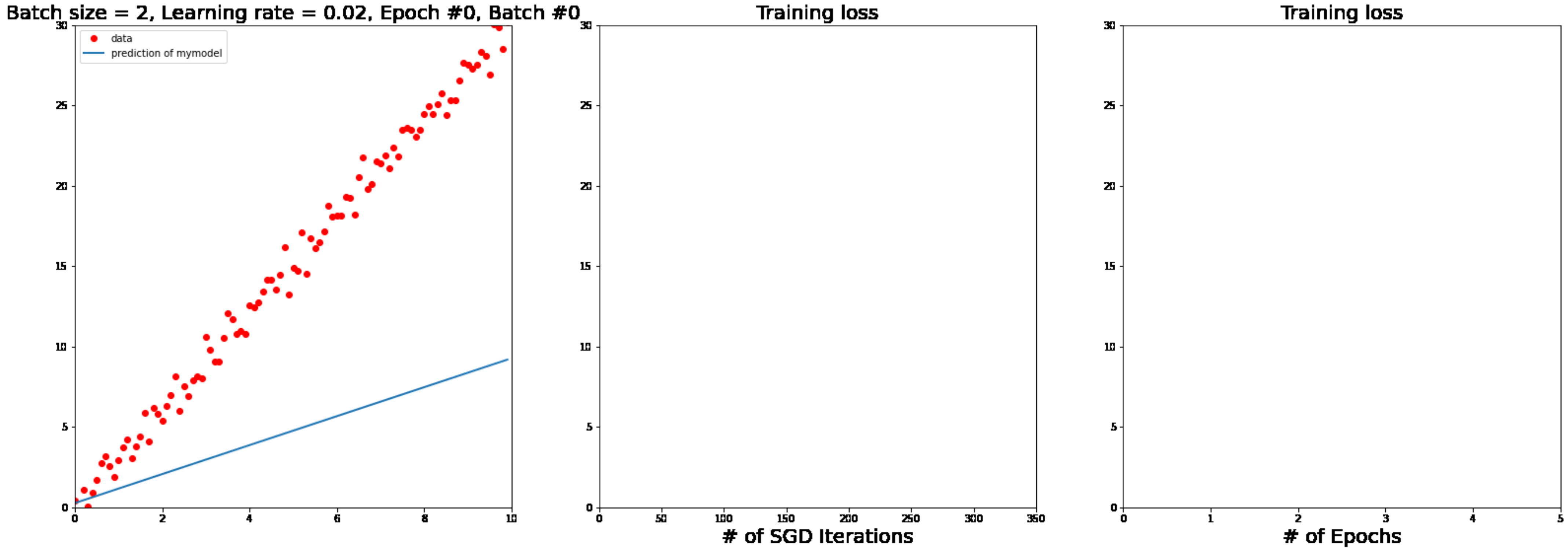
Only use the batch to compute the loss!

The rest of the steps are similar as before

Summary for SGD

- Create Dataset, and use DataLoader to generate batches of data
- Use two nested for-loops
  - The outer loop is for the epochs
  - The inner loop goes through the batches
- Only use the batch data to compute the loss (forward pass)
- Two important parameters: batch size and learning rate

