STATS 507
Data Analysis in Python

Week11-2: Intro to deep learning with PyTorch

Dr. Xian Zhang

Recall the scope of this class

Part 1: Introduction to Python

Data types, functions, classes, objects, functional programming

Part 2: Numerical Computing and Data Visualization

numpy, scipy, scikit-learn, matplotlib, Seaborn

Part 3: Dealing with structured data

pandas, SQL, real datasets

Part4: Intro to Deep Learning

PyTorch, Perceptron, Multi-layer perceptron, SGD, regularization, ConvNets

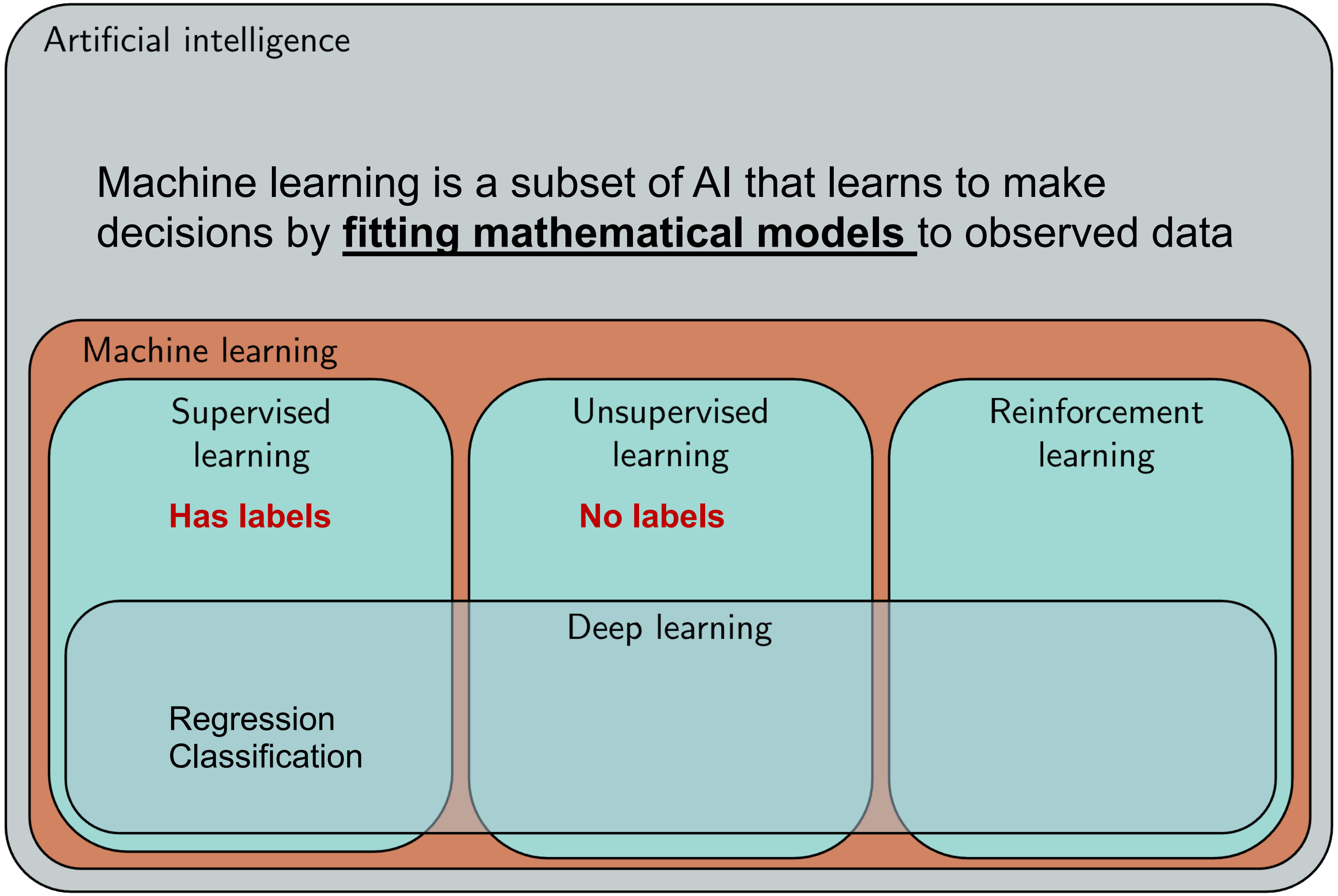
Slide Credit: Jeffrey Regier

1. Deep Learning Concepts Recap

2. Intro to PyTorch

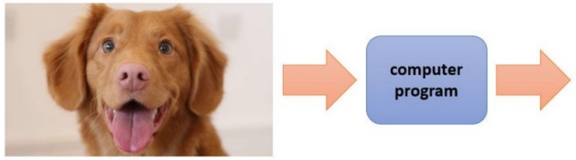
3. Perceptron and MLP

Recap: What is machine learning?



Recap: What is machine learning?

