CS167: Machine Learning

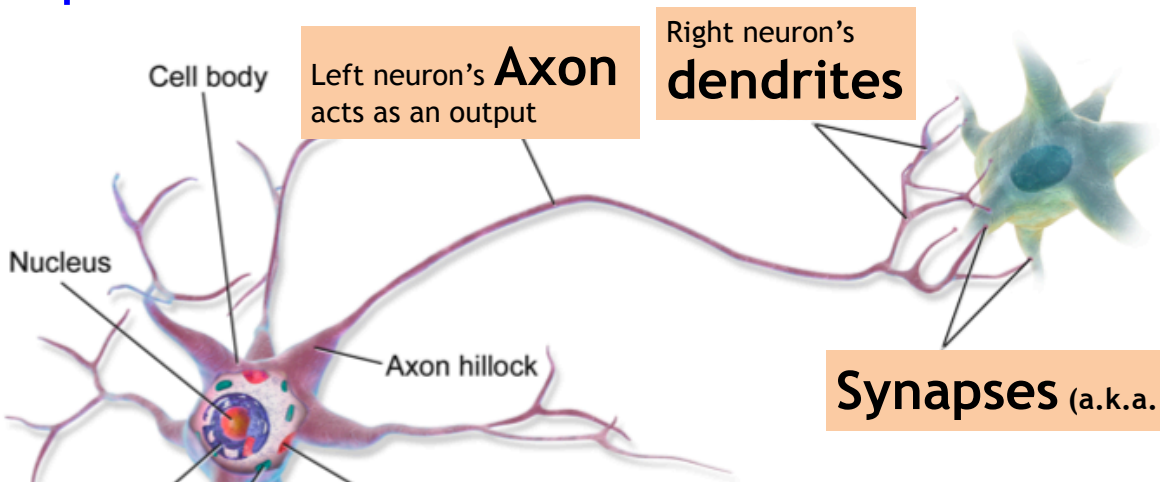
PyTorch Basics
A simple MLP implementation with PyTorch

Wednesday, April 8th, 2026

Recap: Last Class

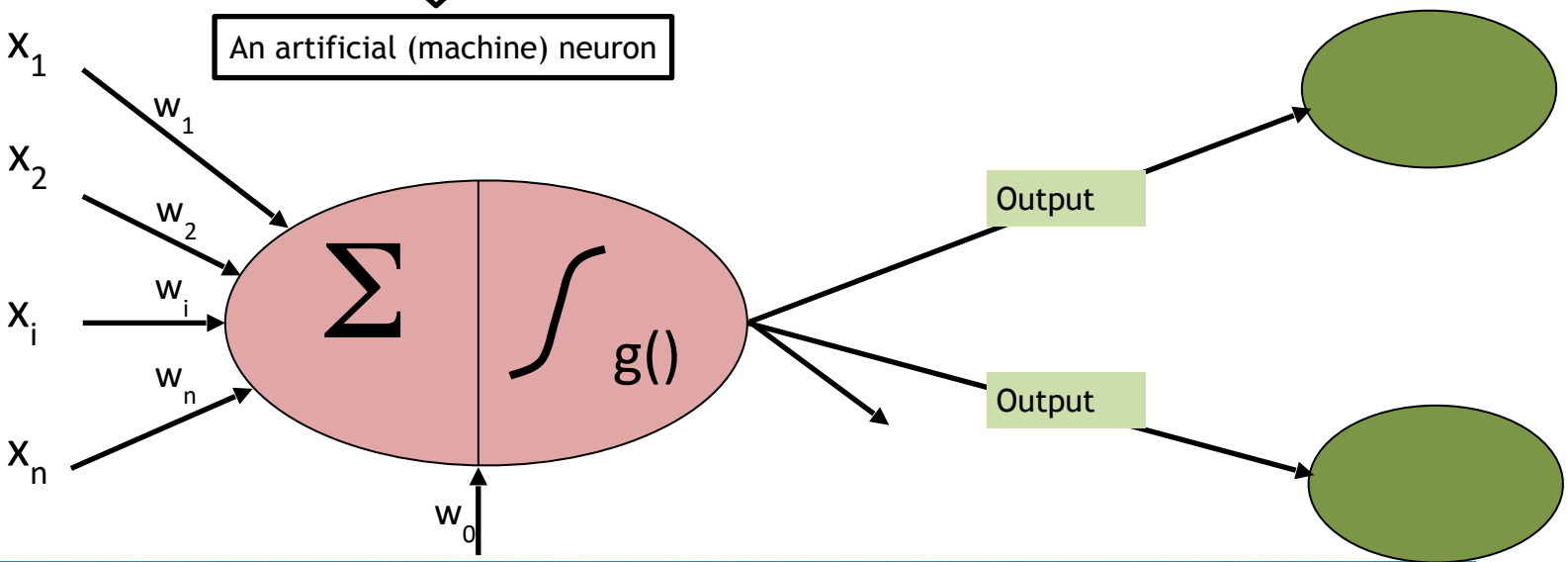
- Connections with biology: natural neurons vs. artificial neurons
- Multilayer Perceptrons (MLP)
- MLP Structure
- Learning MLP Weight Parameters
  - Recap from last week's offline lecture
  - Trainable parameters and their learnable weights

