Introduction to PyTorch

Lecture 11 for 14-763/18-763

Guannan Qu

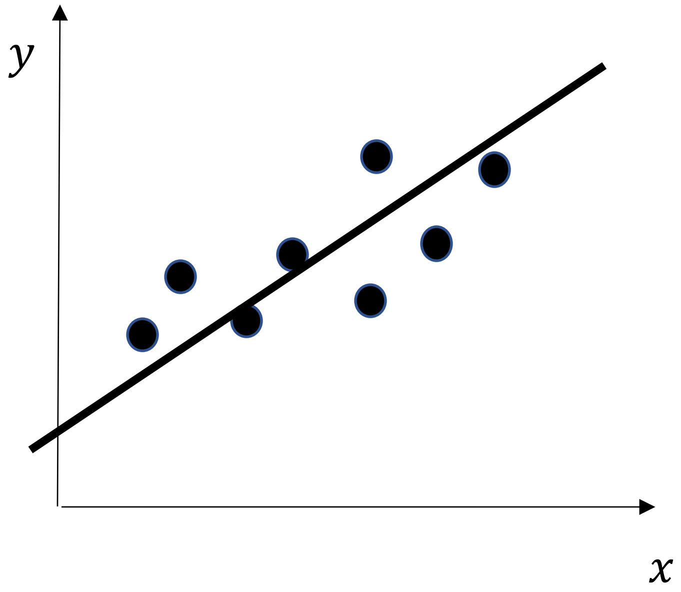
Oct 7, 2024

Traditional ML vs Deep Learning

Traditional ML suffers from some issues including:
- Not good at handling high dimensional data (e.g. image and texts).
    - For a 32*32 image, # of input features is 1024
    - For a paragraph of texts, can be hundreds of words
- Need to do feature extraction (like Fourier Transform) which is difficult

Deep Learning is capable of
- Handling high dimensional data (image, texts)
- No need to do feature extraction
    - Feature extraction is done automatically in deep learning.

What is Deep Learning?

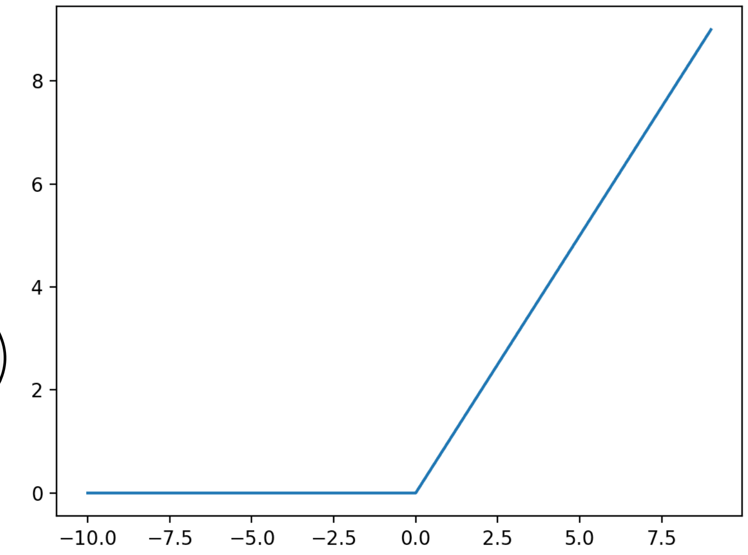


Recall: linear regression

Input: x → Linear Model $y = ax + b$ → Output y

Deep learning replaces the linear model
with “layers” of linear models with non-linear activation!

What is Deep Learning?



A.k.a. activation function.
A popular choice is the ReLU
$f(y') = \max(y', 0)$

Input:
x

Linear Function
$y' = \sum_{i=1}^{n} w_i x_i + b$

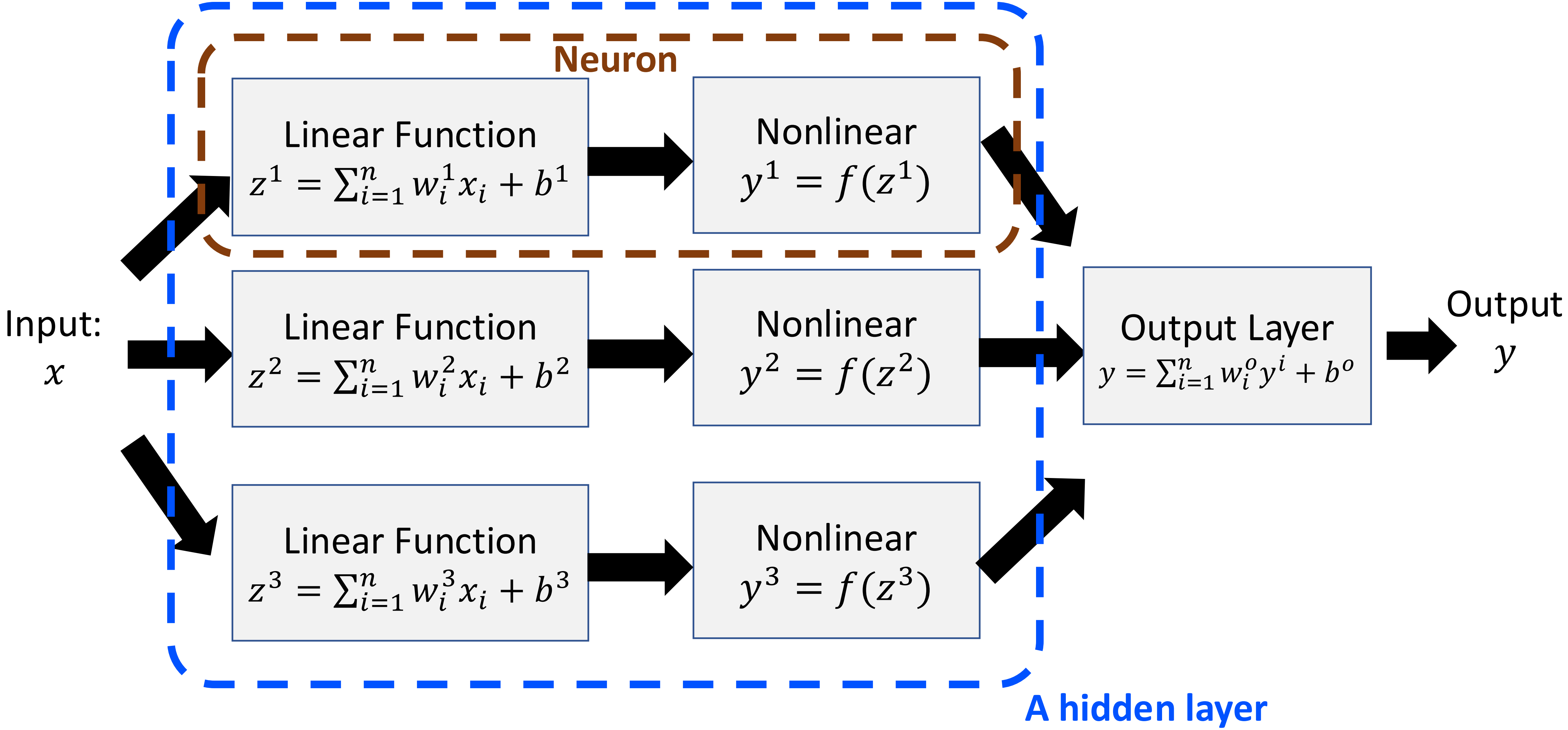
Nonlinear
$y = f(y')$

Output
y

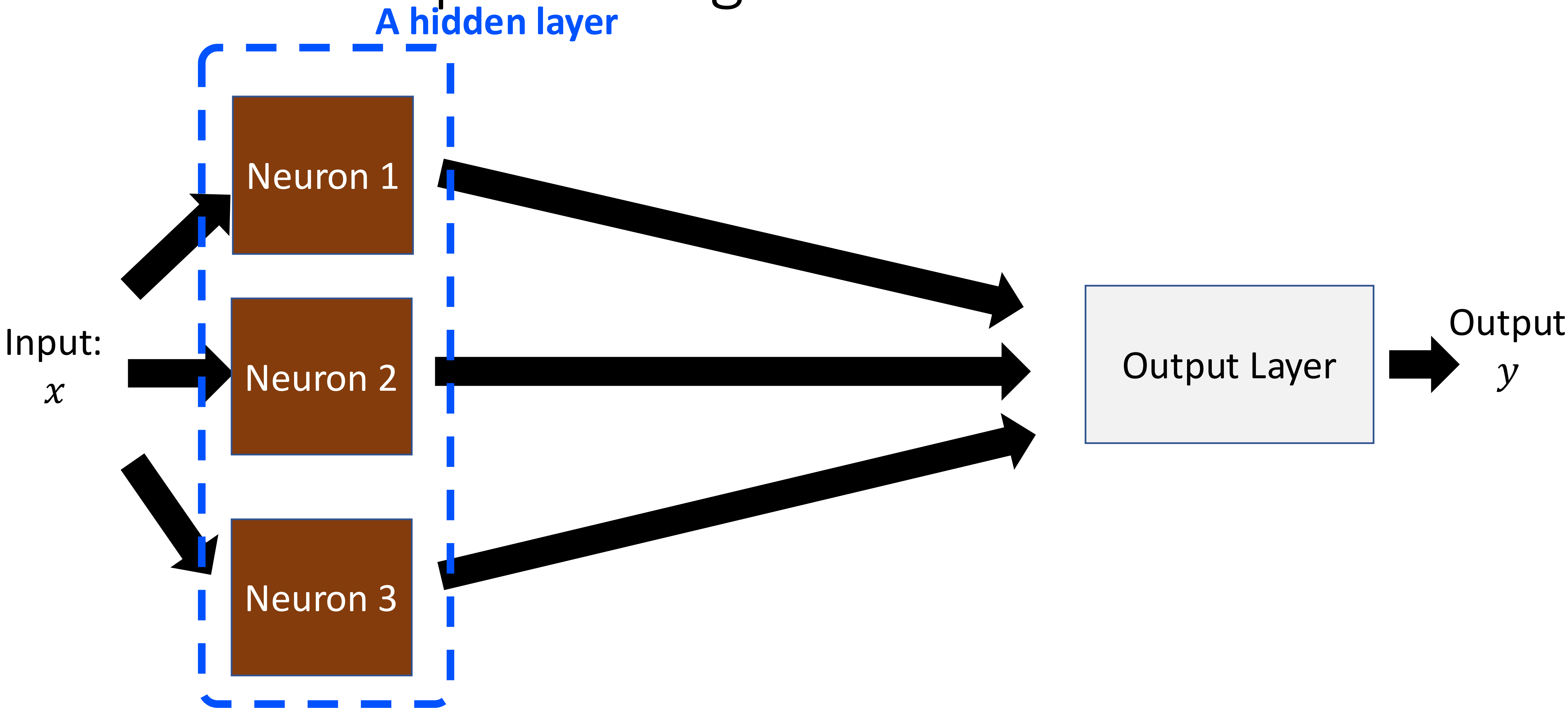
In deep learning, the input is typically high-dimensional
$x = [x_1, \ldots, x_n]$

This is a neural network with 1 layer with width 1
Next: increase the width

What is Deep Learning?