supervised learning



input: $\mathbf{x}$

output: $\mathbf{y}$

data: $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$

goal: $f_\theta(\mathbf{x}_i) \approx \mathbf{y}_i$

someone gives this to you

unsupervised learning

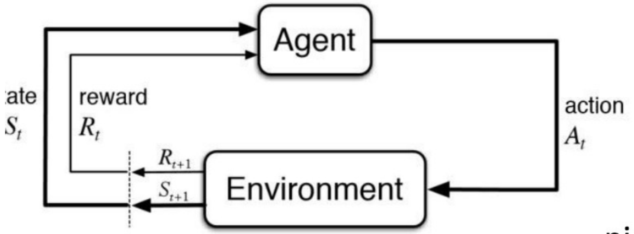
input: unlabeled data

output: Hidden structure of the data

data: $\boldsymbol{D} = \{\boldsymbol{x_i}\}$

goal: learn some hidden or underlying structure of the data

reinforcement learning



pick your own actions

input: $\mathbf{s}_t$ *at each time step*

output: $\mathbf{a}_t$ *at each time step*

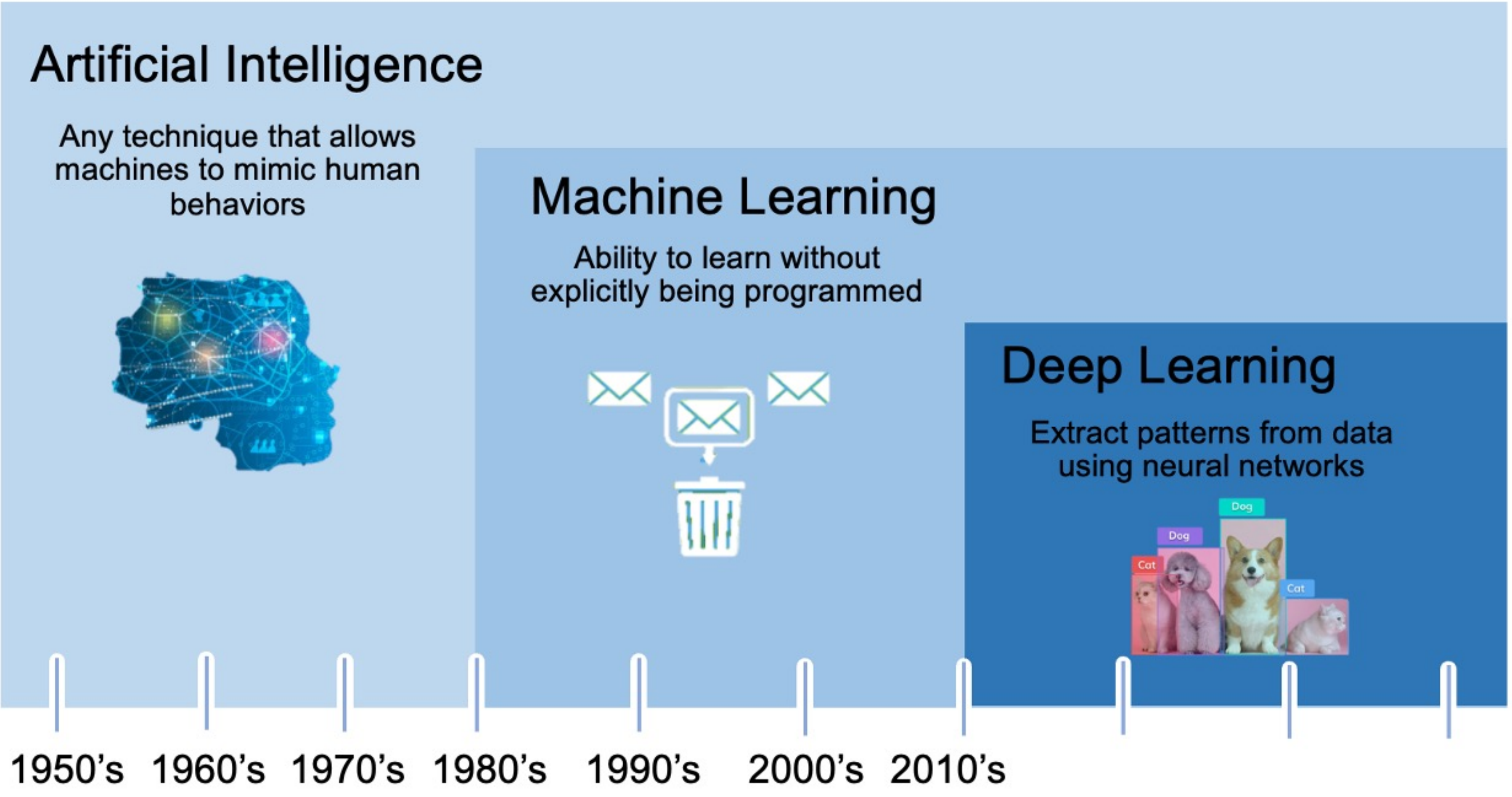
data: $(\mathbf{s}_1, \mathbf{a}_1, r_1, \ldots, \mathbf{s}_T, \mathbf{a}_T, r_T)$

goal: learn $\pi_\theta : \mathbf{s}_t \rightarrow \mathbf{a}_t$

to maximize $\sum_t r_t$

Ref: https://rail.eecs.berkeley.edu/deeprlcourse/

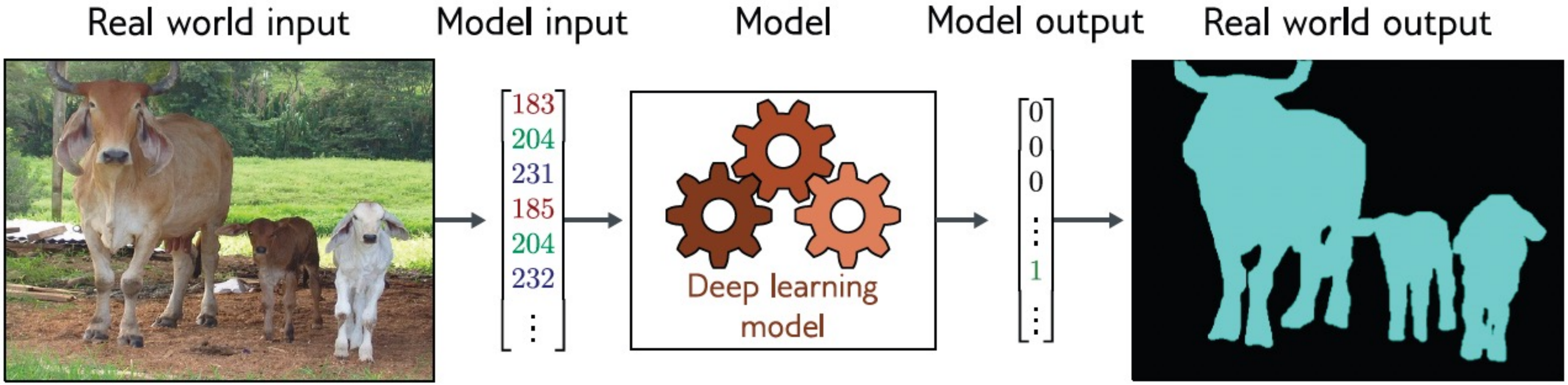
Recap: What is deep learning?



Slide Credit: Alexander Amini

Modified from MIT open course: 6.S191 and Nvidia blog

Why deep learning?



- Traditional machine learning needs **hand-engineered features**: time-consuming, brittle, and not scalable
- DP can learn the underlying feature directly from the data, little or no human intervention needed

Note: This might be a good picture for your final project

Slide Credit: Simon J.D. Prince

Can we actually do it now?