Recap: Natural neurons vs. artificial neurons



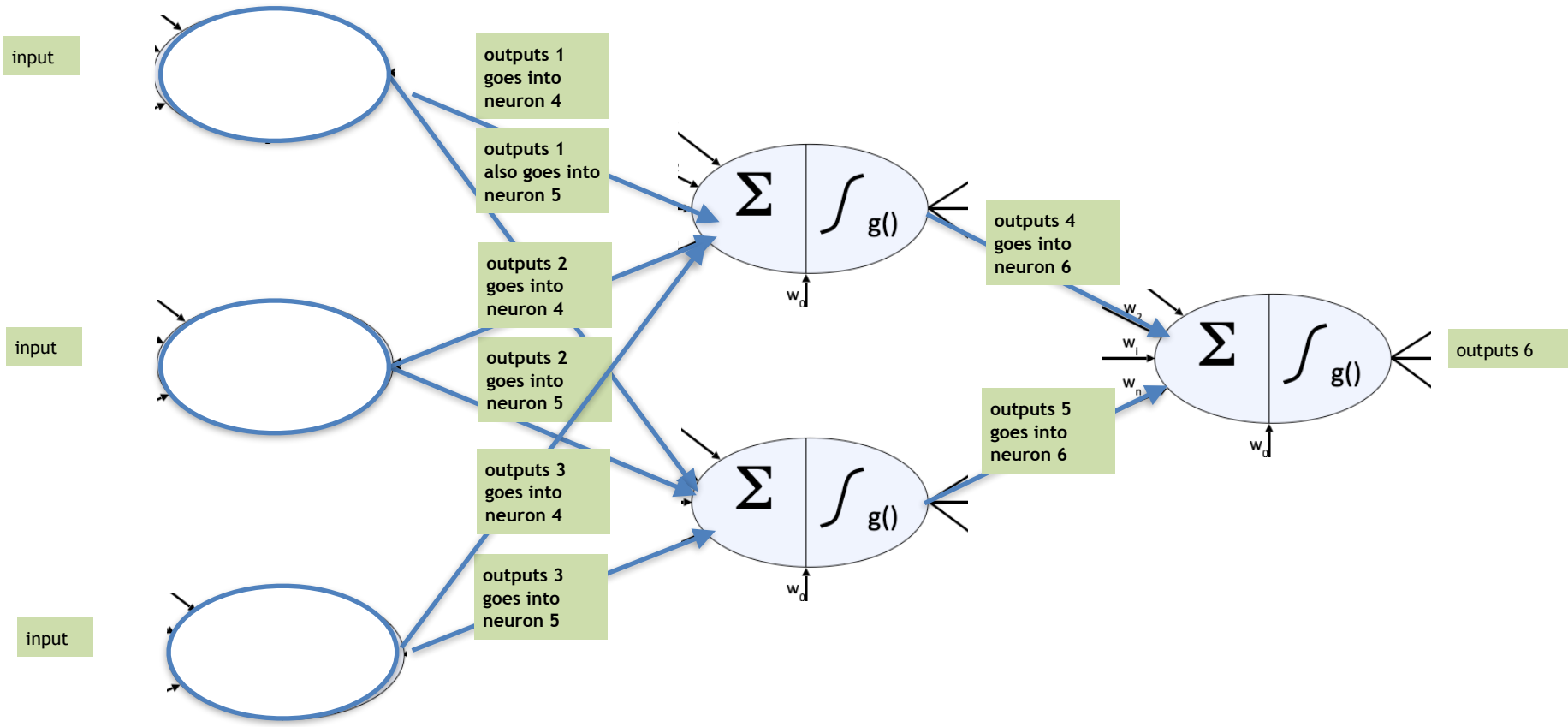
Natural neuron

An artificial (machine) neuron



Recap: 1-Hidden Layer Neural Network

- We created our first multilayer perceptron (MLP)
- Any layers in between input layer and output layer are called hidden layers
- Hence this MLP can also be called 1-hidden layer neural network

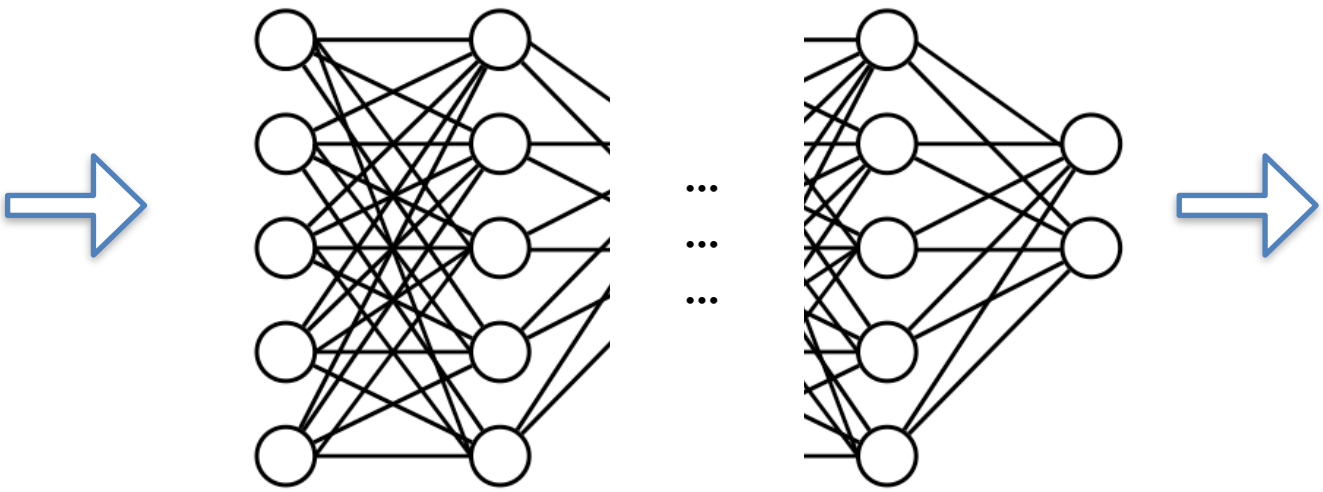


Neurons in the input layer do not have activation functions

Input Layer | Hidden Layer | Output Layer

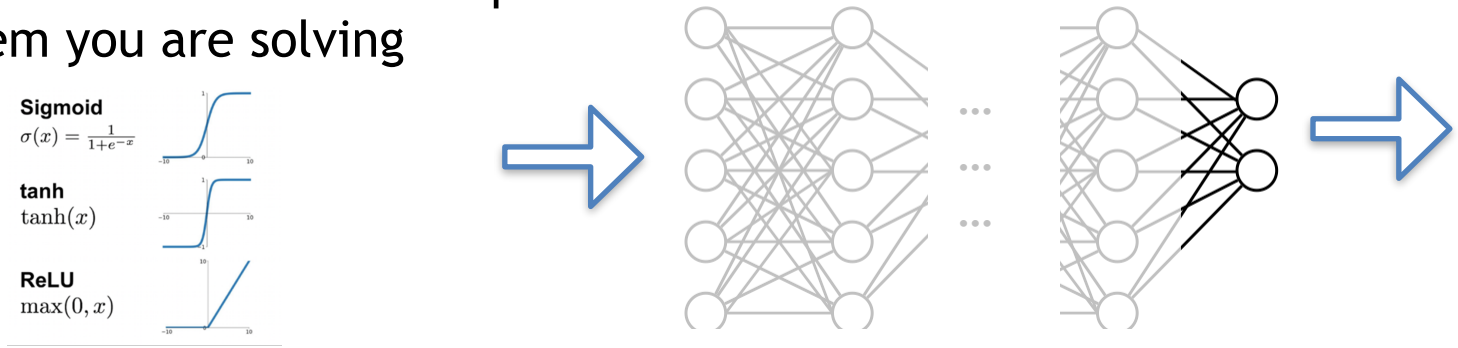
Recap: MLP (Network) Structure

- Each of these questions need to be answered before you set up your neural network:
  - how many hidden layers should I have? (depth)
  - how many neurons should be in each layer? (width)
  - what should your activation be at each of the layers?



Recap: Final Output Nodes

- In general, the complexity of your network should match the complexity of your problem. The final output nodes should be related to what kind of problem you are solving



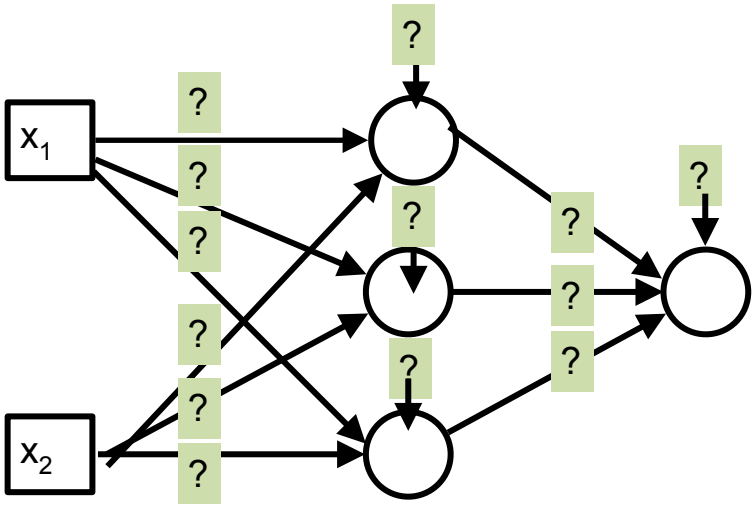
Activation Function	Function	Lower bound	Upper bound	Type of Machine Learning
Linear	$f(z) = az$	$-\infty$	∞	regression where results can be negative
Rectified Linear Unit (ReLU)	$relu(z) = max(0, z)$	0	∞	regression where results can't be negative
Sigmoid	$sigmoid(z) = \frac{1}{1+e^{-z}}$	0	1	binary classification
Softmax	$softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$	0	1	multiclass classification