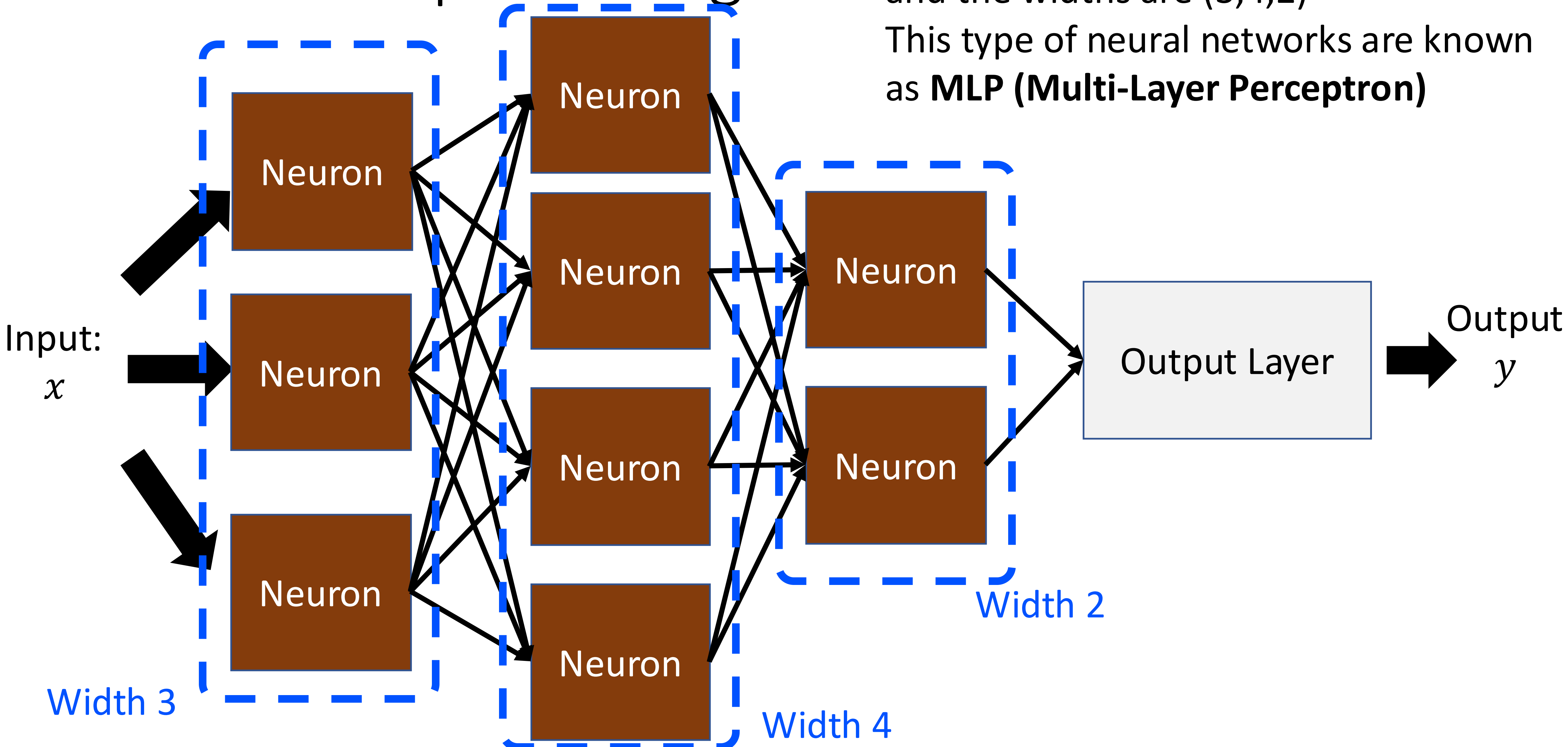


What is Deep Learning?



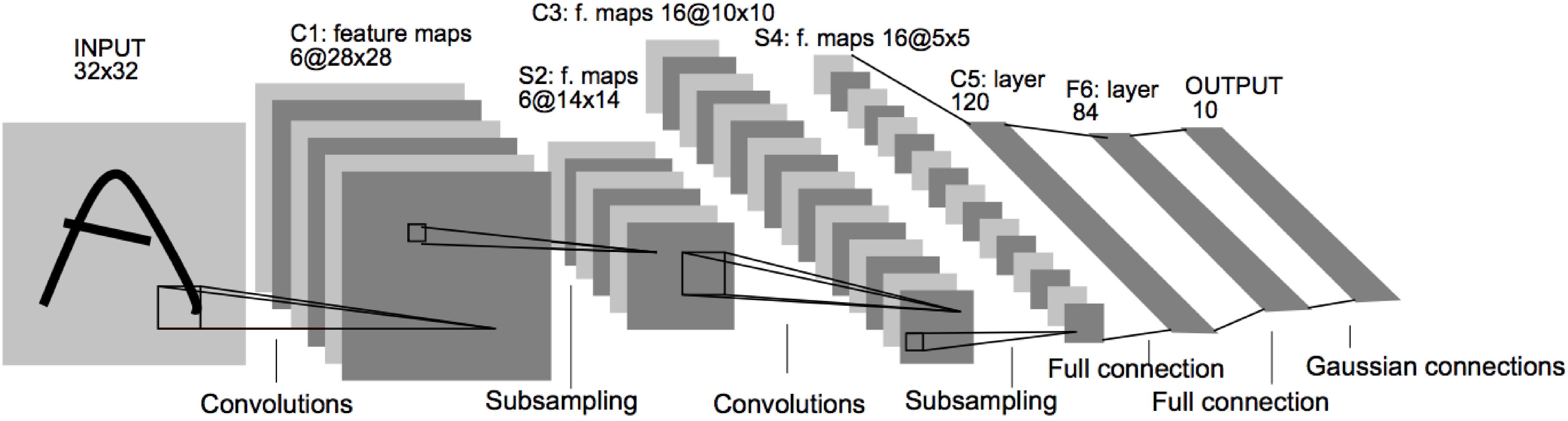
What is Deep Learning?

A neural network with 3 hidden layers and the widths are (3,4,2)
This type of neural networks are known as **MLP (Multi-Layer Perceptron)**



Common NN structures

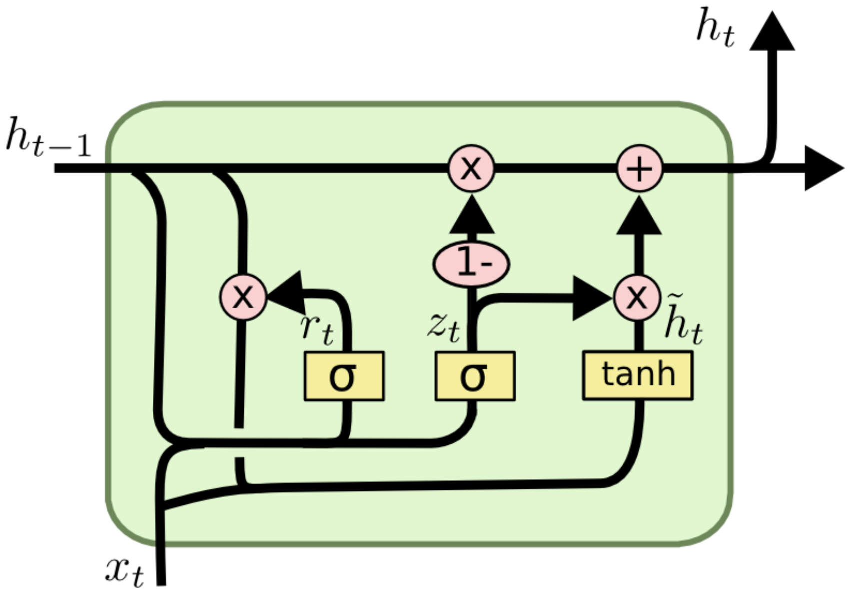
LeNet-5 (Convolution NN)



https://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

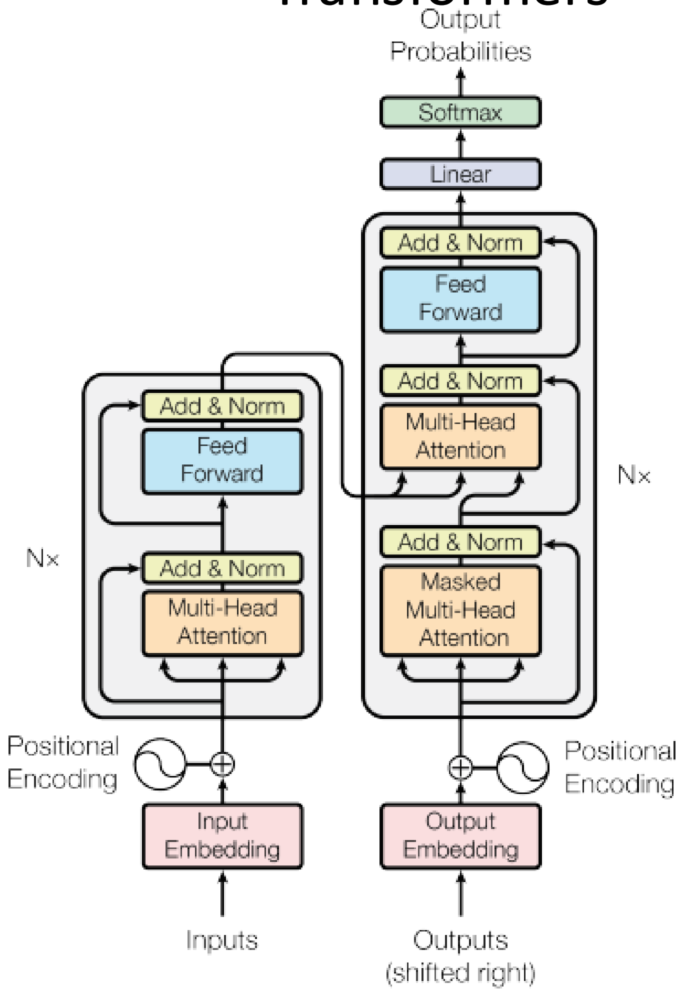
Common NN structures

LSTM (Recurrent NN)



https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Transformers



https://arxiv.org/abs/1706.03762

What is Deep Learning?