1952 Stochastic gradient descent

1958 Perceptron

- Learnable weights

⋮

1986 Back propagation

- Multi-Layer perceptron

1995 Deep convolutional NN

⋮

ChatGPT

1. Big data

- Larger dataset
- Easier collection and storage

IMAGENET

WIKIPEDIA
The Free Encyclopedia

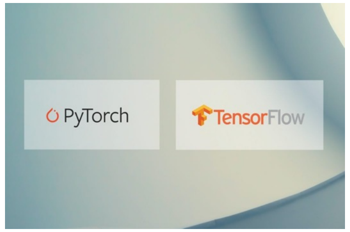
Open

2. Hardware

- GPUs
- Massively parallelizable

3. Software

- Improved techniques
- New models
- Toolboxes



Slide Credit: Alexander Amini

Modified from MIT open course: 6.S191

1. Deep Learning Concepts Recap

2. Intro to Pytorch

3. Perceptron and MLP

What's PyTorch

It's a Python-based open-sourced scientific computing package targeted at two sets of audiences:

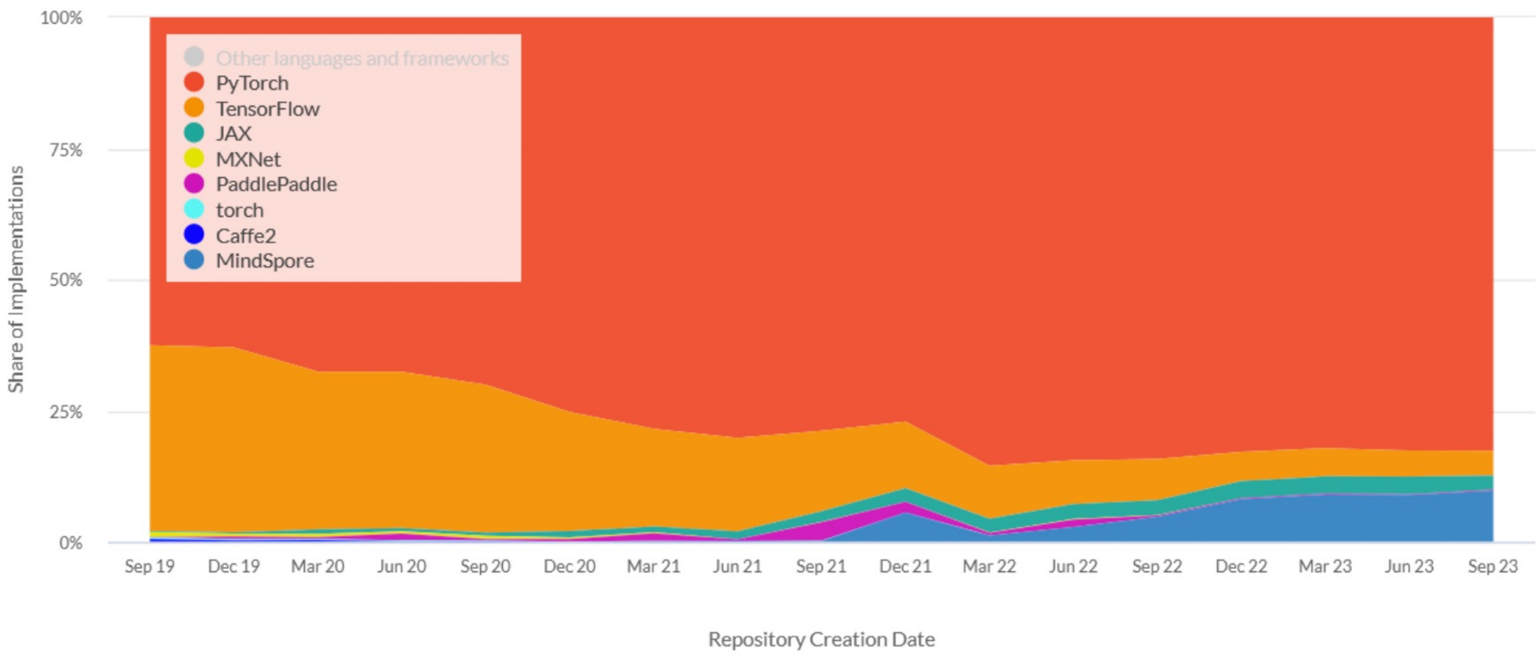
1) A replacement for **NumPy** to use the power of **GPUs**

2) A **deep learning** research platform that provided maximum flexibility and speed

https://pytorch.org/get-started/locally/

Slide Credit: Jeffrey Regier

Why PyTorch



Trends of paper implementations grouped by framework: Comparison of PyTorch vs. TensorFlow (Dec. 2023) [From viso.ai]

1. Many companies (especially open-sourced community) has standardized the usage of PyTorch internally (OpenAI…)
2. Support varying input sizes, well-documented, favored for fast experimentational and prototyping

```python
# -*- coding: utf-8 -*-
import numpy as np

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# Randomly initialize weights
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Numpy

```python
import torch


dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

# Randomly initialize weights
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Update weights using gradient descent
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