Recap: Last Class

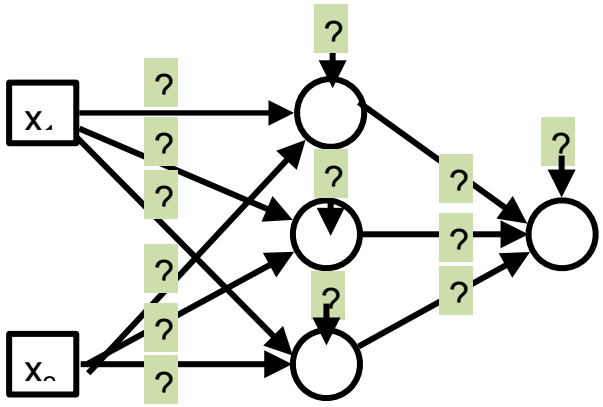
- Connections with biology: natural neurons vs. artificial neurons
- Multilayer Perceptrons (MLP)
- MLP Structure
- Learning MLP Weight Parameters
  - Recap from previous week's offline lecture
  - Trainable parameters and their learnable weights

Training to Learn MLP (Network) Structure Parameters

- The trainable parameters are the *weights (w's)* which are learned from the training data



Training to Learn MLP (Network) Structure Parameters



- The goal is to **minimize the error** predicted by the network (from last lecture) from the training data
  - Gradient Descent
  - Stochastic Gradient descent

- Gradient Descent
  - calculate the gradient vector based on that batch $\nabla E(\mathbf{w})$
  - adjust (or update) the values of the weights based on the gradient vector to that batch

$$\mathbf{w}^{\mathbf{new}} = \mathbf{w}^{\mathbf{old}} - \eta \nabla \mathbf{E}(\mathbf{w})$$

Training to Learn MLP (Network) Structure Parameters

- The specific name for the weight learning algorithm is Backpropagation. It is glorified name but it is gradient descent under the hood.
- It tunes **the weights** over a neural network using **gradient descent** to iteratively reduce the error in the network.

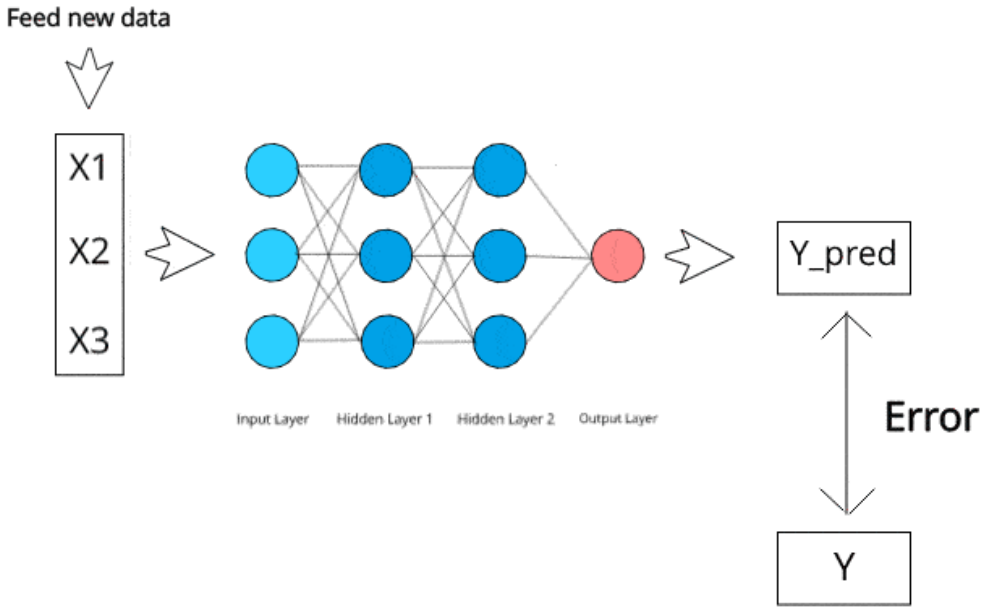
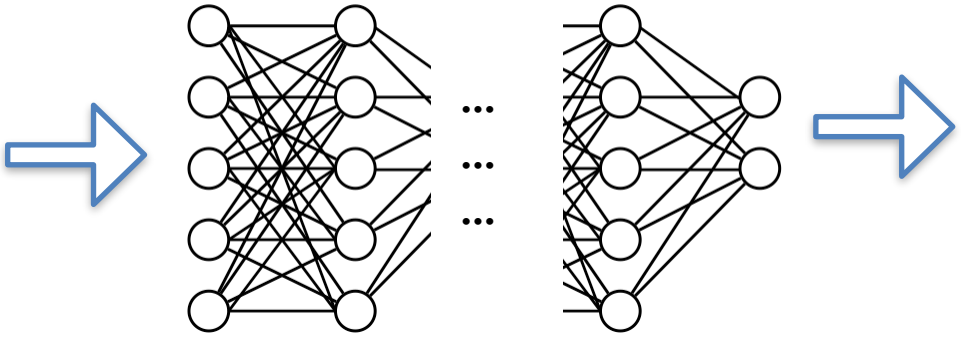