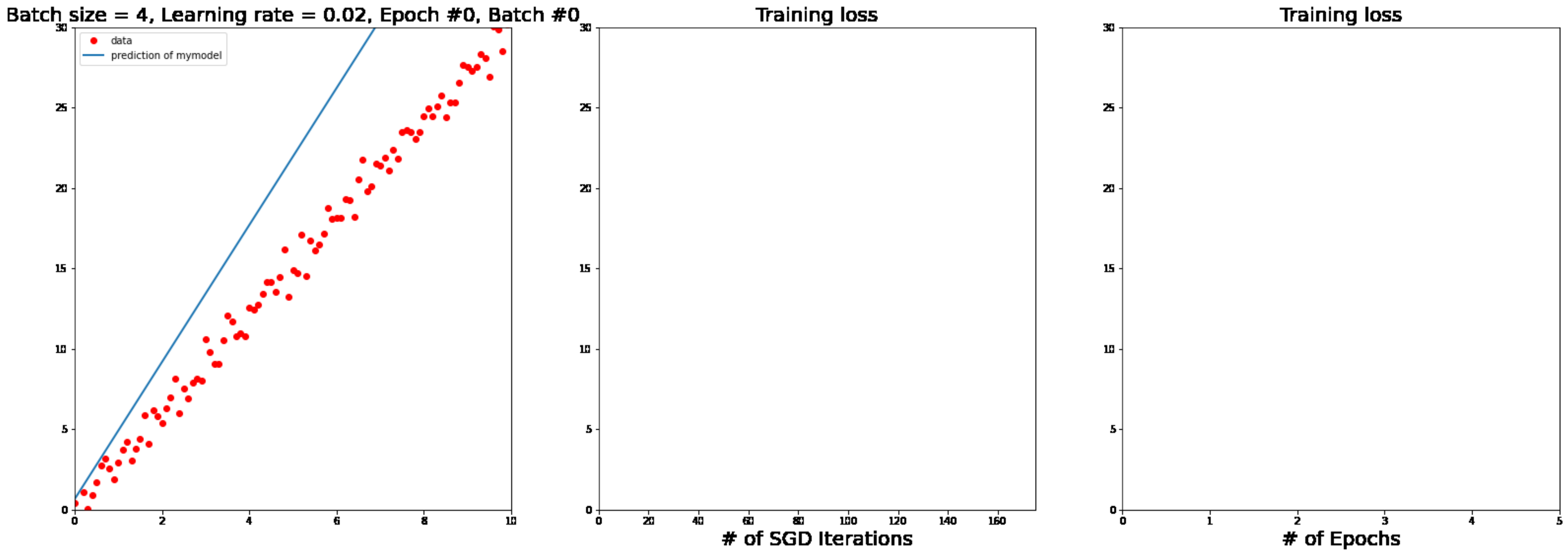
Visualization of SGD



A lot of randomness! How to reduce it?