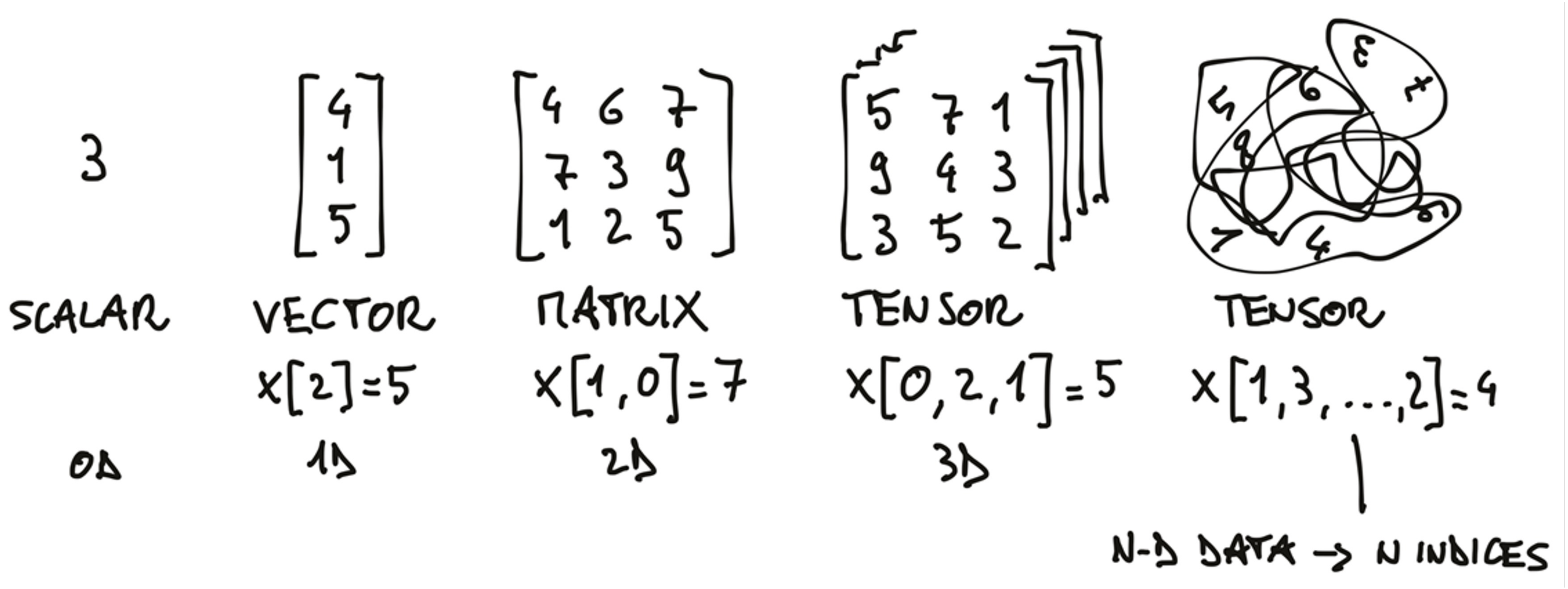
PyTorch

numpy and PyTorch have many similarities

Slide Credit: Soumith Chintala

Basic Datatype: Tensors

Just like NumPy multi-dimensional array, tensors generated vector and matrix.



Credit: E. Stevens, L. Antiga, and T. Viehmann. *Deep Learning with PyTorch*. 2020.

Properties of Tensors

TENSORS are a generalization of vectors and matrices. In PyTorch, they are a multi-dimensional matrix containing elements of a single data type.

```python
import torch
new = torch.tensor([[1,2], [3,4]])
print(new)
print(new.dtype)
print(new.shape)
print(new.device)
```

```
tensor([[1., 1.],
        [1., 1.]])
tensor([[1, 2],
        [3, 4]])
torch.int64
torch.Size([2, 2])
cpu
```

Can also be supported on GPU

Creating Tensors

With built-in functions:

```
# construct a 5*3 matrix, uninitialized
x = torch.empty(5,3)

# construct a 5*3 matrix, randomly initialized initialized
x1 = torch.rand(5,3)

# construct a 5*3 matrix of zeros, specify the data type
x2 = torch.zeros(5,3, dtype = torch.long)

# construct a tensor directly from data
x3 = torch.tensor([5.0, 3])
print(x, "\n", x1, "\n", x2, "\n", x3)
print(x3.shape)
print(x3.dtype)
print(x3.device)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
 tensor([[0.3458, 0.8035, 0.8819],
        [0.4148, 0.8149, 0.4572],
        [0.9114, 0.4643, 0.5603],
        [0.8981, 0.5040, 0.0601],
        [0.0742, 0.5991, 0.2510]])
 tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
 tensor([5., 3.])
torch.Size([2])
torch.float32
cpu
```

Can also be constructed based on existing tensor.

```
# The returned Tensor has the same torch.dtype and torch.device as this tensor
x = x.new_ones(5,3, dtype = torch.double)
print(x)

# still have the same shape, but can override value and data type
x = torch.rand_like(x, dtype=torch.float)
print(x)
x.dtype
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
tensor([[0.3273, 0.1406, 0.9821],
        [0.5814, 0.2370, 0.8258],
        [0.6330, 0.5444, 0.3414],
        [0.9174, 0.9574, 0.3635],
        [0.7715, 0.8241, 0.9388]])

torch.float32
```

Operation examples: add

Add

```
x = torch.empty(5,3)
y = torch.rand(5,3)
# syntax 1: directly use the "+" operator
z = x + y

# syntax 2: use the torch add methods
z1 = torch.add(x, y)

# syntax 3: providing an output tensor as argument
z2 = torch.empty(5,3)
torch.add(x, y, out = z2)

# syntax 4: add in-place, will modify the operated variable
y.add(x)
z3 = y

print("z", z)
print("z1", z1)
print("z2", z2)
print("z3", y)
```

Operation examples: resize

```
x = torch.rand(4, 4)

# can use .view or reshape to perform resize to PyTorch tensors
y = x.view(16)
# the size -1 is inferred from other dimensions
z = x.reshape(-1, 8)
print("x.shape: ", x.shape, "\ny.shape: ", y.shape, "\nz.shape: ", z.shape)
```

```
x.shape:  torch.Size([4, 4])
y.shape:  torch.Size([16])
z.shape:  torch.Size([2, 8])
```

Tensor to and from a NumPy Array

```
a = torch.ones(5, dtype=torch.float64)
b = a.numpy()

print("From PyTorch tensor: ", a, "\nTo NumPy Array:", b)
print(a.dtype)
print(b.dtype)
```

```
From PyTorch tensor:  tensor([1., 1., 1., 1., 1.], dtype=torch.float64)
To NumPy Array: [1. 1. 1. 1. 1.]
torch.float64
float64
```

```
import numpy as np
b = np.ones(5)
a = torch.from_numpy(b)
print("From NumPy Array: ", b, "\nTo NumPy Array:", a)
```

```
From NumPy Array:  [1. 1. 1. 1. 1.]
To NumPy Array: tensor([1., 1., 1., 1., 1.], dtype=torch.float64)
```

CUDA Tensors

```
[36] # let us run this cell only if CUDA is available
     # We will use ``torch.device`` objects to move tensors in and out of GPU
     if torch.cuda.is_available():
         device = torch.device("cuda")          # a CUDA device object
         y = torch.ones_like(x, device=device)  # directly create a tensor on GPU
         x = x.to(device)                       # or just use strings ``.to("cuda")``
         z = x + y
         print(z)
         print(z.to("cpu", torch.double))       # ``.to`` can also change dtype together!
```

x is now stored on the GPU

“cuda” stands for “Compute Unified Device Architecture”. NVIDIA's parallel computing platform and programming model that allows developers to use NVIDIA GPUs for general-purpose computing.