(Deep) Neural Networks are a type of ML model:

- Use a cascade of multiple layers of nonlinear processing units (neurons). Each successive layer uses the output from the previous layer as input.
- Has a long history, perhaps first dates back to 1943, but limited success until the 2000s
- Become extremely successful in the 2010s in various domains (image classification, NLP…)
- Various architectures, Multi-Layer Perceptron (MLP), Convolutional NN (CNN), Recurrent NN (RNN), Transformer…

Deep learning requires new ML platform

SparkML (based on Transformer/Estimator) is not adequate for deep learning

- Deep neural networks has a highly flexible structure
  - # layers, # of neurons for each layer, choice of activation function
  - CNN, RNN, ResNet has more complicated structure
- The training of neural network requires a lot of tuning, and therefore needs to get to the low-level detail
- The training of neural network is data intensive and computationally heavy

We need specialized ML platform for deep learning!

What is TensorFlow (v1)?

- History: Developed by the **Google Brain Team** to accelerate deep neural network research, **TensorFlow v1** was first made public in late 2015
- Built to run on multiple CPUs or GPUs and even mobile operating systems.
- Multiple languages like Python, C/C++ or Java.
- End-to-end, free, and open source. Early dominant player in deep learning.

TensorFlow v1 was the early dominant player

2015

PyTorch

- **PyTorch** was released in 2017 by Facebook AI (now Meta) and soon became popular
    - Known for its simplicity, ease of use, flexibility
    - Uses dynamic computation graph
- In contrast, **TensorFlow v1** at the time
    - Not user friendly, steeper learning curve, not well organized
    - Used static computation graph
    - But TensorFlow still had advantages, e.g. in deployment, visualization

TensorFlow v2

The comparison became more complicated when TensorFlow 2 was released in 2019

- TensorFlow 2 became much more user friendly and the APIs were cleaned up
- However, many compatibility issues remained. Code written in TensorFlow v1 cannot easily migrate to v2, frustrating many users
- From 2019 onwards, more and more people (especially in research) switch from tensorflow to PyTorch, but TensorFlow still remain relevant in industry.

TensorFlow v1 was the early dominant player

TensorFlow v2 was released to improve ease-of-use but many compatibility issues

2015

2017

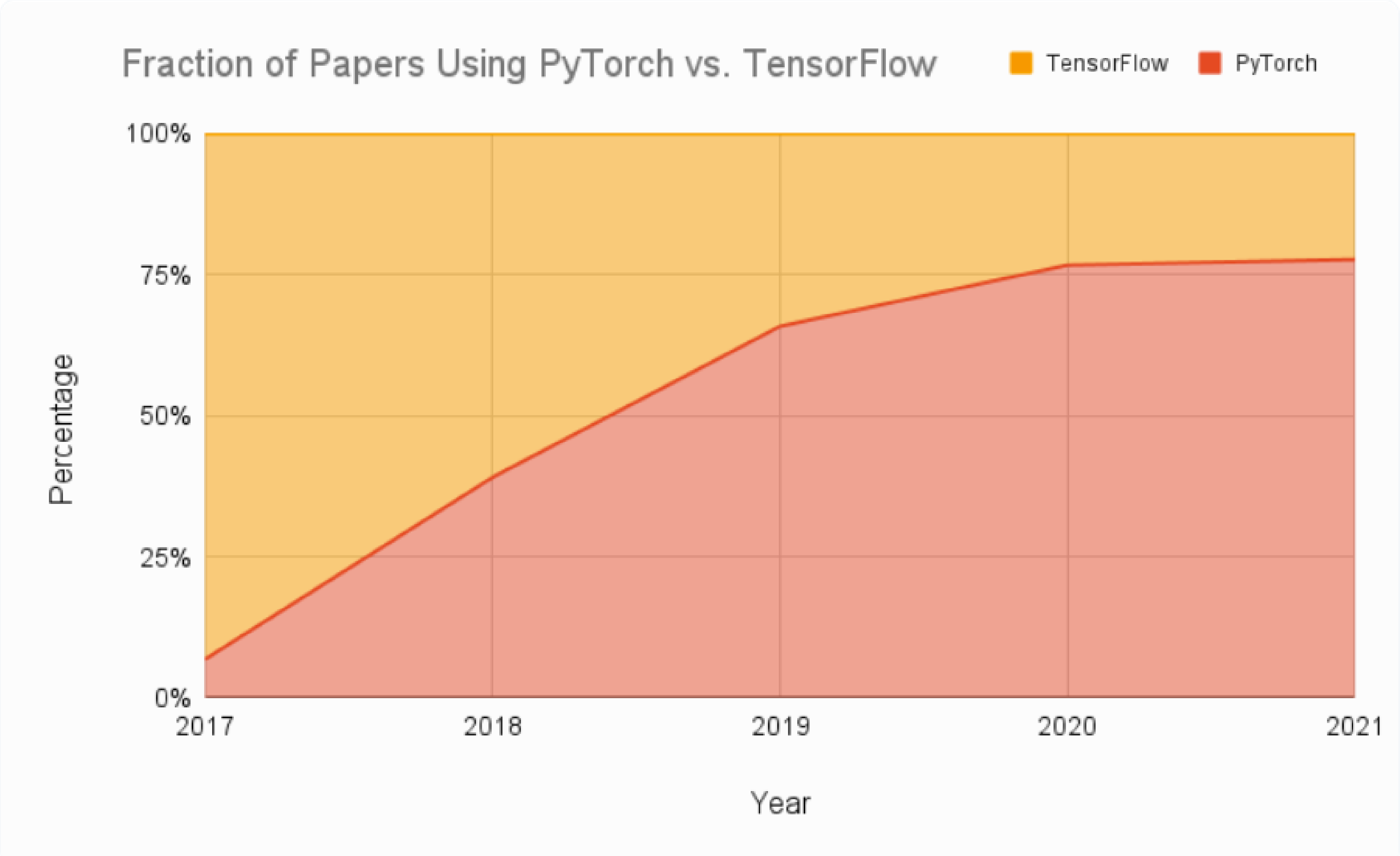
2019

2019-2021

PyTorch was released popular due to ease of use

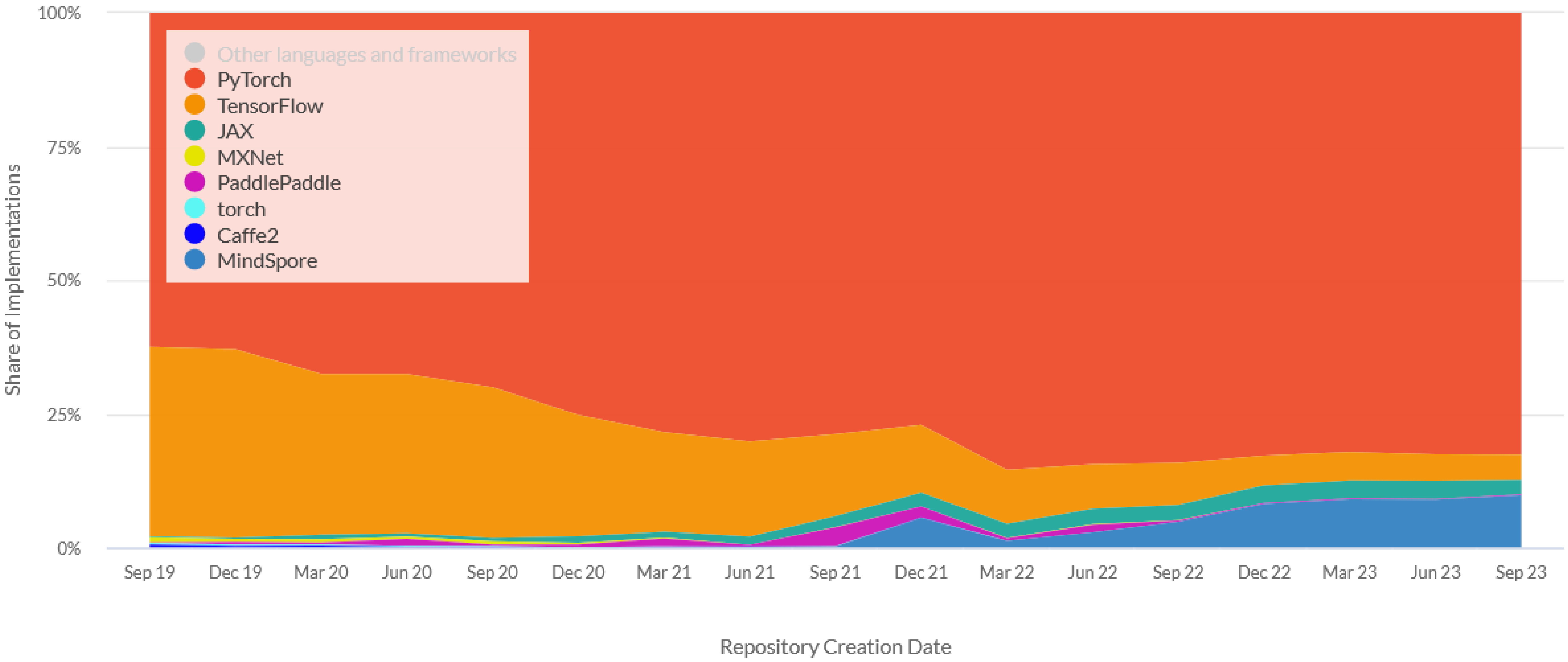
TensorFlow was gradually surpassed by PyTorch, especially in research