Visualization of SGD

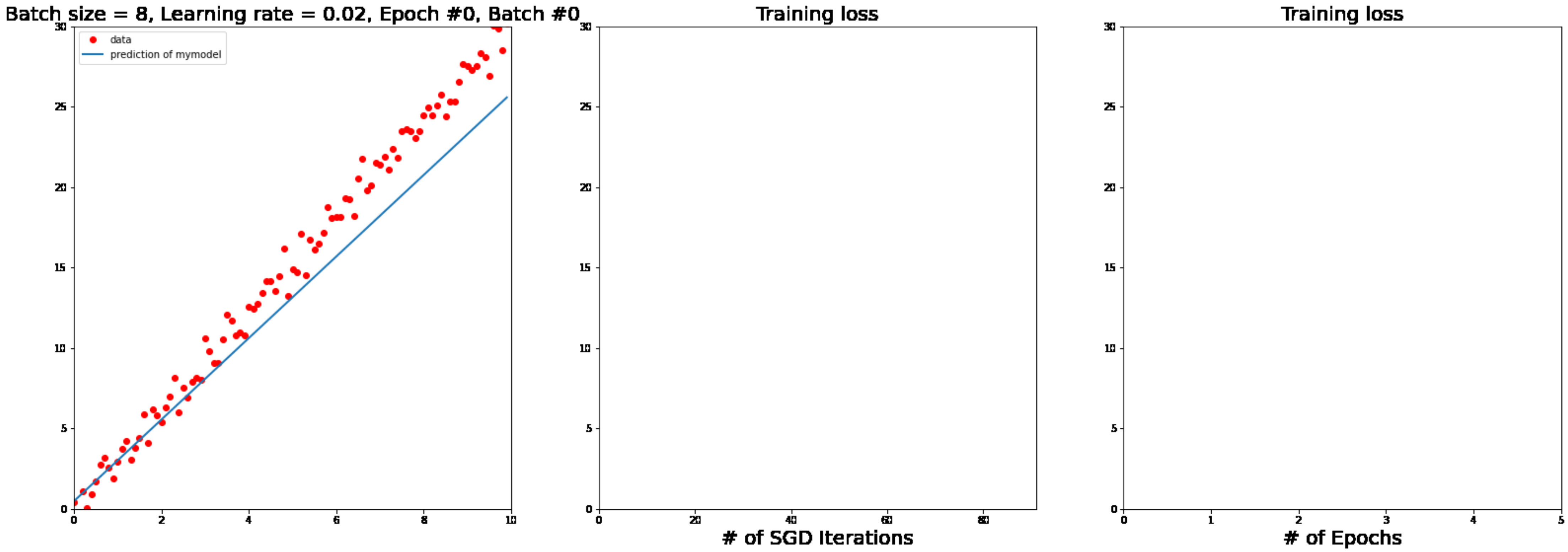
Increase batch size to 4



Randomness and oscillations become smaller

Visualization of SGD

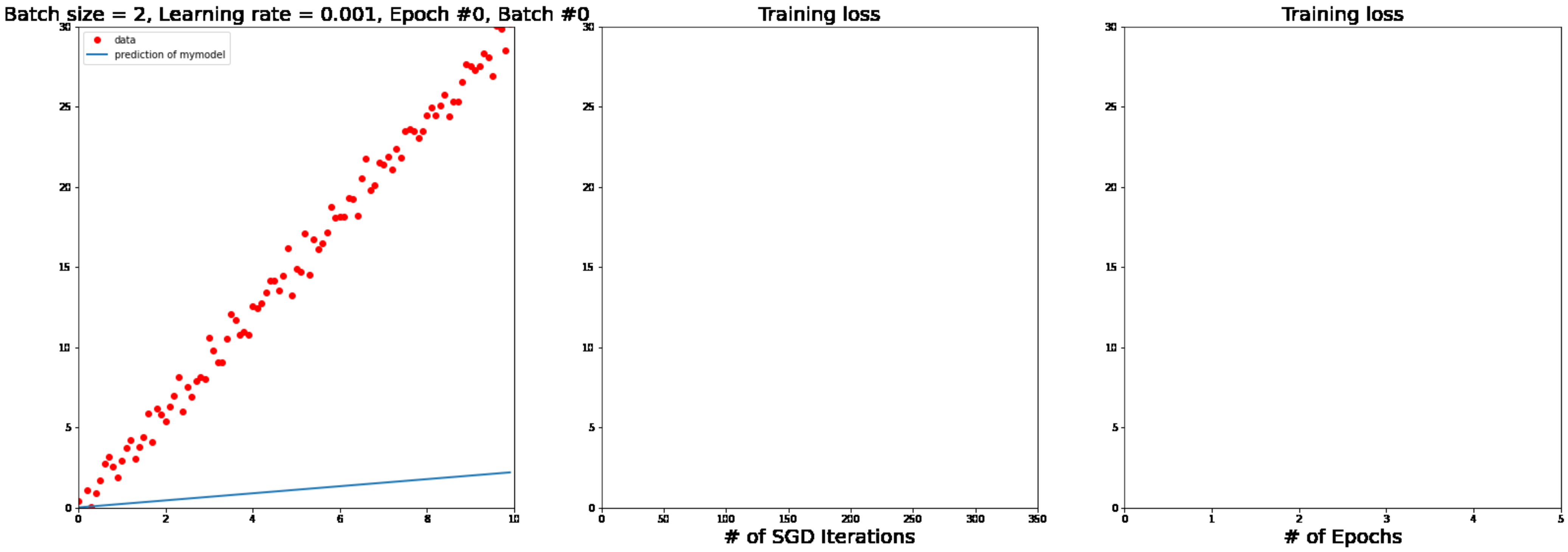
Increase bath size to 8



Randomness and oscillation becomes smaller

Visualization of SGD

Still batch size 2, but smaller learning rate



Randomness and oscillations quite small!