Slide credit: Christian S. Perone

GPU speedup on Google colab

```
[1] import torch
    import time
```

```
[2] x_cpu = torch.randn(120000, 10000)
    y_cpu = torch.randn(10000, 1)
```

```
[3] x_gpu = x_cpu.cuda()
    y_gpu = y_cpu.cuda()
```

```
[4] start = time.time()
    x_cpu.mm(y_cpu)
    print(time.time() - start)
```

```
0.39958739280700684
```

```
start = time.time()
x_gpu.mm(y_gpu)
print(time.time() - start)
```

```
0.004973173141479492
```

Matrix-vector multiplication on the GPU is nearly 100x faster!

Slide credit: Christian S. Perone

Automatic Differentiation

```
x = torch.ones(2, 2, requires_grad = True)
```

```
x
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

`requires_grad=True` tells PyTorch to track all operations performed on this tensor so that we can compute gradients (derivatives) with respect to it later. It's essential for automatic differentiation (autodiff).

```
y = x + 2
print(y)
```

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

```
y.grad_fn
```

```
<AddBackward0 at 0x14dd86a10>
```

y was created as a result of operation and it has a `grad_fn`

Gradient function for addition operations in PyTorch's autograd system. It tells `PyTorch` how to compute derivatives when going "backward" through an addition operation during backpropagation.

```
z = y * y + 2
```

```
z
```

```
tensor([[11., 11.],
        [11., 11.]], grad_fn=<AddBackward0>)
```

```
out = z.mean()
```

```
print(z, out)
```

```
tensor([[11., 11.],
        [11., 11.]], grad_fn=<AddBackward0>) tensor(11., grad_fn=<MeanBackward0>)
```

Can add more operations.

Slide credit: Christian S. Perone

Gradients

When we call backward(), PyTorch uses these `grad_fns` to compute:

```
x = torch.tensor([1.0], requires_grad=True)
y = x * 2   # grad_fn=<MulBackward0>
z = y + 3   # grad_fn=<AddBackward0>

# When we call backward(), PyTorch uses these grad_fns to compute:
z.backward()
print(x.grad)
```

Results?

```
dz/dx = (dz/dy) * (dy/dx) = 1 * 2 = 2
```

```
tensor([2.])
```

Reference: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

Slide credit: Christian S. Perone

Test your understanding about autograd

```
# Example 1: Simple autograd
x = torch.tensor([2.0], requires_grad=True)
y = x * 2  # Operation
y.backward()  # Compute gradient
print(x.grad)
```

2

```
# Example 2: Multiple operations
x = torch.tensor([2.0], requires_grad=True)
y = x * 2
z = 2 * y + 1
z.backward()
print(x.grad)
```

4