TensorFlow vs PyTorch



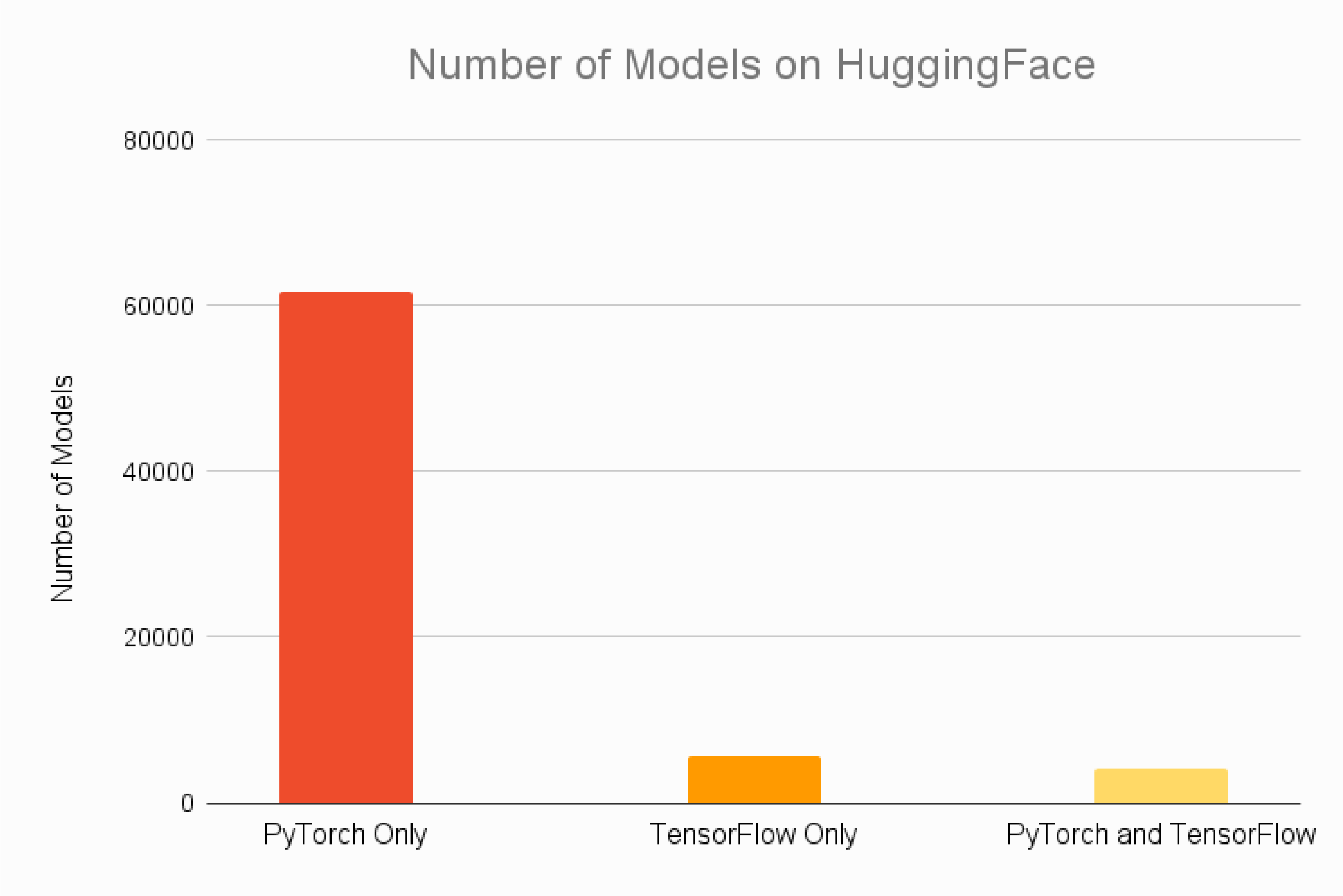
https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/

TensorFlow vs PyTorch



https://viso.ai/deep-learning/pytorch-vs-tensorflow/

TensorFlow vs PyTorch



https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/

Introduction to PyTorch

Today:

- Use linear regression as warm-up, but go to low level details this time
- Understanding ML: loss function and optimization

This Wed (Oct 9): SGD and Neural Networks

Oct 21: hyper-parameter tuning and best practices

Oct 23: computation graph and GPU

Oct 28: distributed training

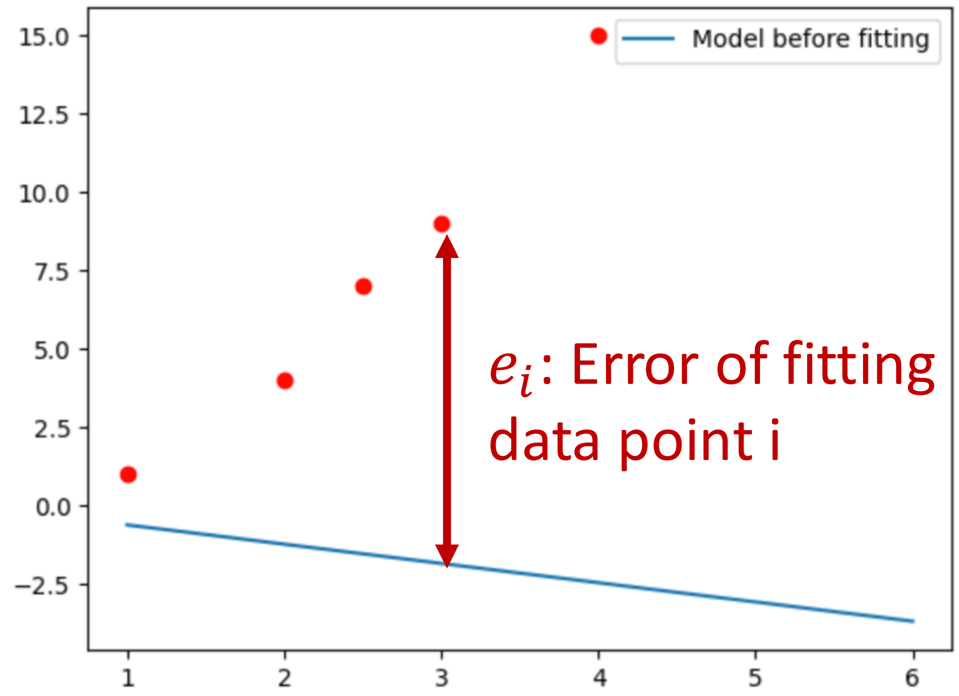
Oct 30: ML Ecosystem

Loss Function

Linear Model: $y = wx + b$

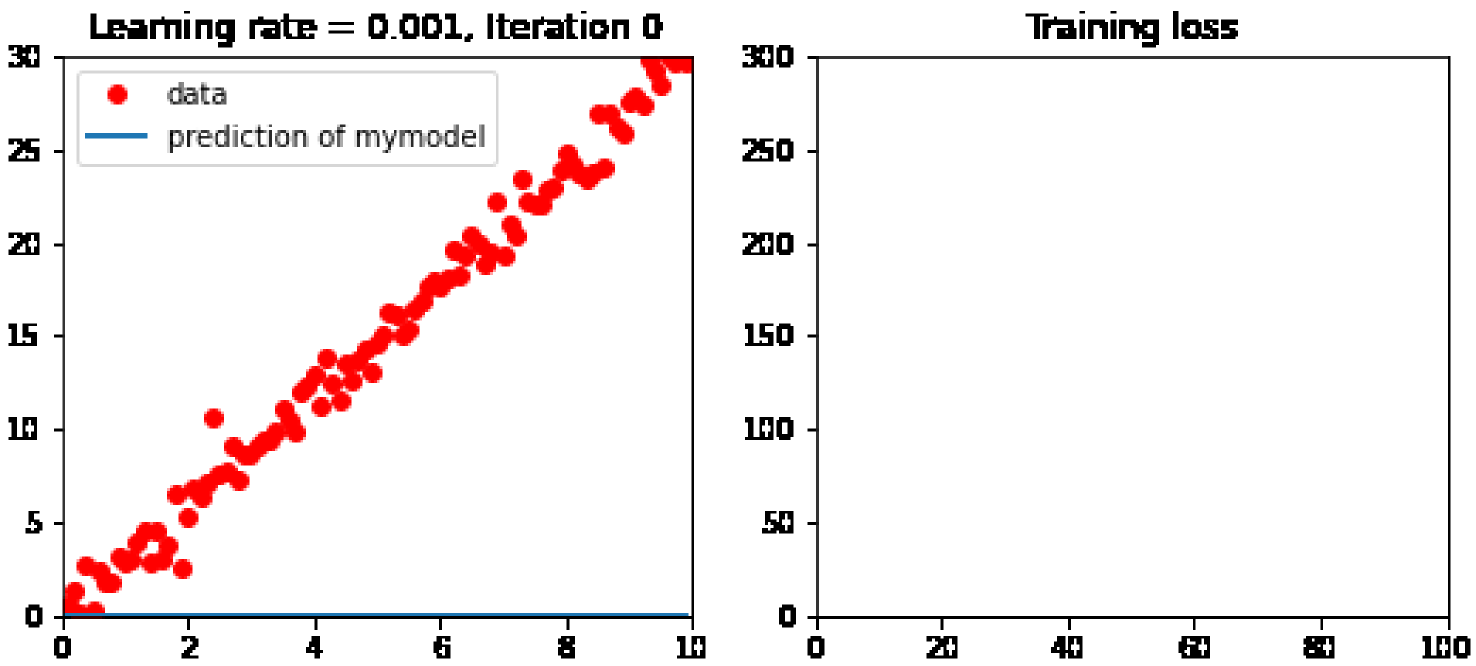
Model Parameters: w, b

$$loss(w, b) = \frac{1}{N} \sum_i \underbrace{\left(y_i - (wx_i + b)\right)^2}_{e_i}$$



The training/fitting process finds the w,b with the smallest loss, but how?

How does training work?



The training/fitting process finds the w,b with the smallest loss, but how?