Lessons Learned

- SGD will reduce the computation cost for each gradient step
- But will bring randomness in the training process
  - Larger batch size will reduce randomness
  - Smaller learning rate will also reduce randomness

- In deep learning, batch size is typically around 32, 64 (related to GPU acceleration), and we won’t tune the batch size that much.
- We tune the learning rate more heavily

Up Next: use built-in layers and build neural networks!

Built-in Linear layer

```
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 functi
        # we usually create variables for all our model parameters (w and b in
        # need to create them as nn.Parameter so that the model knows it is an
        self.w = nn.Parameter(torch.zeros(1,d, dtype=torch.float32))
        self.b = nn.Parameter(torch.zeros(1,dtype=torch.loat32))
    def forward(self,x):
        # The main purpose of the forward function is to specify given input x,
        return torch.inner(x,self.w) + self.b
```

We manually code the mathematical operations for the linear model y=wx+b

Built-in Linear layer

```python
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
```

Define a built-in layer

Use the linear layer to compute the output

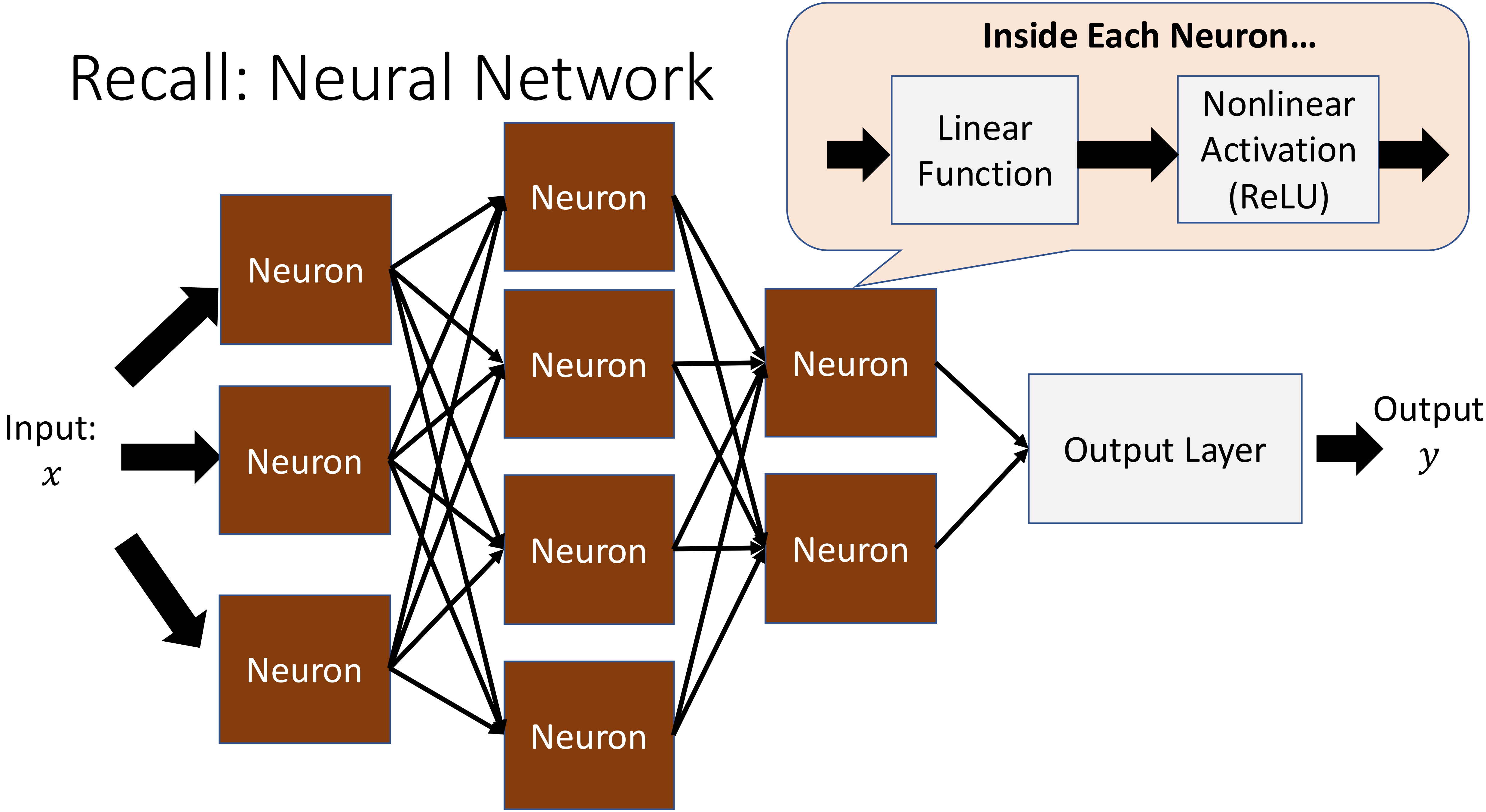
Benefits of using built-in layers:

- No need to define parameters, their initializations
- No need to explicitly code the "mathematical operations"
- Would be very convenient to build neural network models

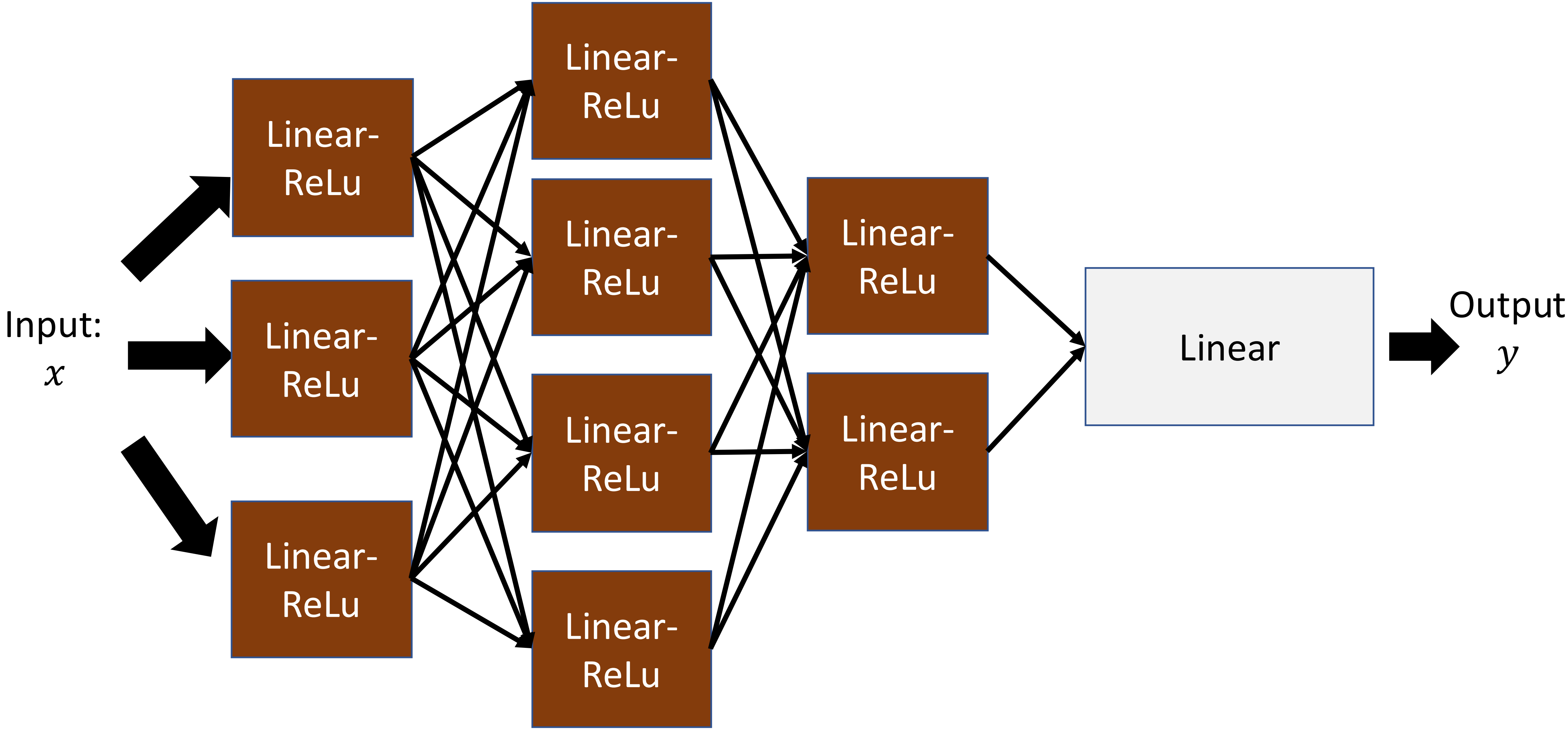
Building Neural Network in PyTorch

1. Define a Neural Network model
2. Generate some training data
3. Calculate gradient and conduct gradient descent

Recall: Neural Network

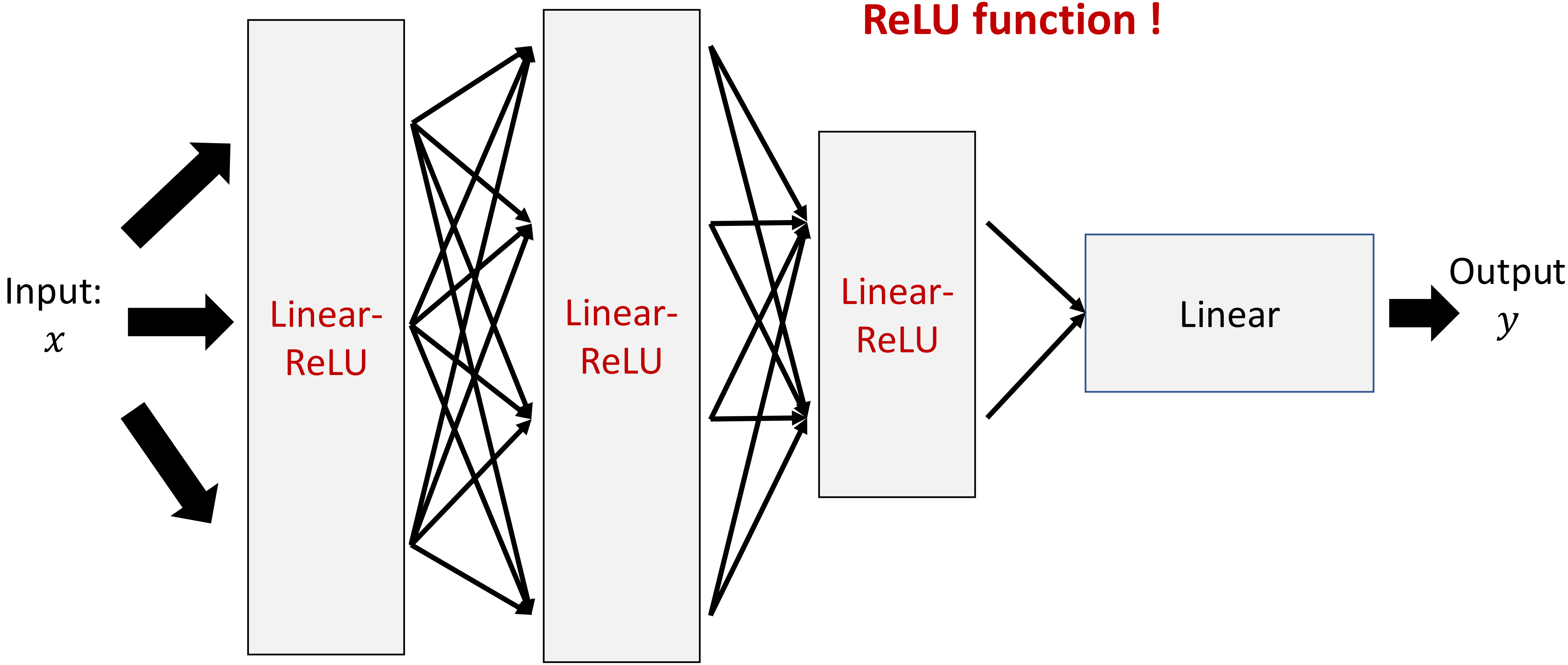


Recall: Neural Network



Recall: Neural Network

Each layer is a connection of a (multi-dimensional) linear and ReLU function !



Build Neural Network in PyTorch

```python
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
```

Overall, we create a "Sequential" of layers

The first layer with width 20

- A Linear followed by a ReLU
- Need to specify input and output dim for Linear