Image reference

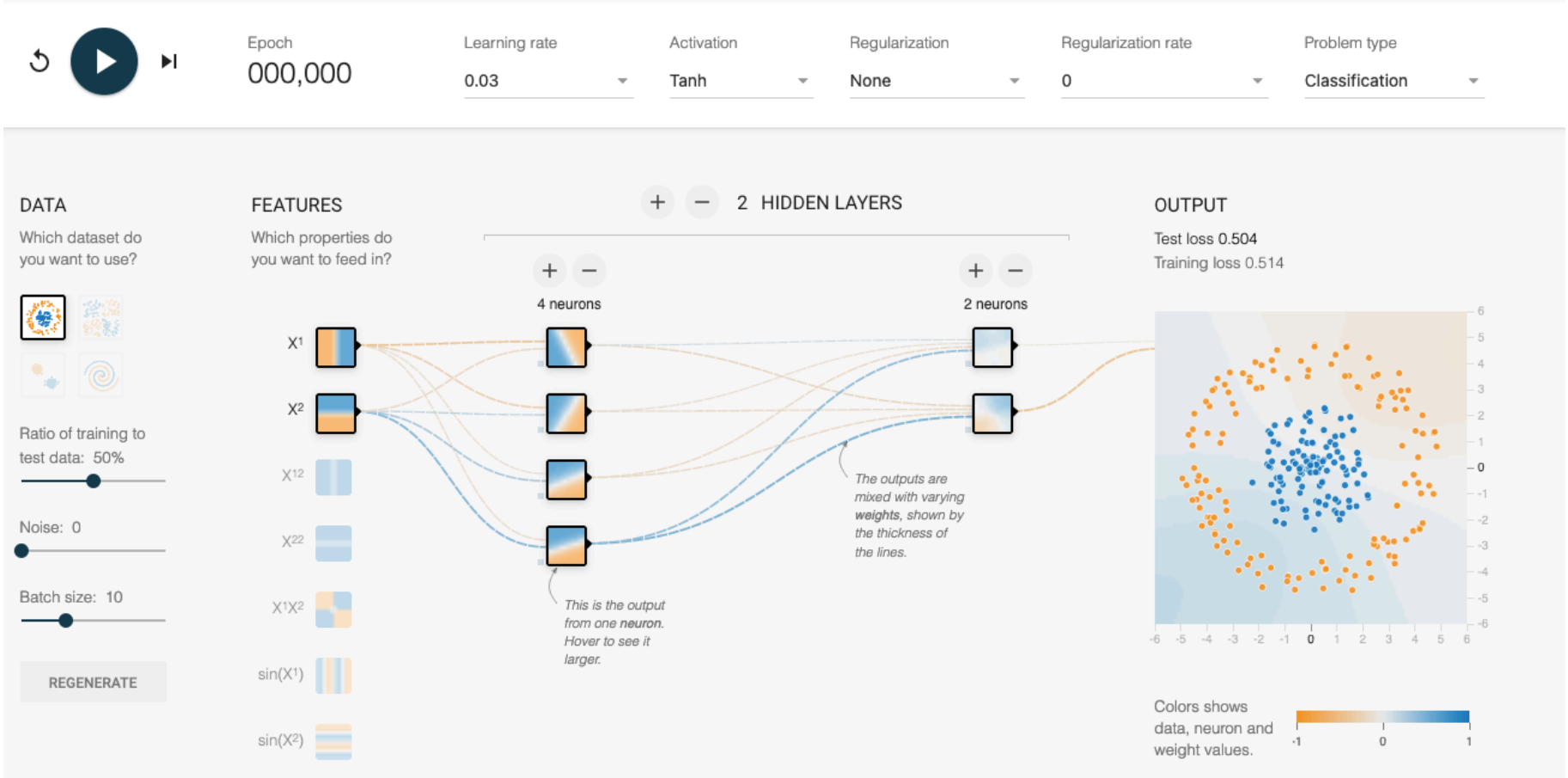
MLP Summary

- MLPs are effective in finding non-linear patterns in the training data
  - can be applied to **regression** or **classification**.
  - **backpropagation** tunes the weights over a neural network using **gradient descent** to iteratively reduce the error in the network
  - **overfitting** the training data is common and is important to avoid
  - the following parameters should be tuned when using MLPs:
    - number of epochs
    - structure of the network (depth, width)
    - activation function
    - eta (learning rate)



Tinker with the Following to See MLP in Action

- MLPs are effective in finding non-linear patterns in the training data



https://playground.tensorflow.org

Today's Agenda

- PyTorch Basics

- Simple Multilayer Perceptrons (MLP) Implementation using PyTorch

PyTorch

- PyTorch is machine learning framework based on Torch library. It has a Python interface.

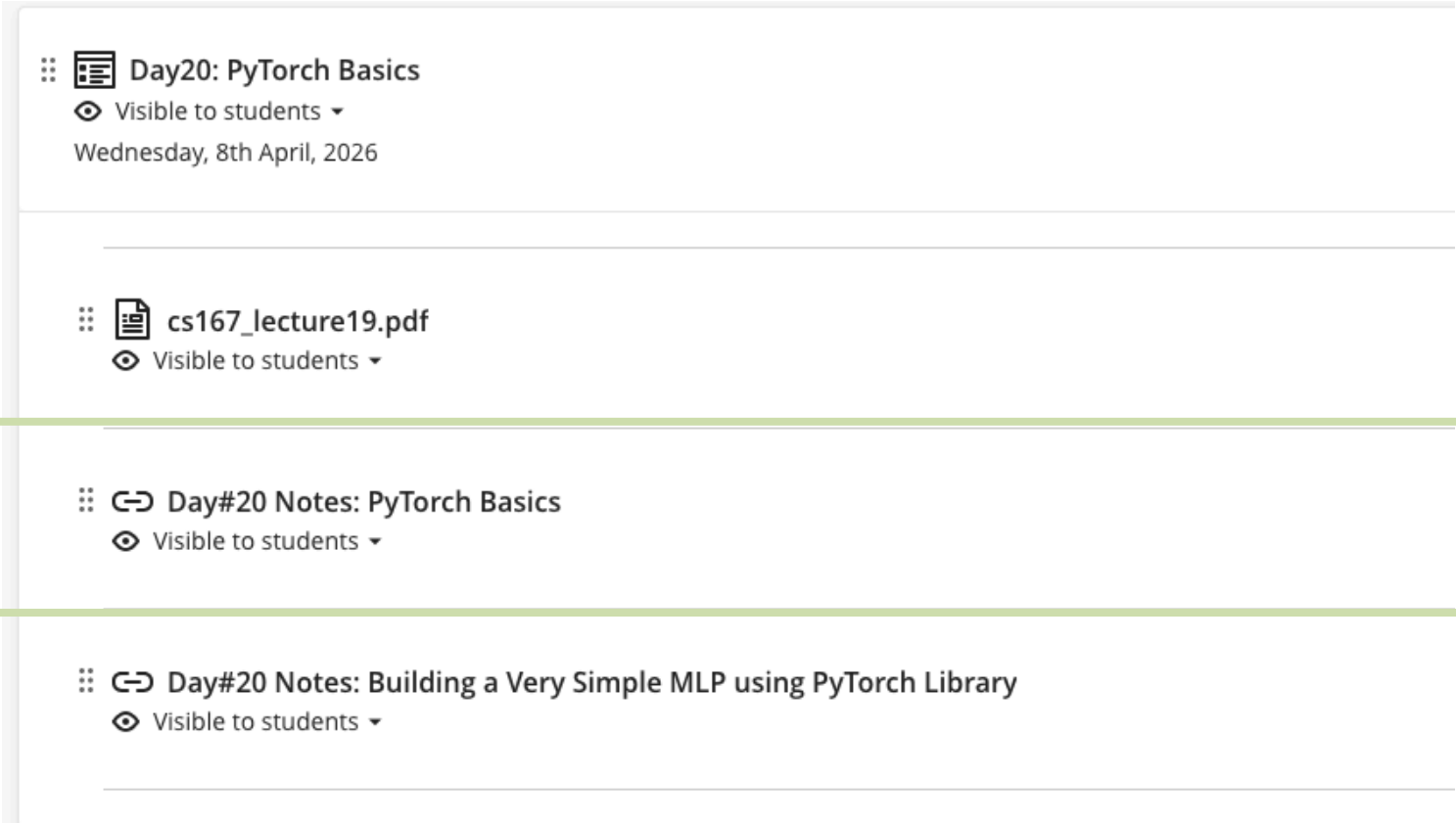
- This is a very popular framework for building and deploying deep learning application including MLP, and other future models we will learn about in this course