Optimization: Gradients

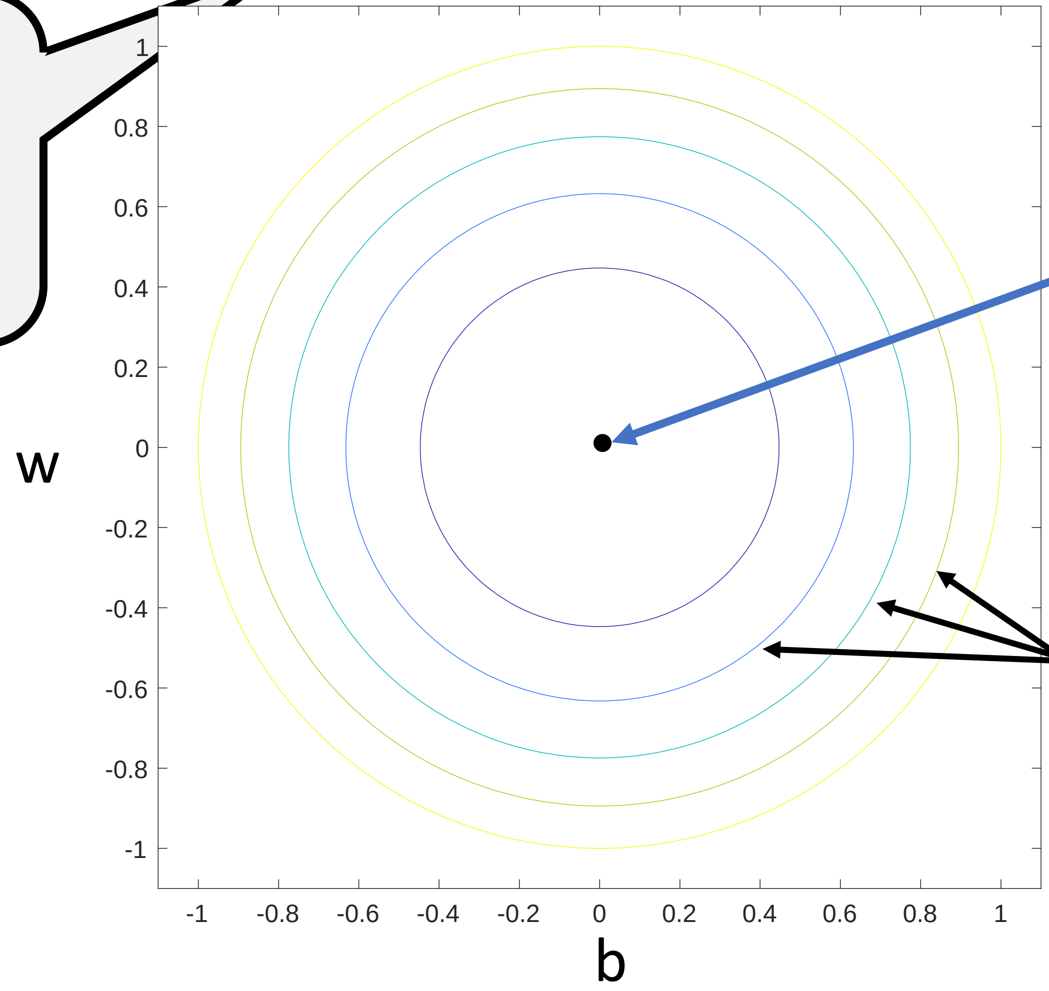
Given a function $loss(\theta)$ that depends on a 2 dimensional $\theta = [w, b]$

Its gradient $\nabla loss(\theta)$ is the direction from θ that will lead to largest increase in $loss(\theta)$

Optimization: Gradients

Think of it as a terrain altitude map, each circle represents the set of θ with the same loss value

Contour Plot of $loss(\theta)$ (Example)



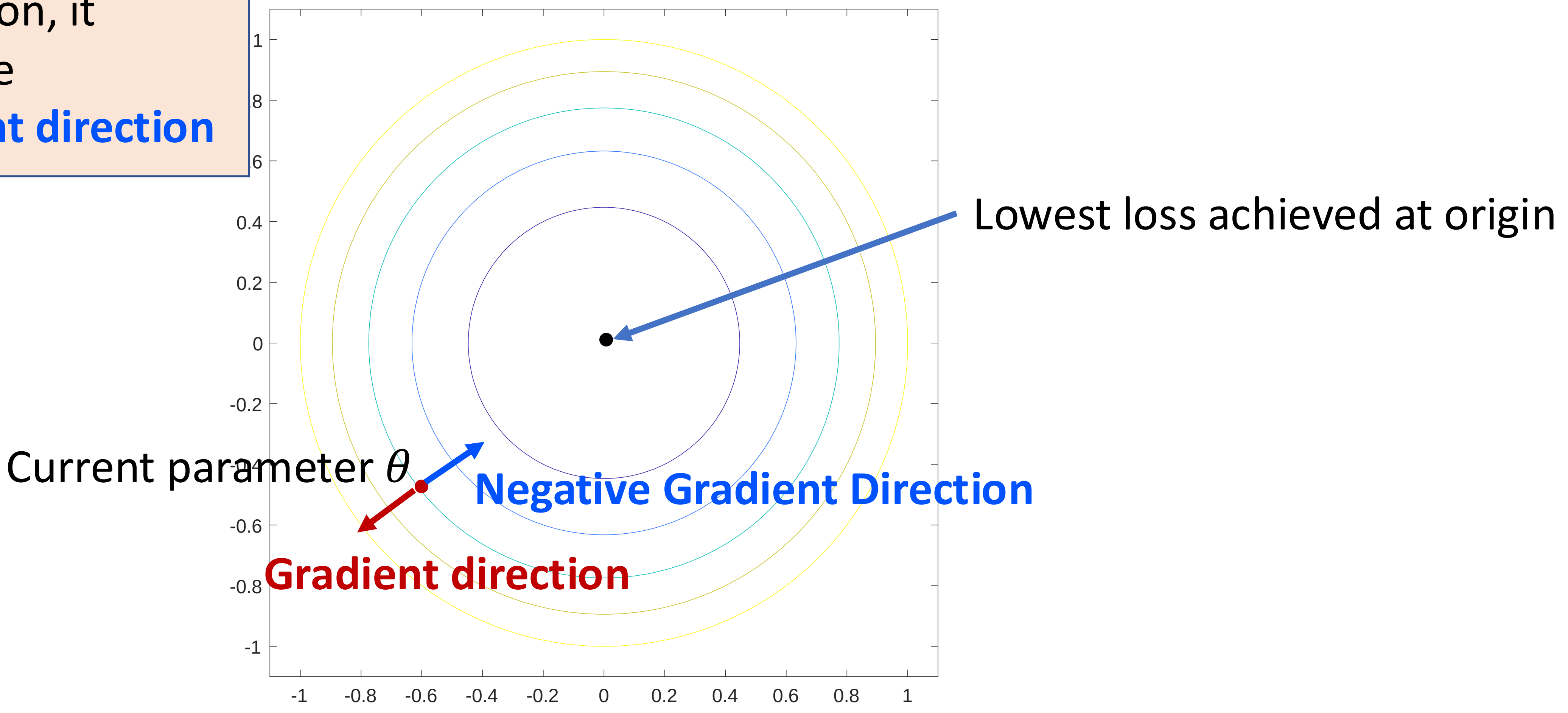
Lowest loss achieved at origin

The farther away from origin, the larger the loss function

Optimization: Gradients

To find a parameter with lower loss function, it should follow the **negative gradient direction**

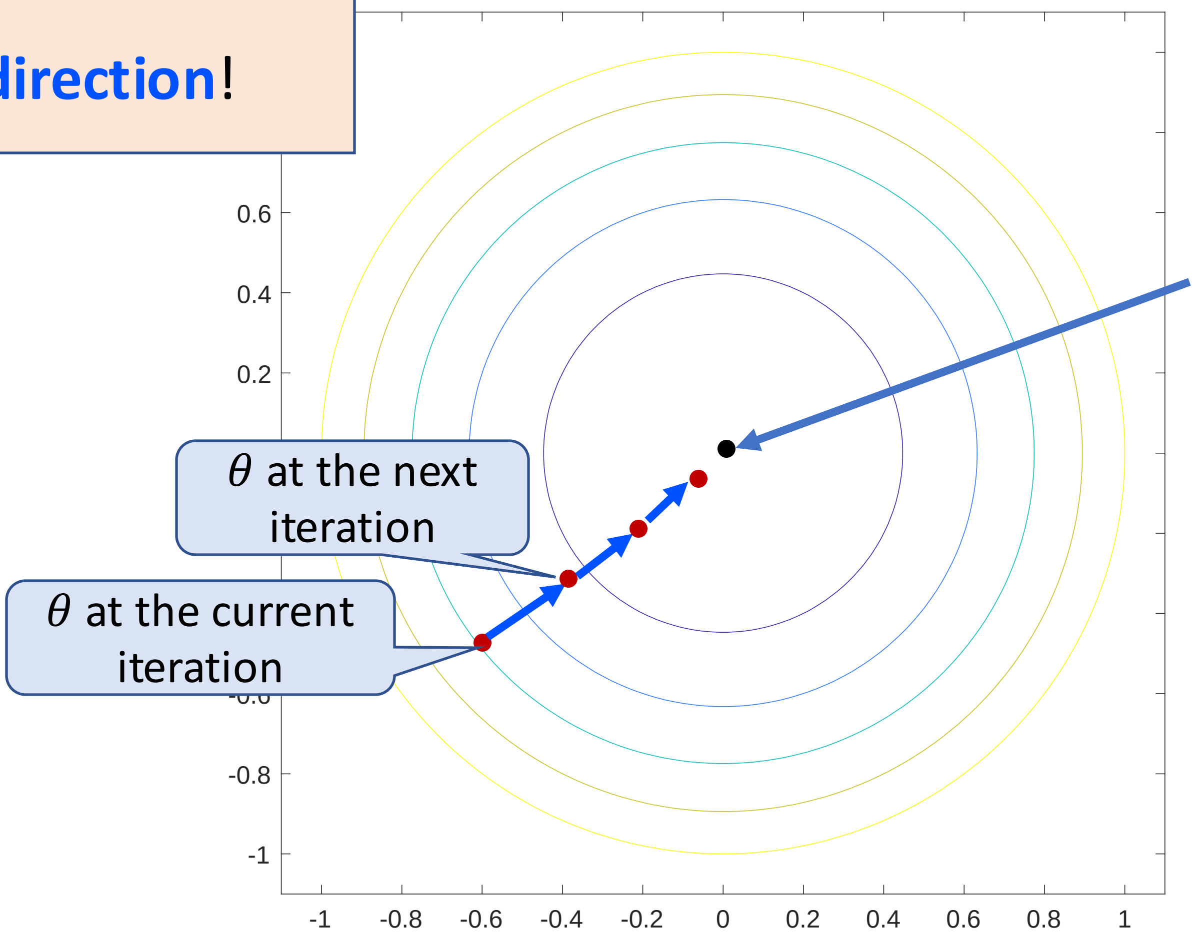
Contour Plot of $loss(\theta)$ (Example)



Optimization: Gradients

Gradient Descent
Keep following the
negative gradient direction!