  - Input dimension is input_dim (user specified)
  - Output dimension is 20 (width of this layer)

Build Neural Network in PyTorch

```python
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
```

Overall, we create a “Sequential” of layers

The first layer with width 20

The second layer with width 20

The third layer with width 20

The fourth layer with width 20

The output layer

Build Neural Network 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
```

The forward method just uses the sequential we created to compute the output.

Build Neural Network in PyTorch

```
mymodel = myMultiLayerPerceptron(1,1) # creating a model instance

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)
  )
)
```

Which one is correct?

```python
class myMultiLayerPerceptron_1(nn.Module):
    def __init__(self,input_dim,output_dim):
        super().__init__()
        self.sequential = nn.Sequential(  # h
            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)
        )
```

A

```python
class myMultiLayerPerceptron_2(nn.Module):
    def __init__(self,input_dim,output_dim):
        super().__init__()
        self.sequential = nn.Sequential(  # he
            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)
        )
```

B

Build Neural Network 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
```

The input dimension of each layer must match the output dimension of the previous layer!

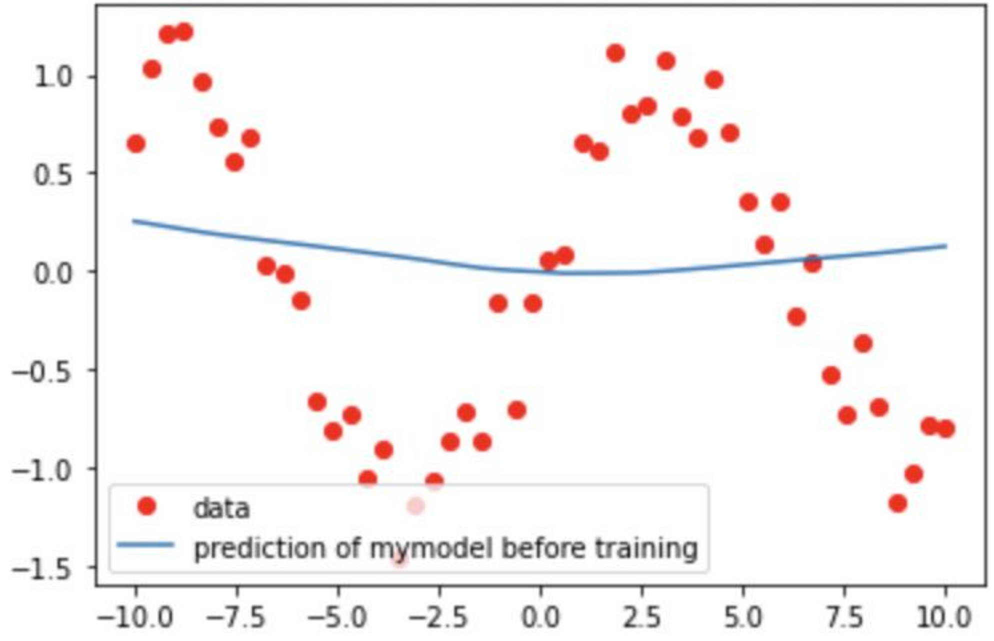
Building Neural Network in PyTorch

1. Define a Neural Network model
2. Generate some training data
3. Calculate gradient and conduct gradient descent

```
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'])
```