```
# Example 3: Using torch.autograd.grad() directly
x = torch.tensor([2.0], requires_grad=True)
y = x * 2
gradient = torch.autograd.grad(y, x)[0]  # Alternative to backward()
print(gradient)
```

2

https://pytorch.org/docs/stable/generated/torch.autograd.grad.html

In-class practice

1. Deep Learning Concepts Recap

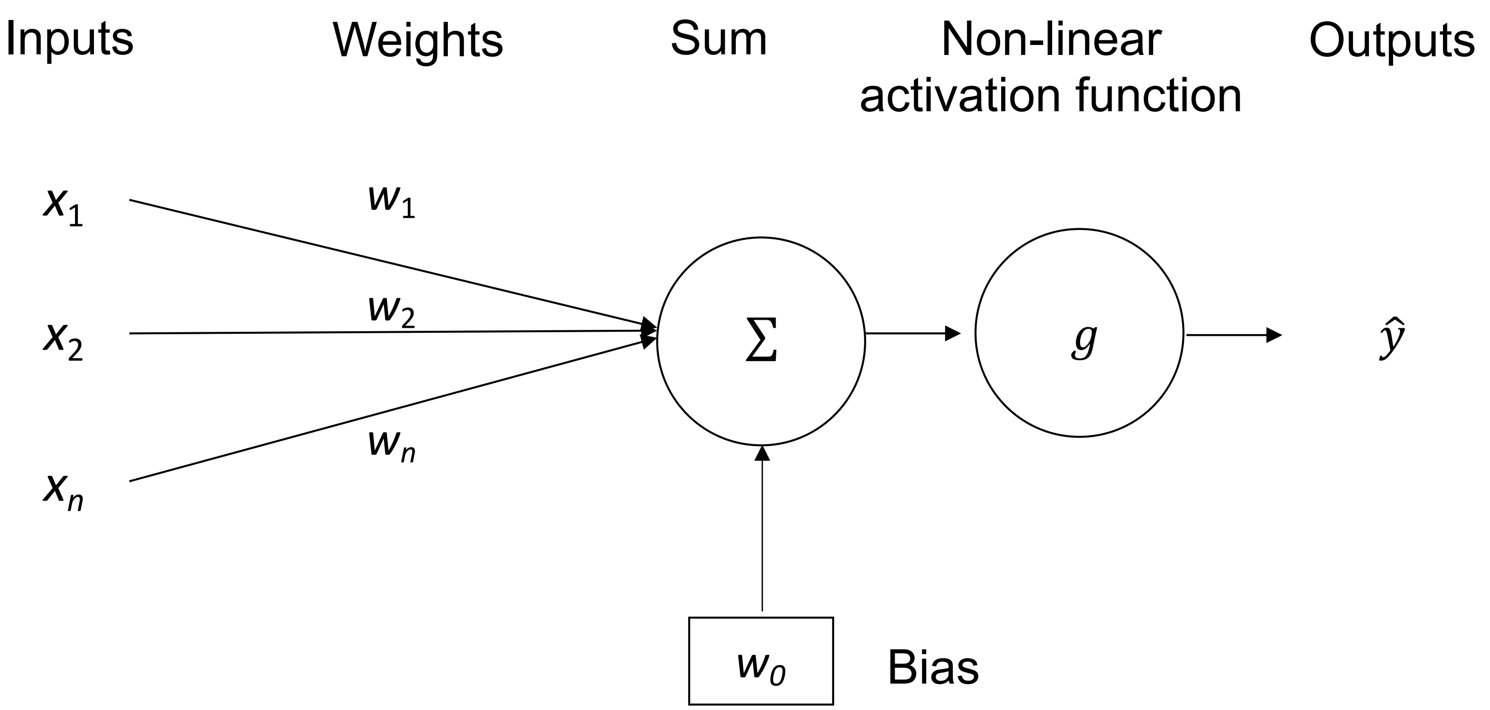
2. Intro to Pytorch

3. Perceptron and MLP

The perceptron (a single neuron)

The structural building block of deep learning

The perceptron (forward propagation)

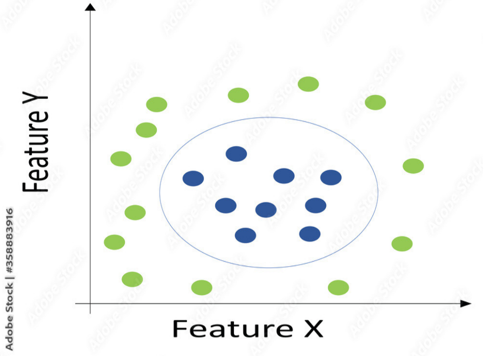


3 2 1

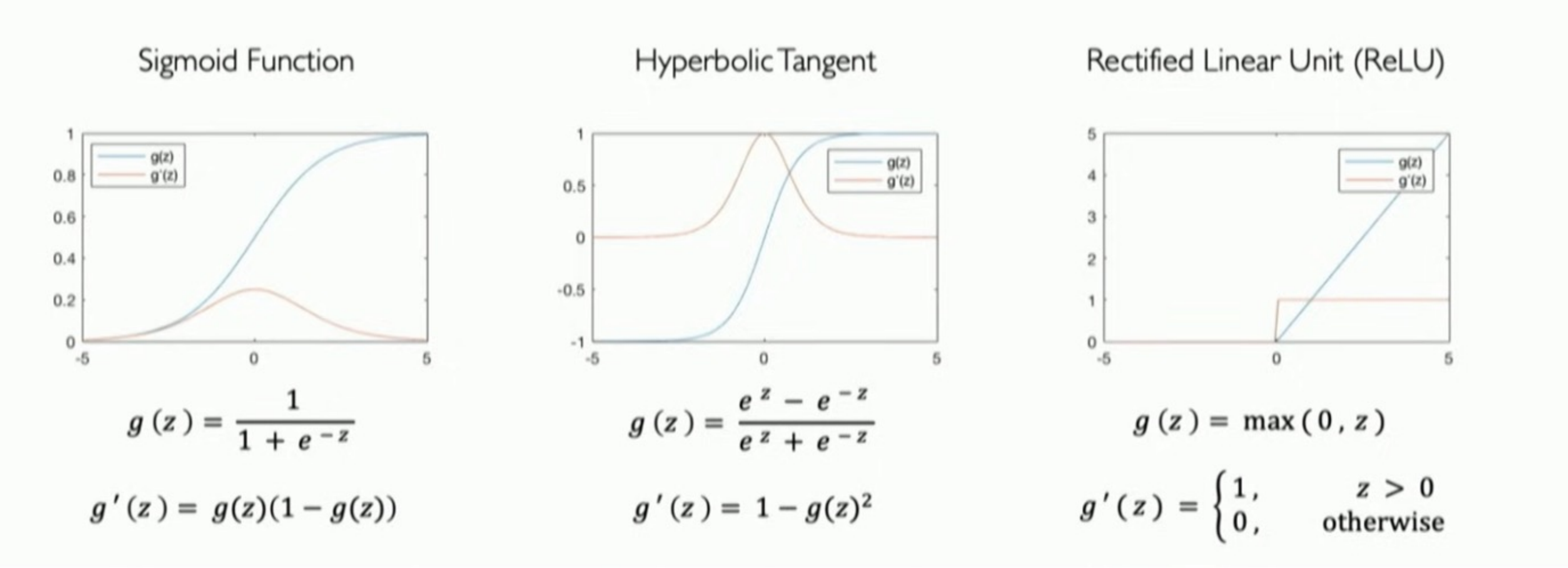
$$\hat{y} = g(w_0 + \sum_{i=1}^{n} x_i w_i) = g(z)$$

More on activation functions

1. It is important to introduce non-linearity when analyze data

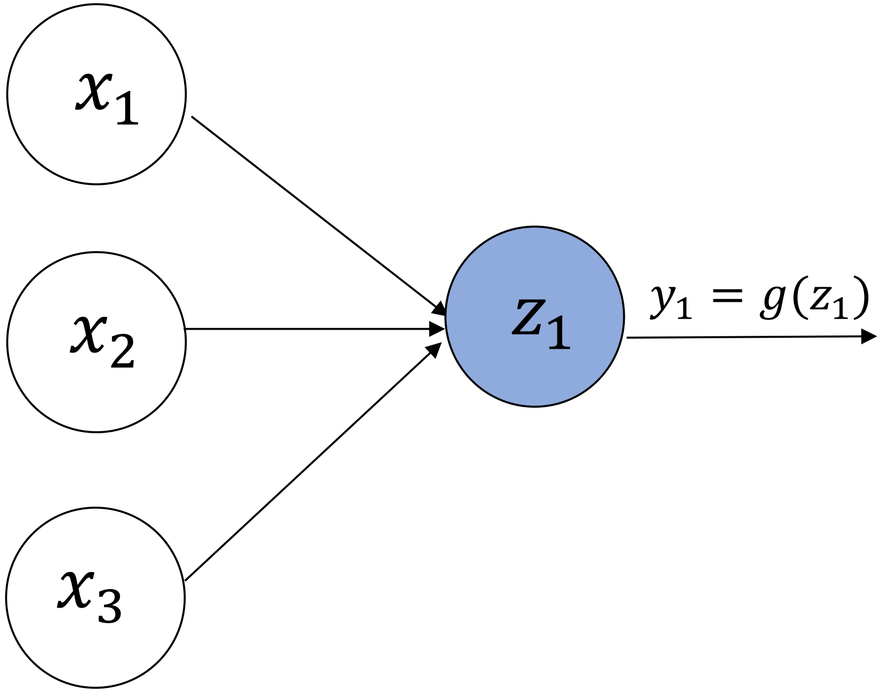


2. Common activation functions to introduce non-linearity



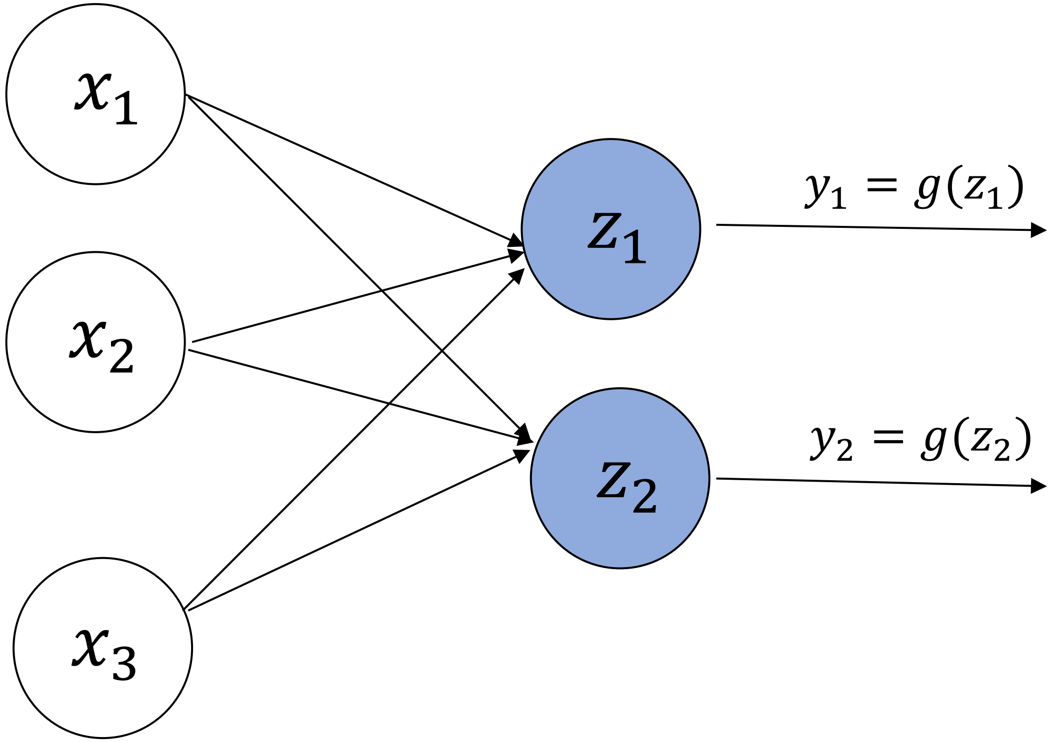