Contour Plot of $loss(\theta)$ (Example)



Optimization: Gradients

Gradient Descent

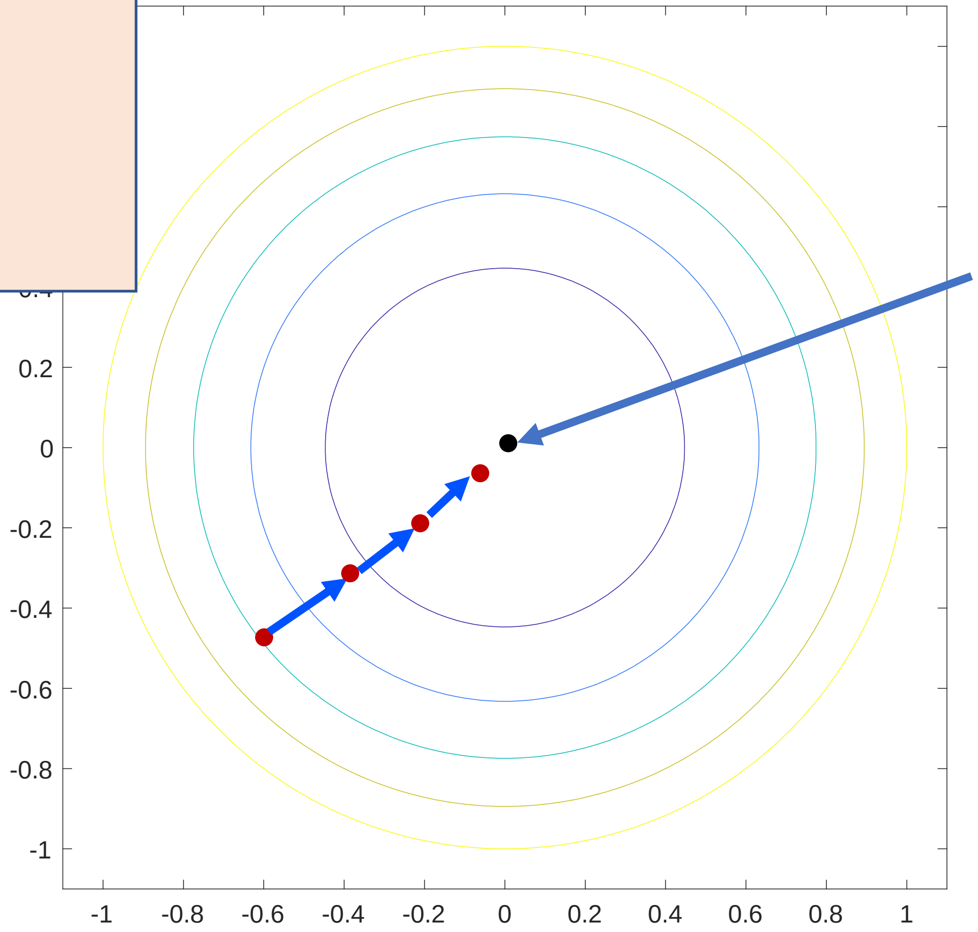
Initialize θ

Repeat **maxIter** steps:

$$\theta \leftarrow \theta - \eta \nabla loss(\theta)$$

η is learning rate, i.e. how large a step one makes in each iteration

Contour Plot of $loss(\theta)$ (Example)



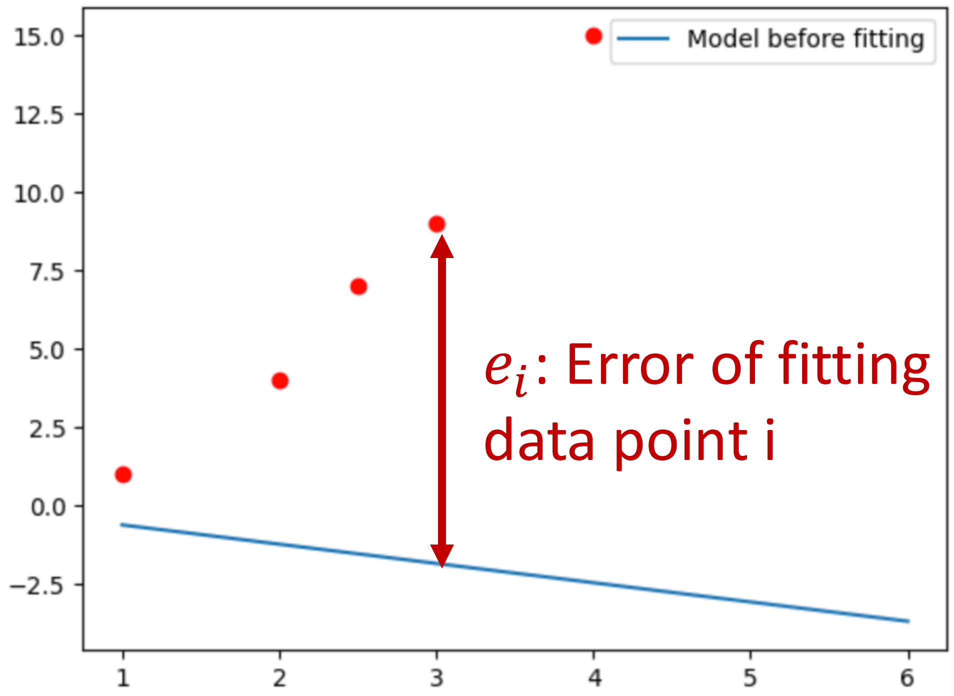
Lowest loss achieved at origin

Optimization: Gradients

Linear Model: $y = wx + b$

Model Parameters: w, b

$$loss(w, b) = \frac{1}{N}\sum_{i}\left(y_i - (wx_i + b)\right)^2$$



How to calculate the gradient of this loss function?

Gradient in PyTorch

The core of PyTorch (and TensorFlow) is their **automatic differentiation (autograd)**

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

PyTorch: Linear Regression Model

Subclassing nn.Module

```
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
```

PyTorch: Linear Regression Model

In __init__ function, define all the parameters of the model as nn.Parameter and give them initial values

```
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))
```

Side note:

- In pytorch, tensor is the most basic building block. nn.Parameter is a special kind of Tensor used to represent model parameters
- In our code, both parameters are initialized as torch.zeros, which are all zero tensors
- Our __init__ function takes d as input, which means the input dimension (we will set d=1)

PyTorch: Linear Regression Model

```
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
```

In forward function, define how the output is computed from input. torch.inner means inner product, and this line of code simply means $w_1 x_1 + \cdots + w_d x_d + b$

PyTorch: Linear Regression 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)
```

```
x = torch.tensor(2)
print(mymodel(x)) # we should expect this to be w*x+b = 0*2+0 = 0 (
```

```
tensor([[0.]], grad_fn=<AddBackward0>)
```

Gradient in PyTorch

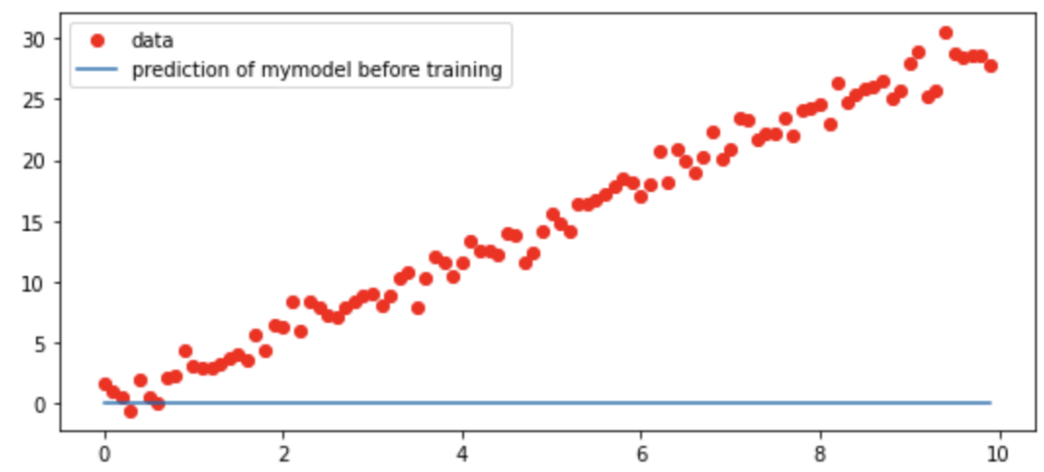
The core of PyTorch (and TensorFlow) is their **automatic differentiation (autograd)**

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

Up Next

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

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



Gradient in PyTorch

Recall: we want to calculate the gradient of this loss function

$$loss(w, b) = \frac{1}{N}\sum_i \left(y_i - (wx_i + b)\right)^2$$

Steps in PyTorch:

- Step 1: Forward Pass, calculate the loss function value

```
prediction = mymodel(x)

loss = torch.mean((prediction - y)**2)

print(loss)
```

✓ 0.1s

```
tensor(296.1156, grad_fn=<MeanBackward0>)
```

Gradient in PyTorch

Recall: we want to calculate the gradient of this loss function

$$loss(w, b) = \frac{1}{N}\sum_{i}\left(y_i - (wx_i + b)\right)^2$$

Steps in PyTorch:

- Step 2: Backward pass.

Before doing backward, let's first check the gradient values now

```
print(mymodel.w.grad,mymodel.b.grad)
✓ 0.6s

None None
```

Gradient in PyTorch

Recall: we want to calculate the gradient of this loss function

$$loss(w, b) = \frac{1}{N}\sum_i \left(y_i - (wx_i + b)\right)^2$$

Steps in PyTorch:

- Step 2: Backward pass.

Let’s now do backward pass and check gradient again

```
loss.backward()
print(mymodel.w.grad,mymodel.b.grad)
```

✓ 0.1s

```
tensor([[-196.8341]]) tensor([-29.6277])
```

Up next: gradient descent, i.e. iteratively compute the gradient and conduct 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()
```

Tell the optimizer what is the parameters to optimize!