Building Neural Network in PyTorch

1. Define a Neural Network model
2. Generate some training data
3. Calculate gradient and conduct gradient descent

Up next!

```
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])
```

Training Loops

mymodel is now the neural network we just defined

```
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
```

Three hyper parameters

```
# 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,
```

DataLoader and Optimizer

Training Loops

The training loop is identical as before!

```
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()
```

Forward pass

Backward pass and compute gradient.

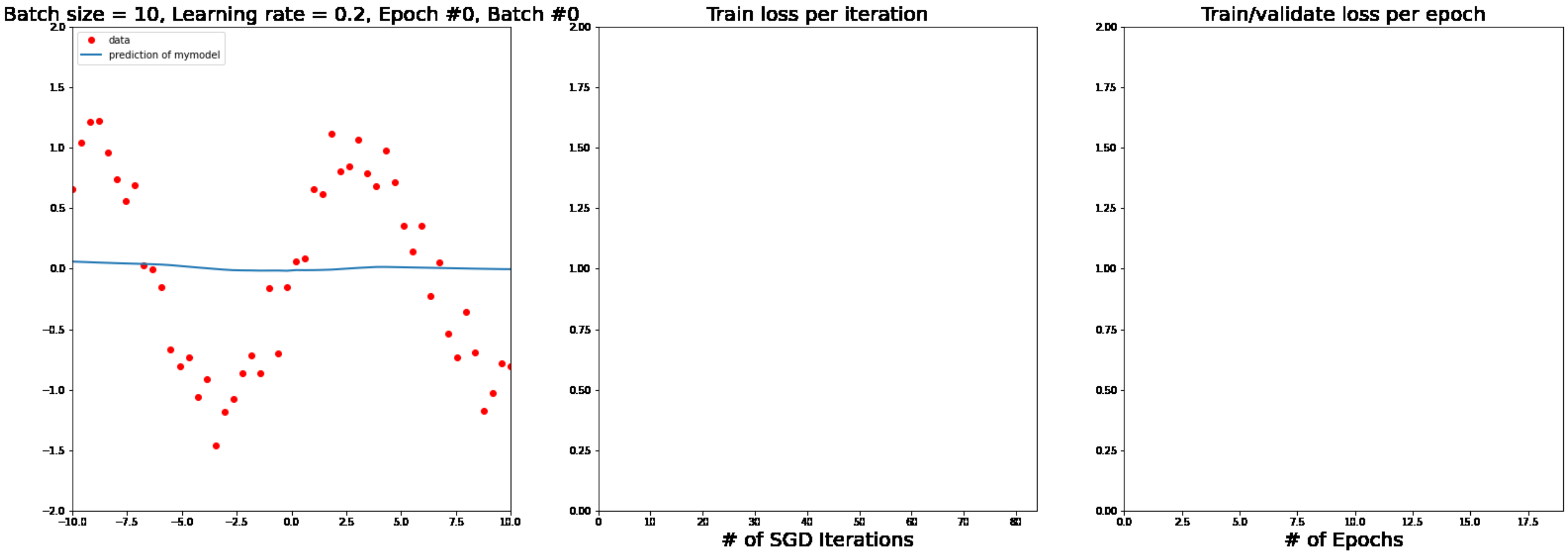
Run a gradient descent step

Building Neural Network in PyTorch

1. Define a Neural Network model
2. Generate some training data
3. Calculate gradient and conduct gradient descent

The main difference compared to linear regression is how we defined the model
The way to calculate gradient and the training loops are (almost) identical

Let’s now visualize the training process and tune some hyper parameters!
Initial parameter values:
- Learning rate = 0.2
- N_epochs = 20
- Batch_size = 10



How to make it better? Hyper Parameter tuning (next lecture)!