`torch.sigmoid(x)` `torch.tanh(x)` `torch.relu(x)`

Multi-output perceptron



one simplified perceptron

$$z = w_0 + \sum_{i=1}^{n} x_i w_i$$



$$z_i = w_{o,i} + \sum_{j=1}^{m} x_j w_{j,i}$$

$$\boldsymbol{z} = \boldsymbol{w_o} + \boldsymbol{x} W^T$$

Build a single layer from scratch

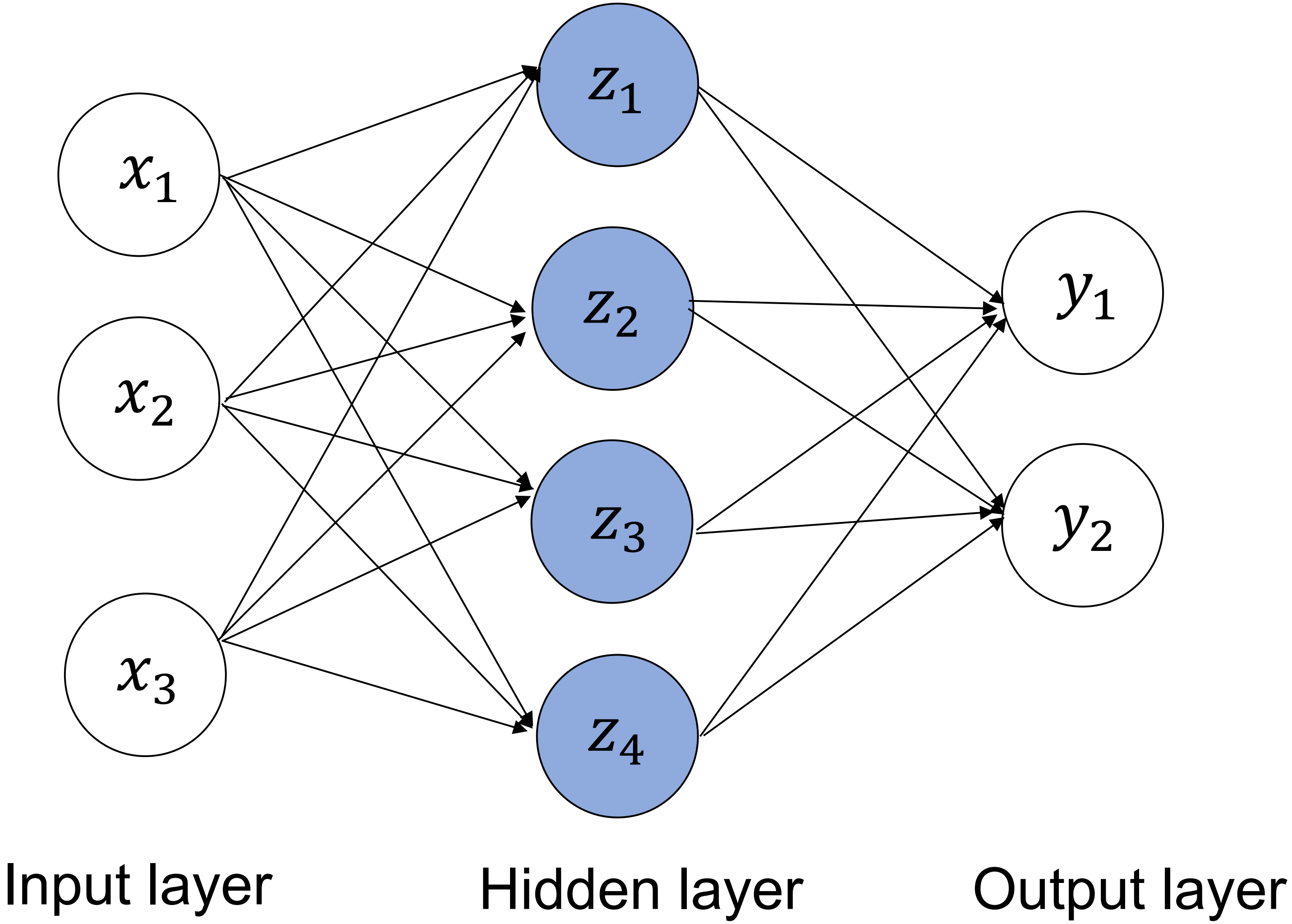
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MyDenseLayer(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(MyDenseLayer, self).__init__()

        # Initialize weights and biases
        self.weights = nn.Parameter(torch.randn(output_dim, input_dim))
        self.bias = nn.Parameter(torch.randn(output_dim))

    def forward(self, x):
        # Perform matrix multiplication and add bias
        z = torch.matmul(x, self.weights.t()) + self.bias

        # Apply ReLU activation function
        y = F.relu(z)
        return y
```

```
self.linear = nn.Linear(input_dim, output_dim)
```