Setting learning rate

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.

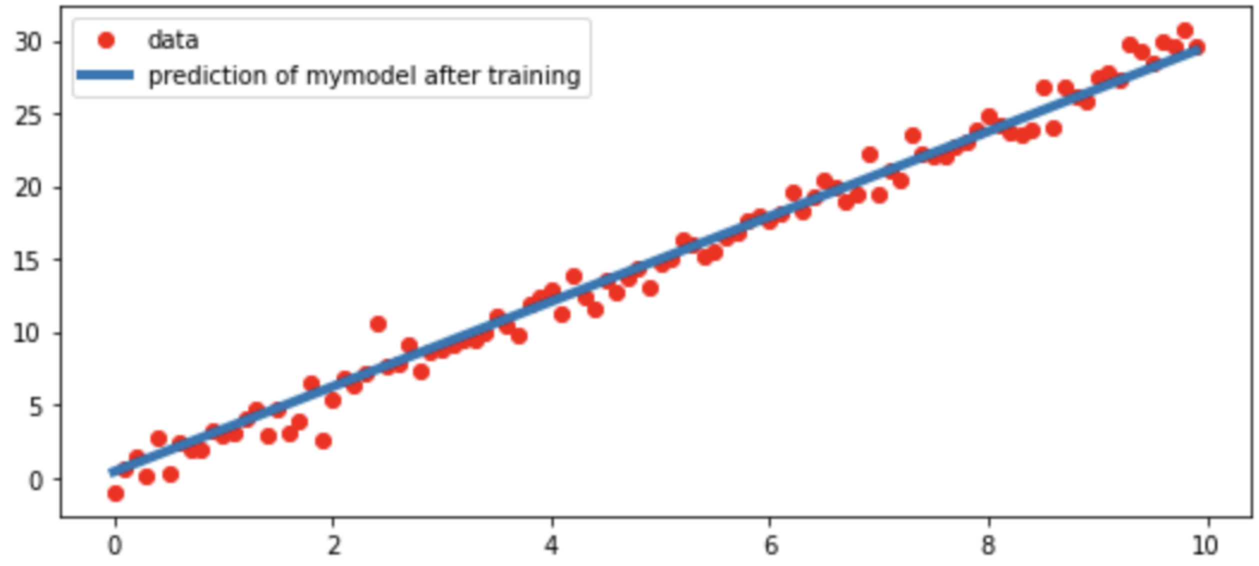
Run a gradient descent on the parameters using the computed gradient and the learning rate

```
print(mymodel.w,mymodel.b)

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

✓ 0.2s