- Colab and Kaggle both has PyTorch support hence we can readily run our PyTorch code without worrying about the installation. But optionally, if you have GPU in your workstation (laptop/desktop), you can install a fresh copy of PyTorch there.

https://pytorch.org/

PyTorch

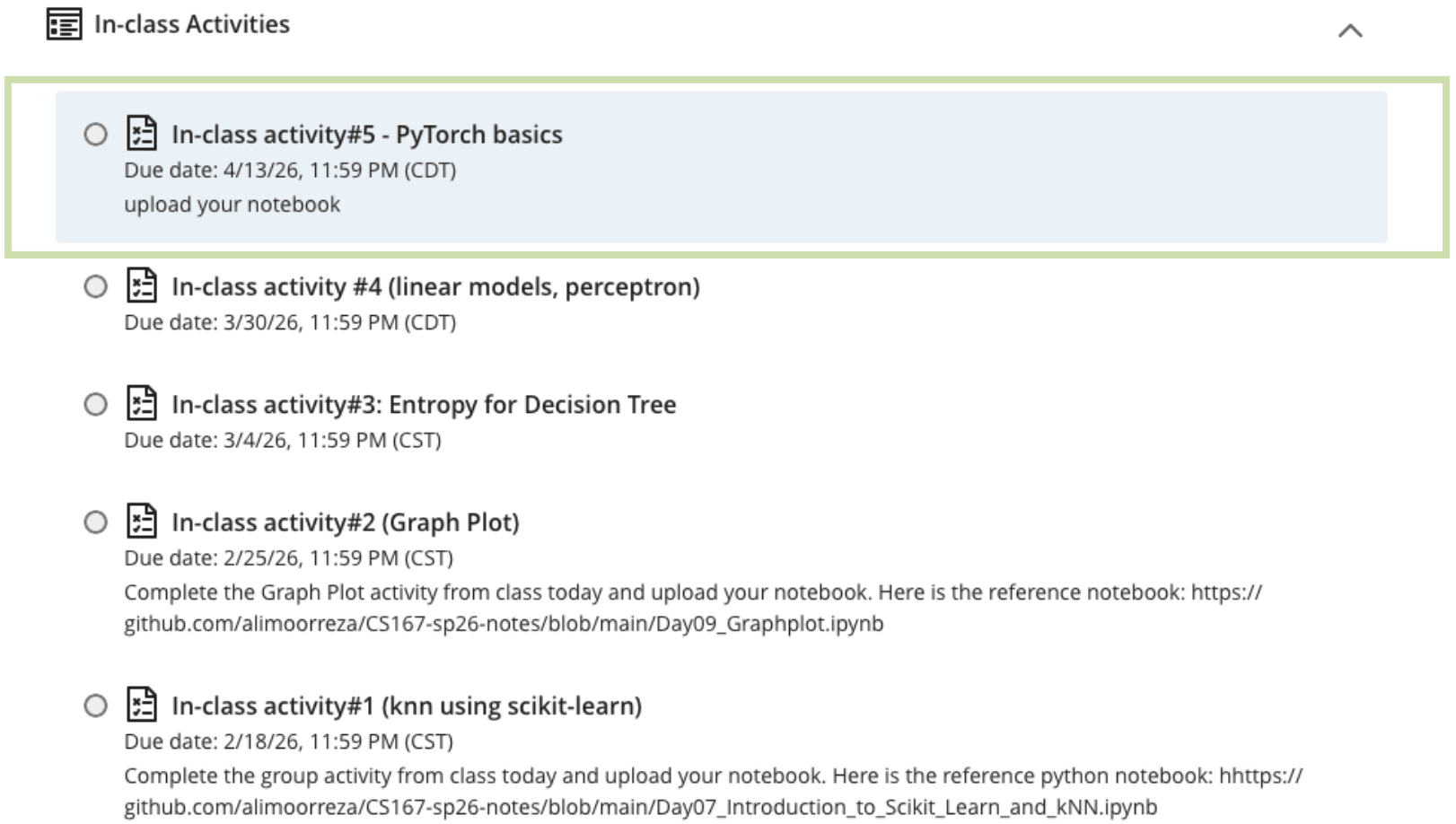
- Go to Blackboard and work on the notebook titled "PyTorch Basics."



https://pytorch.org/

PyTorch

- Upload your notebook to Blackboard (under ‘Assignment’ section) once completed!



https://pytorch.org/

Today's Agenda

- PyTorch Basics

- Simple Multilayer Perceptrons (MLP) Implementation using PyTorch

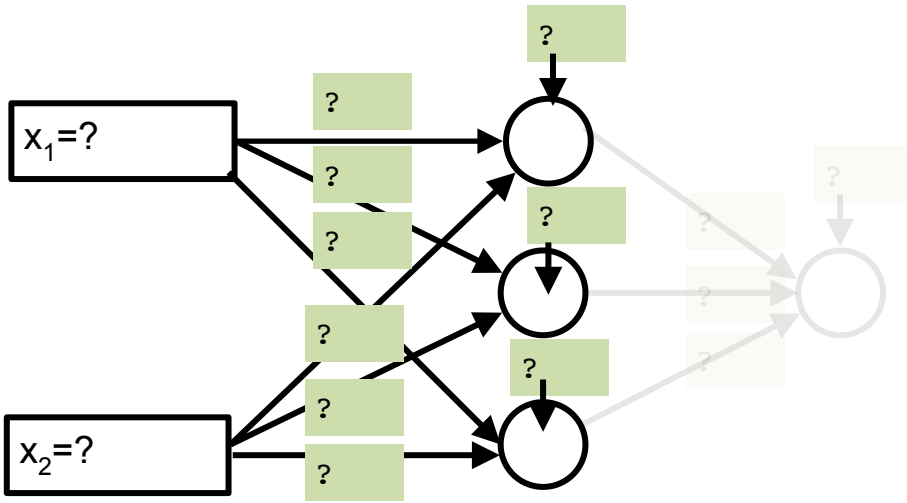
Important Design Questions for MLP

- Each of these questions need to be answered before you set up your **multilayer perceptron**
  - Q1: how many hidden layers should be there? (depth)
  - Q2: how many neurons should be in each layer? (width)
  - Q3: how many dense connections should be there in between each adjacent layers
  - Q4: what should the activation be at each of the intermediate layers?
    - `sigmoid(), tanh(), rectified-linear-unit(), etc`
  - Q5: what should be activation of the final layer
    - `depends the task` *`classification`* `(sigmoid(), softmax()) vs.` *`regression`*