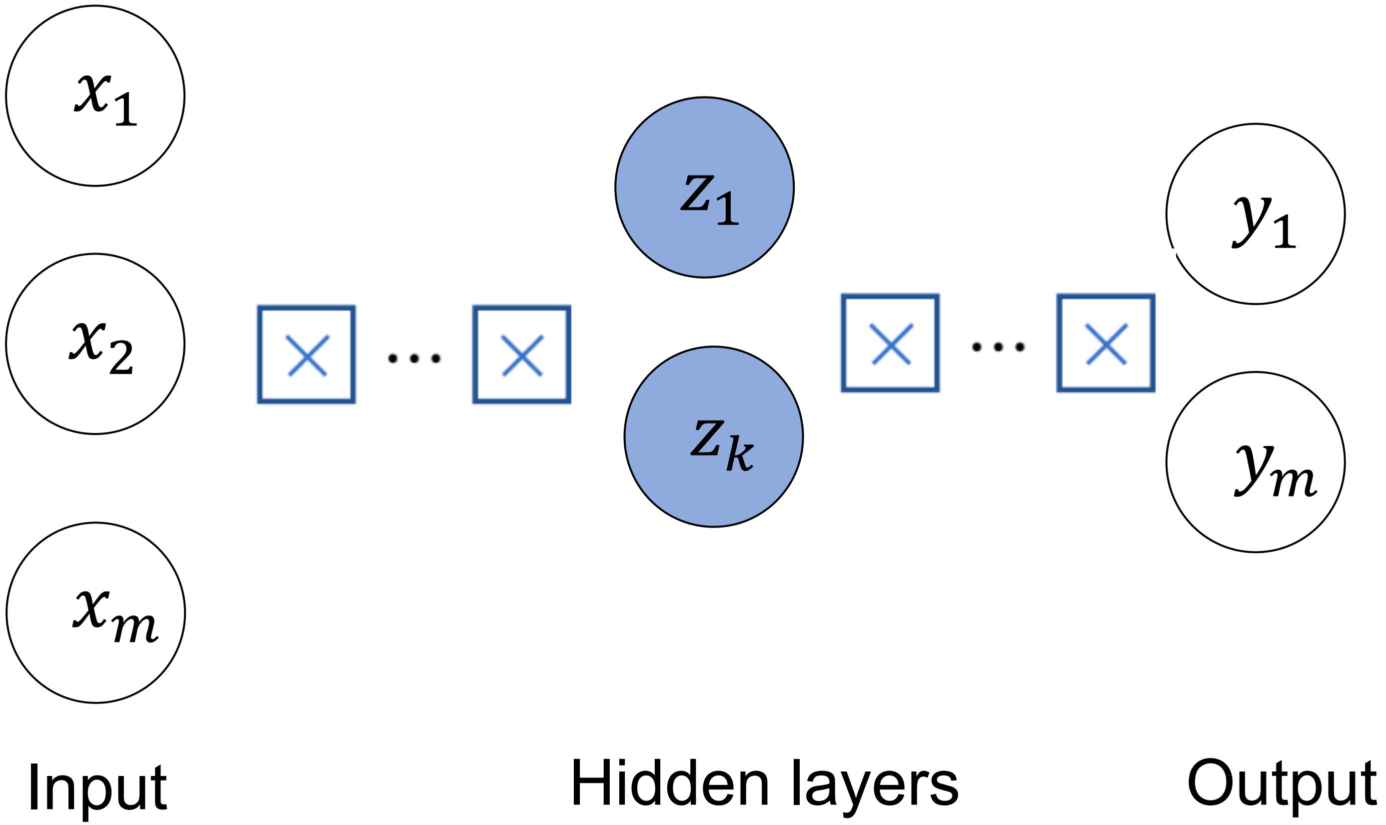
$$\boldsymbol{z} = \boldsymbol{w_o} + \boldsymbol{x}W^T$$

$$\boldsymbol{y} = g(\boldsymbol{z})$$

Shallow(single layer) Neural Network



Deep(multi-layer) Neural Network



Multi-layer perceptron (a two layer DNN)

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MultilayerPerceptron(nn.Module):
    def __init__(self, num_features, hidden_size1, hidden_size2, num_classes):
        super(MultilayerPerceptron, self).__init__()

        # 1st hidden layer
        self.linear_1 = nn.Linear(num_features, hidden_size1)

        # 2nd linear layer
        self.linear_2 = nn.Linear(hidden_size1, hidden_size2)

        # output layer
        self.linear_out = nn.Linear(hidden_size2, num_classes)

    def forward(self, x):
        x = F.relu(self.linear_1(x))
        x = F.relu(self.linear_2(x))
        logits = self.linear_out(x)
        probas = F.softmax(logits, dim = 1)

        return logits, probas
```

Define model parameters that will in instantiated when an object of this class is created

Define how and in what order the model parameters should be used in the forward pass

Other things

HW7 due this week.

HW8 will be about SQL and PyTorch

Coming next:

Loss function and how to train a model