```
Parameter containing:
tensor([[2.9180]], requires_grad=True) Parameter containing:
tensor([0.3993], requires_grad=True)
```



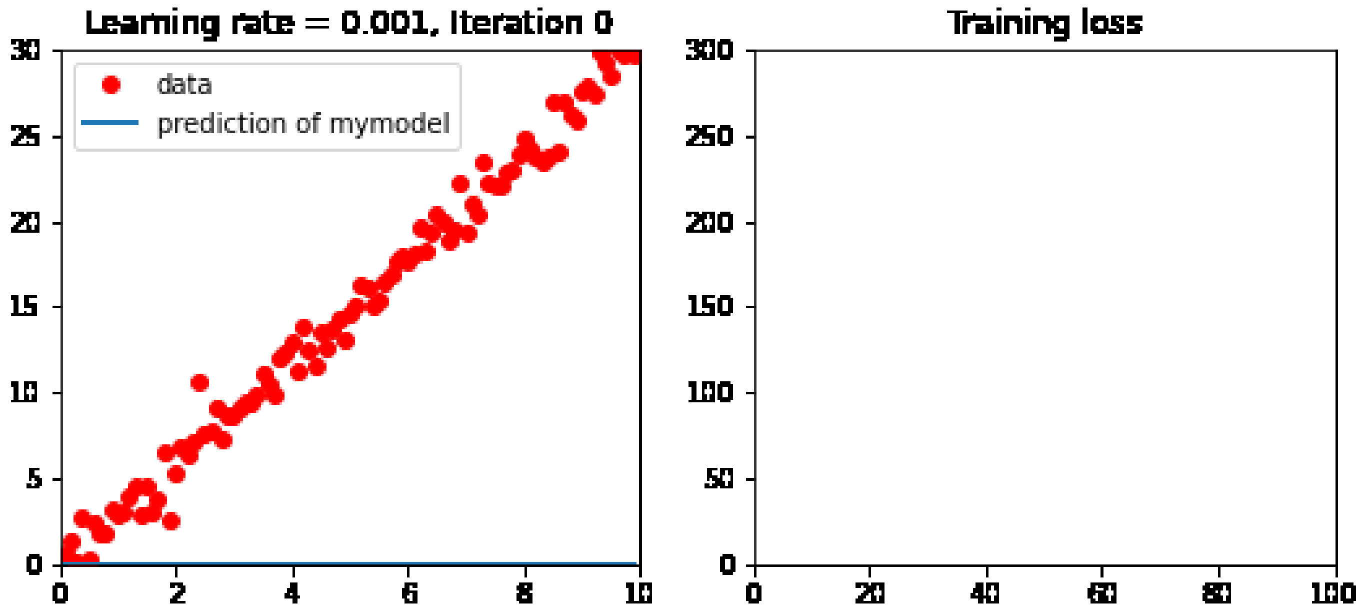
Summary So Far

- Creating a Module: subsclassing nn.Module
  - Define parameters, define forward function
- Calculating gradient
  - Forward and backward pass
- Perform training (gradient descent)
  - Create optimizer and specify the parameters to optimize, and specify learning rate
  - Write training loop that does gradient descent
    - Forward and compute loss
    - Zero-grad
    - Backward
    - Step

How to choose learning rate, maxIter? Let's now visualize the gradient descent process!

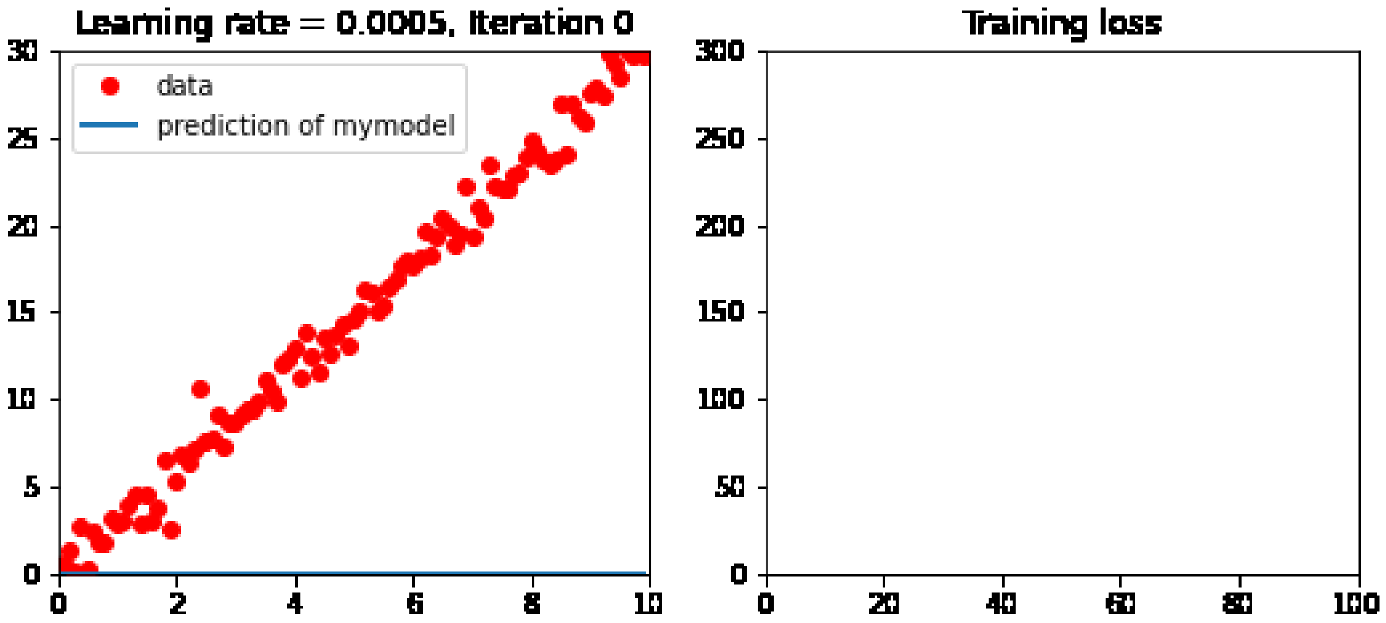
Visualizing Gradient Descent

Learning rate = 0.001



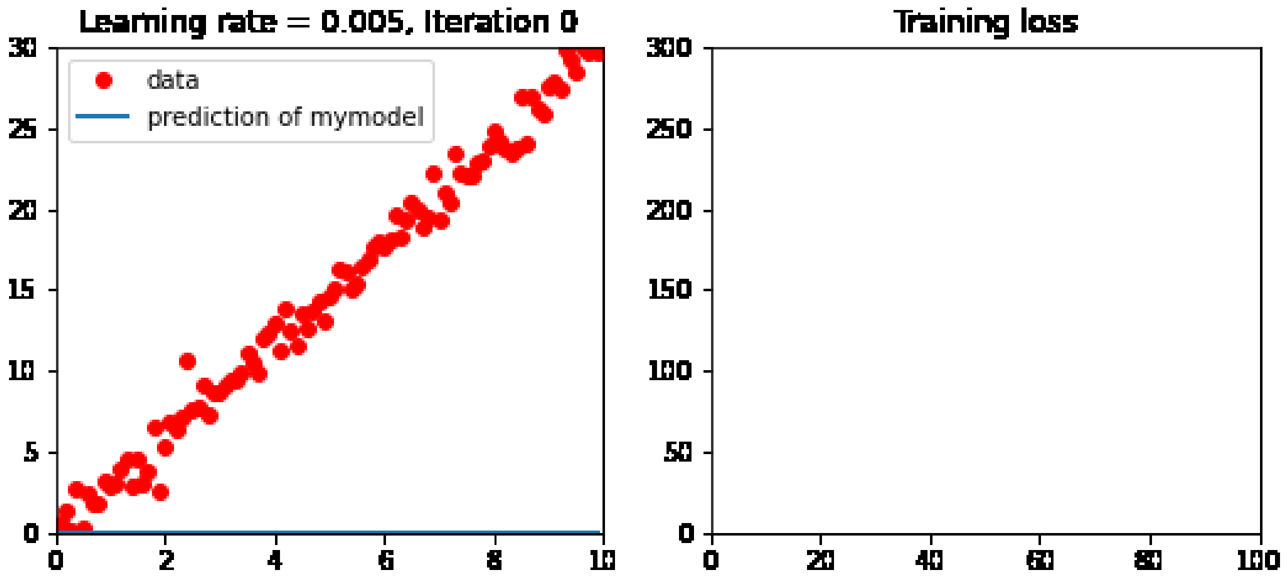
Visualizing Gradient Descent

Learning rate = 0.0005 (smaller than our first trial)



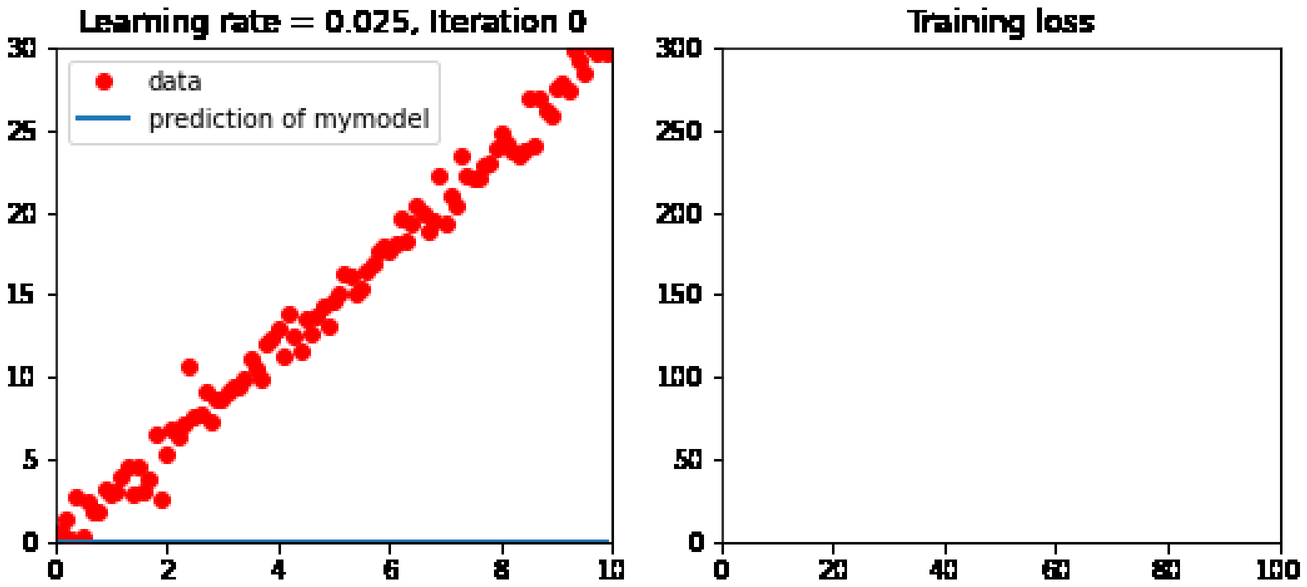
Visualizing Gradient Descent

Learning rate = 0.005 (larger than our first trial)



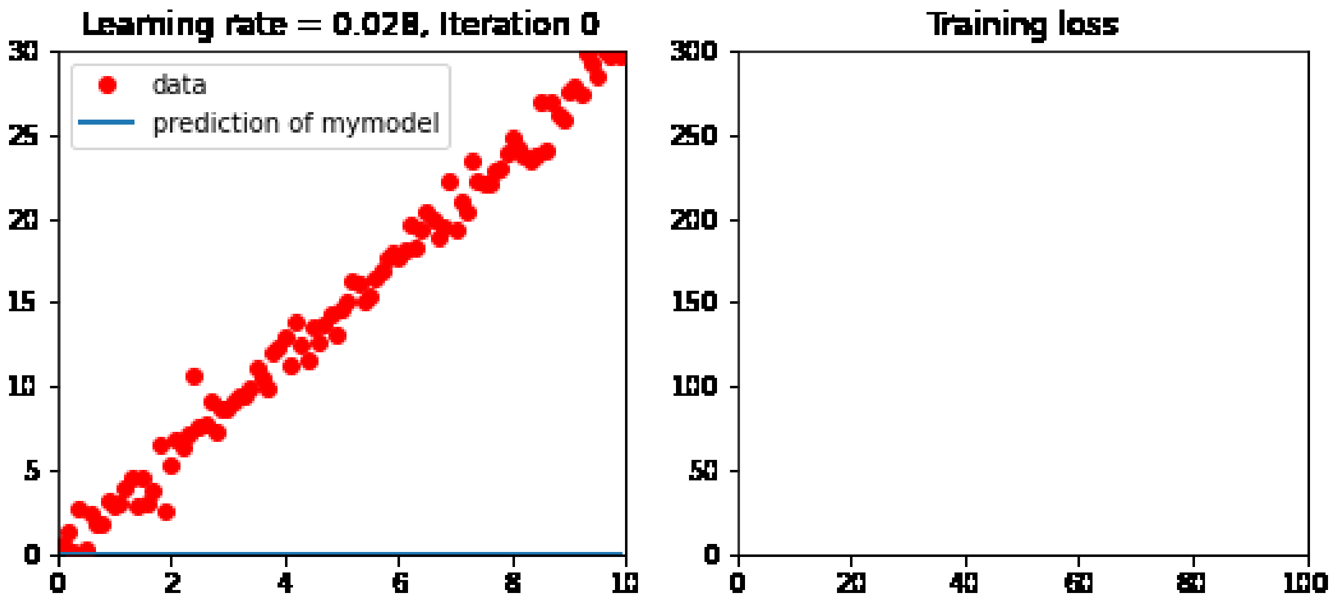
Visualizing Gradient Descent

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



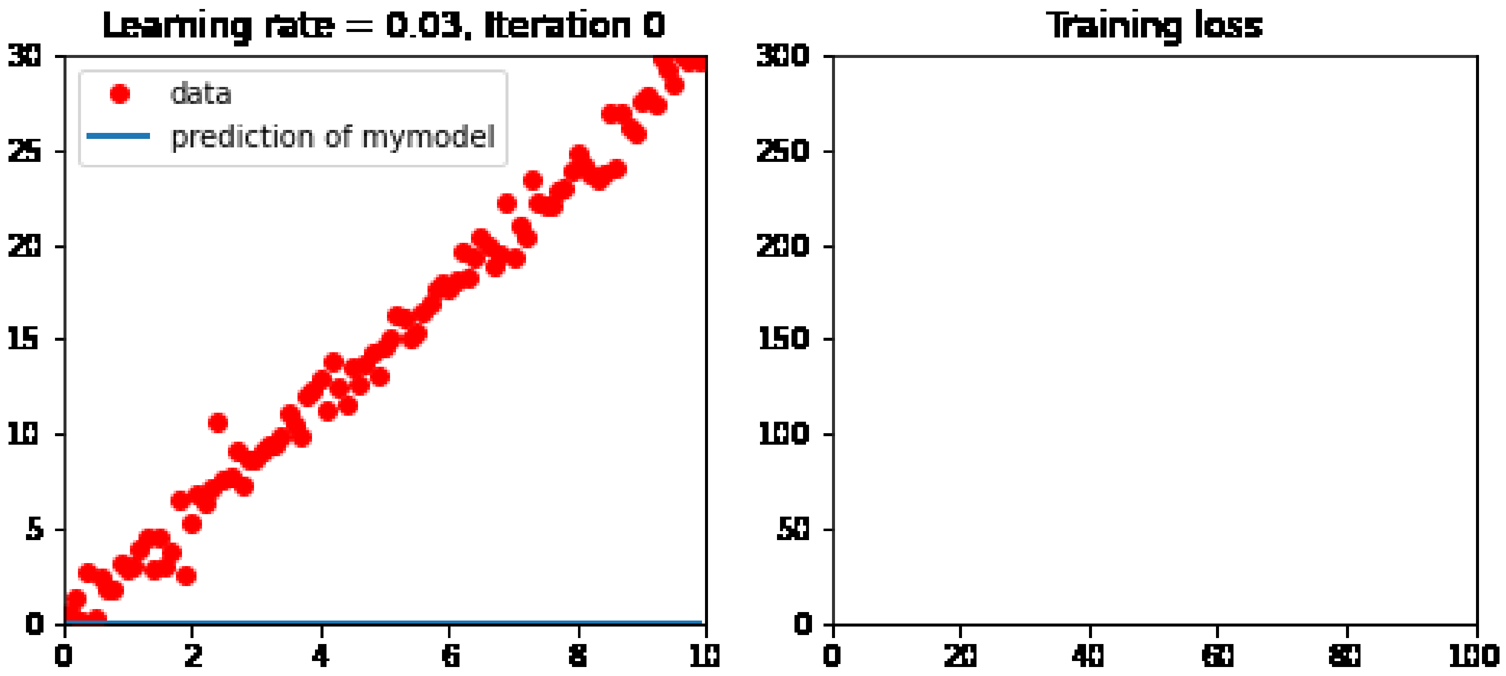
Visualizing Gradient Descent

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