Create MLP Layers

- A **multilayer perceptron** is the simplest type of neural network. It consists of perceptrons (aka nodes, neurons) arranged in layers. Create the first layer.

```
torch.manual_seed(1)                    # for reproducibility
# construction of a linear layer
num_neuron_input    = 2
num_neuron_output   = 3
input_layer_1 = nn.Linear( num_neuron_input, num_neuron_output)
```



num_neuron_input=**2**

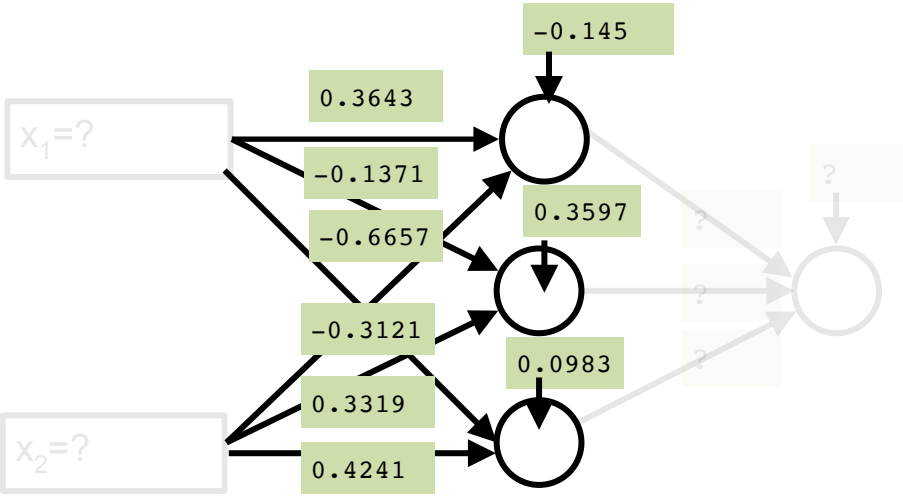
num_neuron_output=**3**

Create MLP Layers

- A **multilayer perceptron:** visualize the weights of the first layer.

```
# Print the weights of the linear layer
print(f'Weights: \n{input_layer_1.weight.data}')
# Print the biases of the linear layer (if they exist)
print(f'Biases: \n{input_layer_1.bias.data}')
```

```
Weights:
tensor([[ 0.3643, -0.3121],
        [-0.1371,  0.3319],
        [-0.6657,  0.4241]])
Biases:
tensor([-0.1455,  0.3597,  0.0983])
```



num_neuron_input=**2**

num_neuron_output=**3**

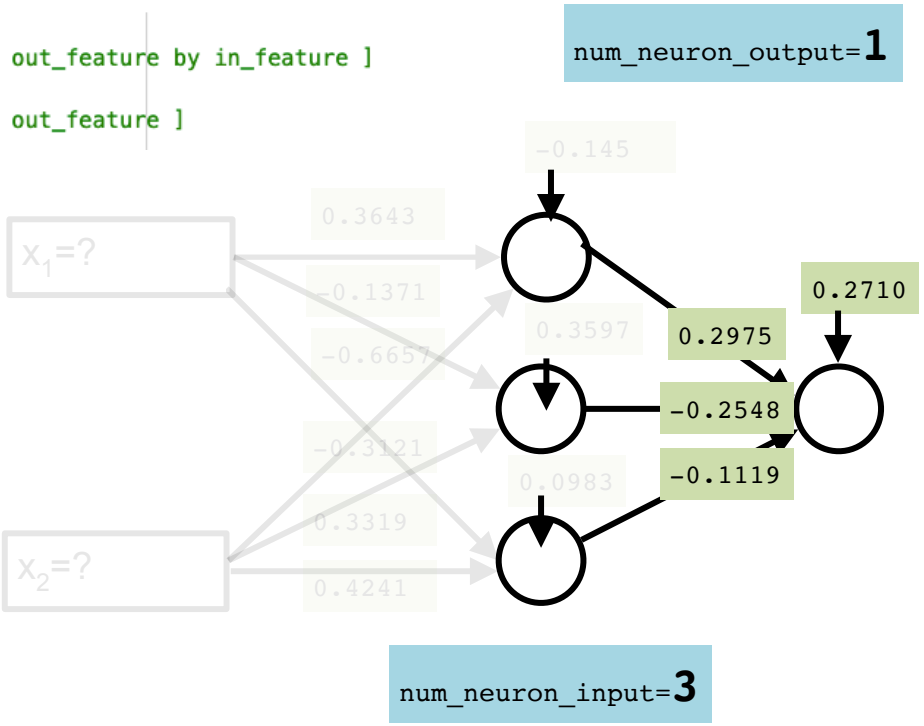
Create MLP Layers

- A **multilayer perceptron:** create the second layer and visualize the weights.

```
torch.manual_seed(1)                    # for reproducibility
num_neuron_input    = 3
num_neuron_output   = 1
output_layer_1      = nn.Linear(num_neuron_input, num_neuron_output)
```

```
Weights:
tensor([[ 0.2975, -0.2548, -0.1119]])
Biases:
tensor([0.2710])
```

```
# print the weights of the linear layer
print(f'Weights: \n{output_layer_1.weight.data}') # matrix of size: [ out_feature by in_feature ]
# print the biases of the linear layer (if they exist)
print(f'Biases: \n{output_layer_1.bias.data}')    # vector of size: [ out_feature ]
```



Generate Random Samples for the MLP

- A **multilayer perceptron** is the simplest type of neural network. It consists of perceptrons (aka nodes, neurons) arranged in layers

```
# Step 1: let's generate 4 random samples of (x1, x2) for the above linear network
torch.manual_seed(0)                    # for reproducibility (you will get the same random number every time you run thi
number_of_samples      = 4
random_input           = torch.randn(number_of_samples, 2) # we have 4 samples each has two predictors (x1 and x2)
print(f'input numbers: \n{random_input.numpy()}\n')
```

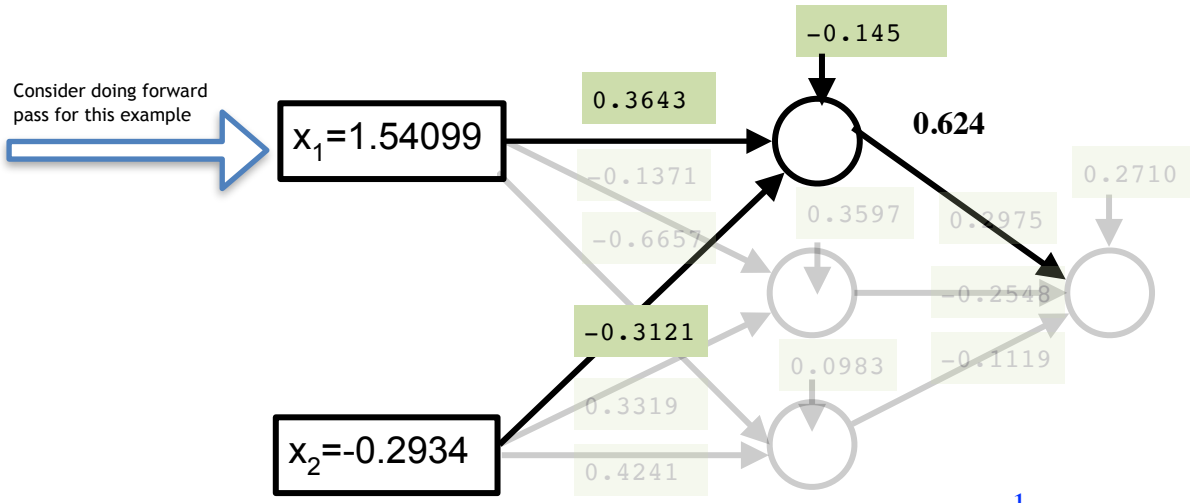
Sample#	x_1	x_2
1	1.5409961	-0.2934289
2	-2.1787894	0.56843126
3	-1.0845224	-1.3985955
4	0.40334684	0.83802634

```
input numbers:
[[ 1.5409961  -0.2934289 ]
 [-2.1787894   0.56843126]
 [-1.0845224  -1.3985955 ]
 [ 0.40334684  0.83802634]]
```

