Introduction to deep learning with PyTorch

I N T R O D U C T I O N T O D E E P L E A R N I N G W I T H P Y T O R C H

By Hina Ali(Exercises collected from multiple sources)

ML vs DL

- Machine learning: relies on hand-crafted feature engineering
- Deep learning: enables feature learning from raw data
- DEEP LEARNING
- Subset of ML
- Inspired by connections in the human brain
- Models require large amount of data

PyTorch: a deep learning framework

- PyTorch is :
- One of the most popular deep learning frameworks
- The framework used in many published deep learning papers intuitive and user-friendly
- Has much in common with NumPy
- **Importing PyTorch and related packages**
  - PyTorch import in Python
  - **import torch**
  - PyTorch supports
  - image data with **torchvision** audio data with **torchaudio** text data with **torchtext**

Tensors: the building blocks of networks in PyTorch

- Load from list

```
import torch

lst = [[1, 2, 3], [4, 5, 6]]
tensor = torch.tensor(lst)
```

- Load from NumPy array

```
np_array = np.array(array)
np_tensor = torch.from_numpy(np_array)
```

Like NumPy arrays, tensors are multidimensional representations of their elements

Tensor attributes

- **Tensor shape**

```
lst = [[1, 2, 3], [4, 5, 6]]
tensor = torch.tensor(lst)
tensor.shape
```

```
torch.Size([2, 3])
```

- Tensor data type

```
tensor.dtype
```

```
torch.int64
```

Tensor device

```
tensor.device
```

```
device(type='cpu')
```

Deep learning often requires a GPU, which, compared to a CPU can offer:

- parallel computing capabilities
- faster training times
- better performance

Getting started with tensor operations

Compatible shapes

```
a = torch.tensor([[1, 1],
                  [2, 2]])

b = torch.tensor([[2, 2],
                  [3, 3]])
```

- Addition / subtraction

```
a + b
```

```
tensor([[3, 3],
        [5, 5]])
```

Incompatible shapes

```
a = torch.tensor([[1, 1],
                  [2, 2]])

c = torch.tensor([[2, 2, 4],
                  [3, 3, 5]])
```

- Addition / subtraction

```
a + c
```

```
RuntimeError: The size of tensor a
(2) must match the size of tensor b (3)
at non-singleton dimension 1
```

Getting started with tensor operations

- Element-wise multiplication

```
a = torch.tensor([[1, 1],
                  [2, 2]])
b = torch.tensor([[2, 2],
                  [3, 3]])
a * b
```

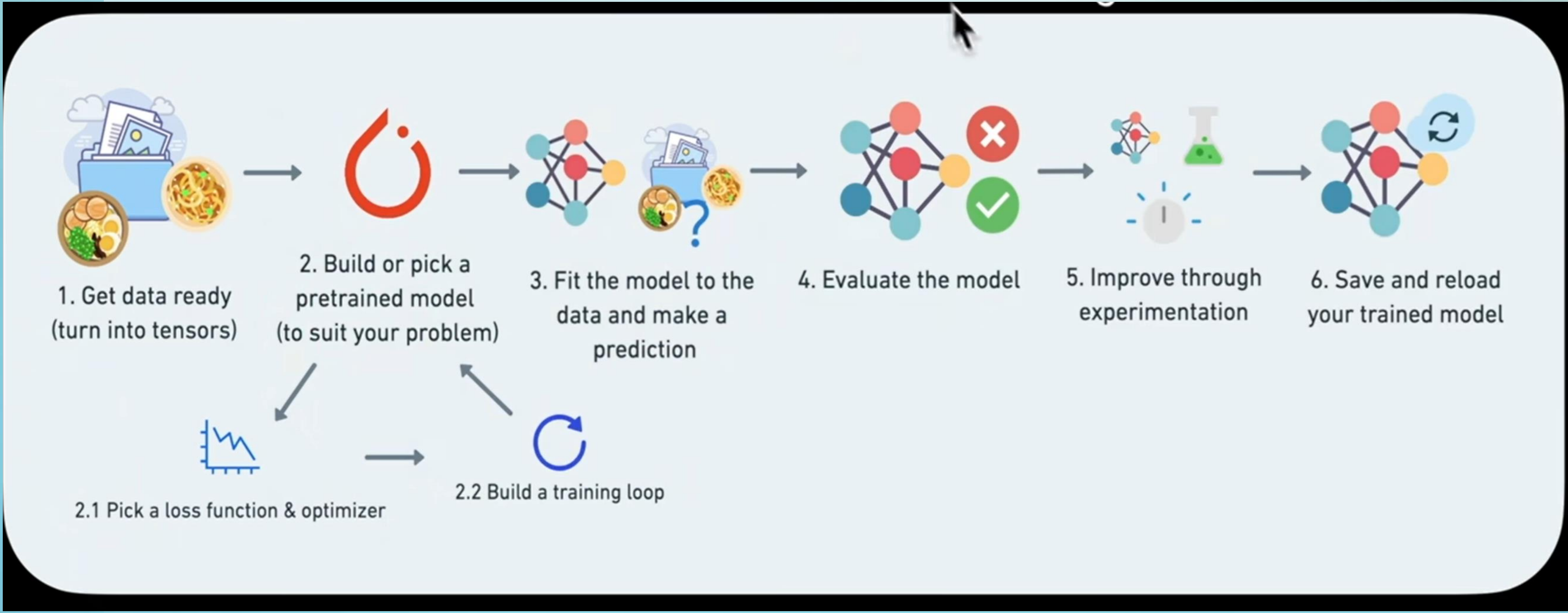
```
tensor([[2,  2],
        [6, 6]])
```