Apply forward pass through the MLP

- Each neuron contains two operations:
  - a dot product between *a weight vector (edges in the graph)* and *an input vector*, which produces a number
  - Then, that number through an activation function, which produces a number as an output
- We can collective do all these dot products in a single layer using a single matrix-matrix multiplication torch.matmul() as follows.
- Also add the bias-term after computing the matrix multiplication

Sample#	x_1	x_2
1	1.5409961	-0.2934289



$$\mathbf{w^T x} = [w_0 \quad w_1 \quad w_2] \begin{bmatrix} 1 \\ 1.54 \\ -0.293 \end{bmatrix} = [-0.145 \quad 0.3643 \quad -0.3121] \begin{bmatrix} 1 \\ 1.54 \\ -0.293 \end{bmatrix} = (-0.145) + 1.54 * 0.3643 + (-0.293) * (-0.3121)$$

$$= 0.507$$

$$= \frac{1}{1 + exp^{-\mathbf{w^T x}}}$$

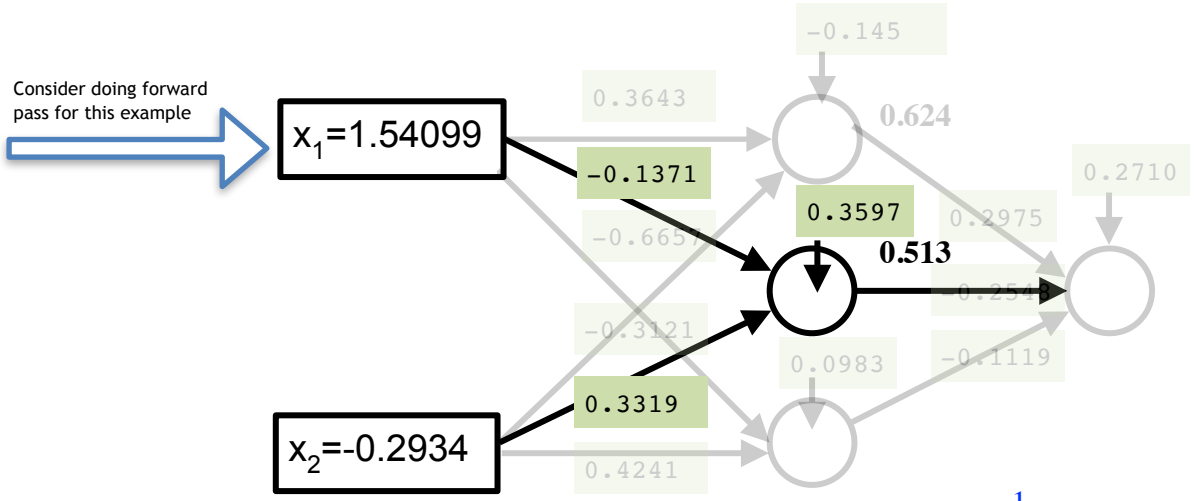
$$= \frac{1}{1 + exp^{-0.507}}$$

$$= 0.624$$

Apply forward pass through the MLP

- Each neuron contains two operations:
  - a dot product between *a weight vector (edges in the graph)* and *an input vector*, which produces a number
  - Then, that number through an activation function, which produces a number as an output
- We can collective do all these dot products in a single layer using a single matrix-matrix multiplication torch.matmul() as follows.
- Also add the bias-term after computing the matrix multiplication

Sample#	x_1	x_2
1	1.5409961	-0.2934289



$$\mathbf{w^T x} = [w_0 \quad w_1 \quad w_2] \begin{bmatrix} 1 \\ 1.54 \\ -0.293 \end{bmatrix} = [0.3597 \quad -0.1371 \quad 0.3319] \begin{bmatrix} 1 \\ 1.54 \\ -0.293 \end{bmatrix} = 0.3597 + (-0.1371) * 1.54 + 0.3319 * (-0.293)$$

$$= 0.0513$$

$$= \frac{1}{1 + exp^{-\mathbf{w^T x}}}$$

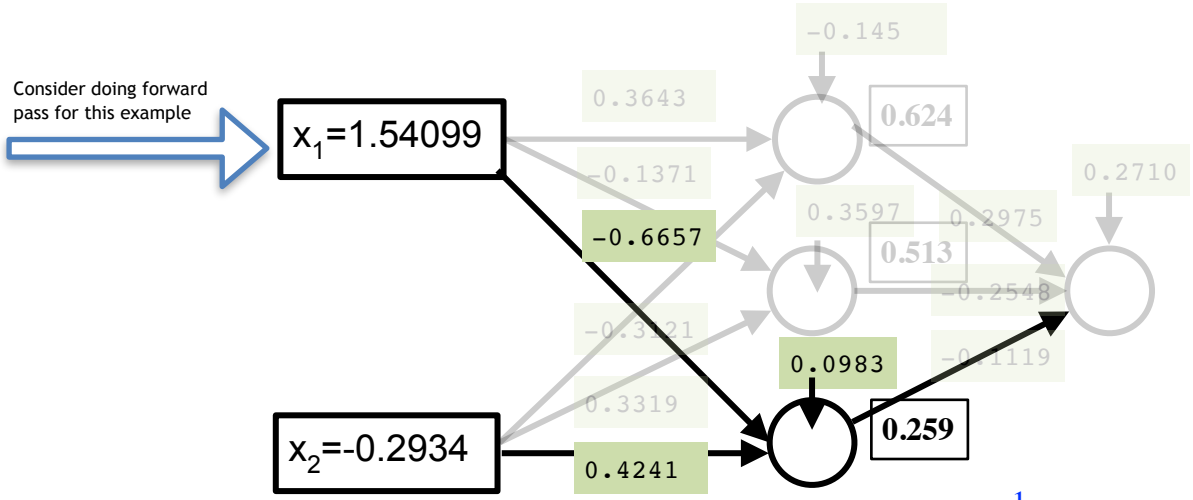
$$= \frac{1}{1 + exp^{-0.0513}}$$

$$= 0.513$$

Apply forward pass through the MLP

- Each neuron contains two operations:
  - a dot product between *a weight vector (edges in the graph)* and *an input vector*, which produces a number
  - Then, that number through an activation function, which produces a number as an output
- We can collective do all these dot products in a single layer using a single matrix-matrix multiplication torch.matmul() as follows.
- Also add the bias-term after computing the matrix multiplication

Sample#	x_1	x_2
1	1.5409961	-0.2934289



$$\mathbf{w^T x} = [w_0 \quad w_1 \quad w_2] \begin{bmatrix} 1 \\ 1.54 \\ -0.293 \end{bmatrix} = [0.0983 \quad -0.6657 \quad 0.4241] \begin{bmatrix} 1 \\ 1.54 \\ -0.293 \end{bmatrix} = 1 * 0.0983 + (-0.6657) * 1.54 + 0.4241 * (-0.293)$$

$$= -1.052$$

$$= \frac{1}{1 + exp^{-\mathbf{w^T x}}}$$

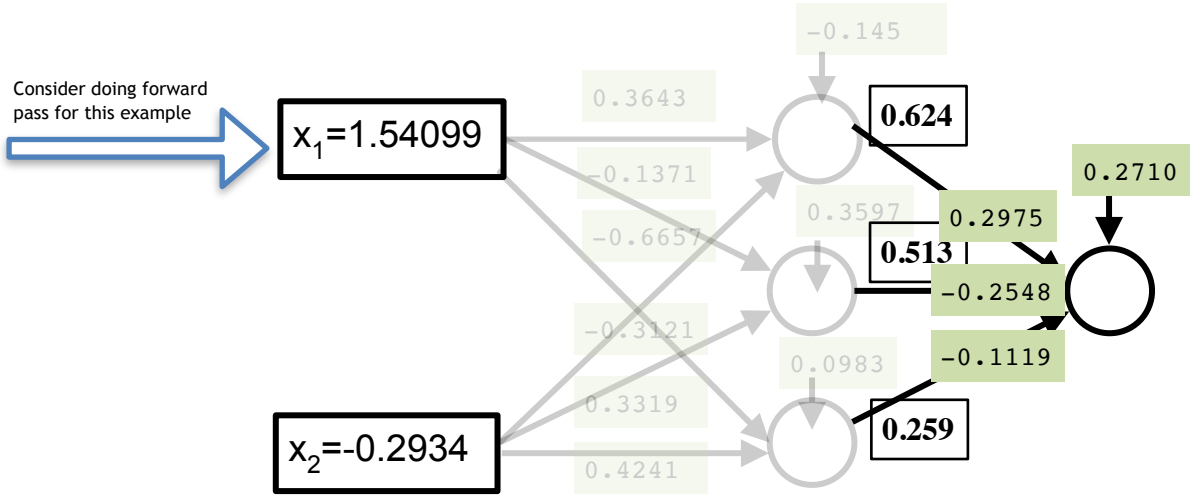
$$= \frac{1}{1 + exp^{1.0519}}$$

$$= 0.259$$

Apply forward pass through the MLP

- Each neuron contains two operations:
  - a dot product between *a weight vector (edges in the graph)* and *an input vector*, which produces a number
  - Then, that number through an activation function, which produces a number as an output
- We can collective do all these dot products in a single layer using a single matrix-matrix multiplication torch.matmul() as follows.
- Also add the bias-term after computing the matrix multiplication

Sample#	x_1	x_2
1	1.5409961	-0.2934289



$$\mathbf{w^T x} = [w_0 \quad w_1 \quad w_2 \quad w_3] \begin{bmatrix} 1 \\ 0.624 \\ 0.513 \\ 0.259 \end{bmatrix}$$

$$= [0.271 \quad 0.2975 \quad -0.2548 \quad -0.1119] \begin{bmatrix} 1 \\ 0.624 \\ 0.513 \\ 0.259 \end{bmatrix} = 0.271 * 1 + 0.2975 * 0.624 + (-0.2548) * 0.513 + (-0.1119) * 0.259$$

$$= 0.2971$$

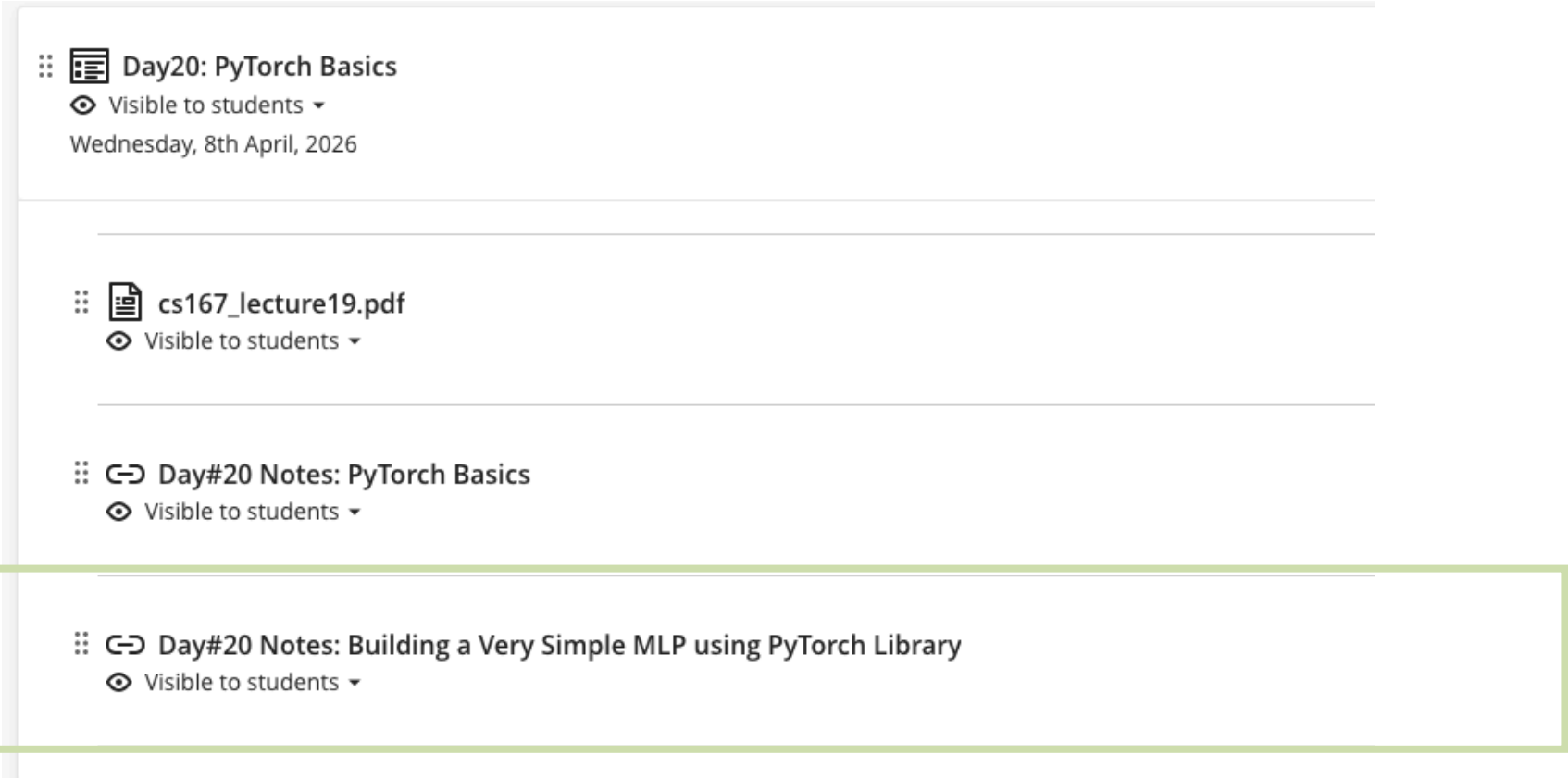
$$= \frac{1}{1 + exp^{-\mathbf{w^T x}}}$$

$$= \frac{1}{1 + exp^{-(0.2971)}}$$

$$= 0.5737$$

PyTorch

- Go to Blackboard and work on the notebook titled “Building a very simple MLP”



https://pytorch.org/

Next lecture: Modular Code Multilayer Perceptron using MLP

- A **multilayer perceptron** is the simplest type of neural network. It consists of perceptrons (aka nodes, neurons) arranged in layers