- ... and much more
  - Transposition
  - Matrix multiplication
  - Concatenation
- Most NumPy array operations can be performed on PyTorch tensors

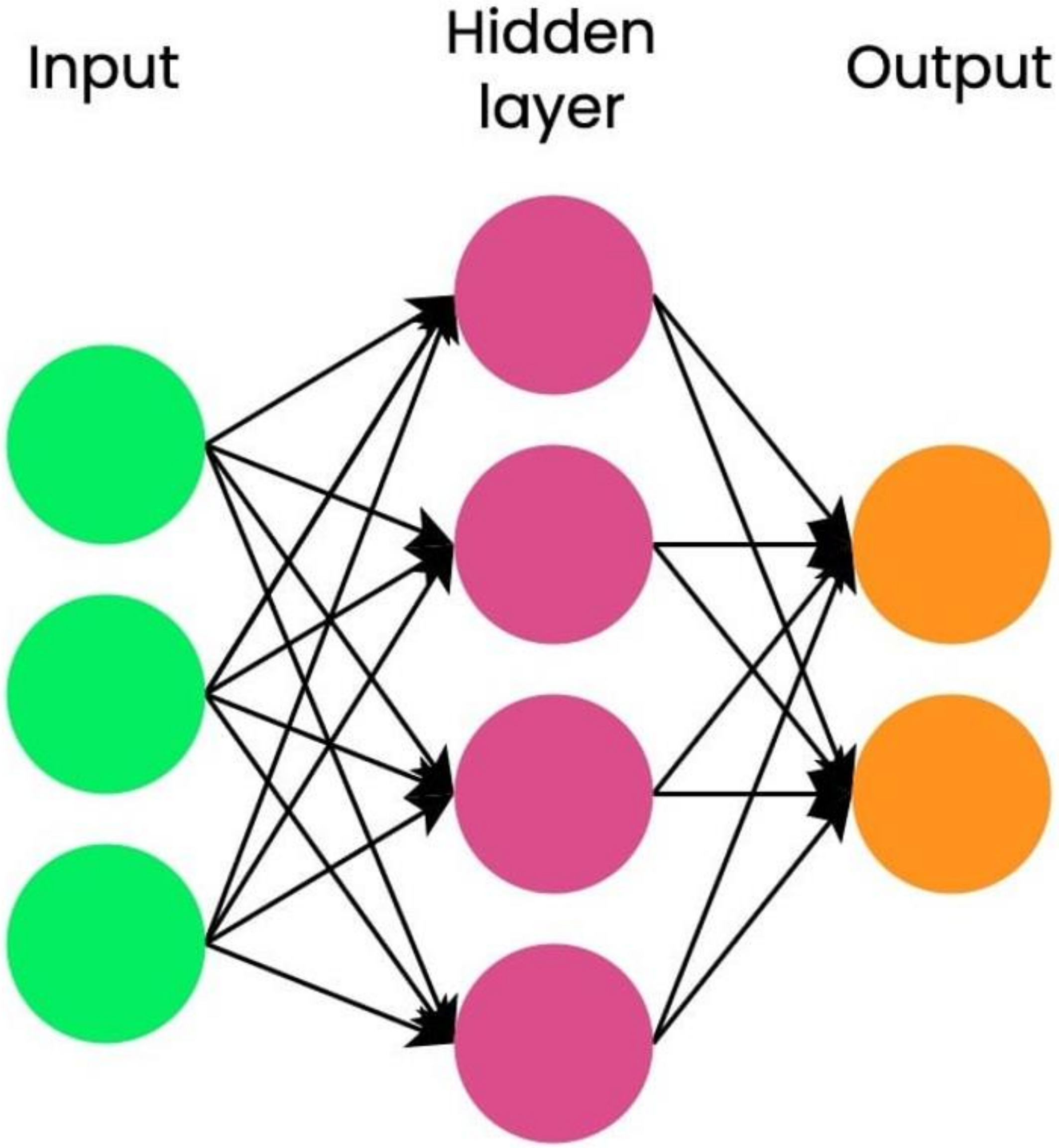
Creating our first neural network

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

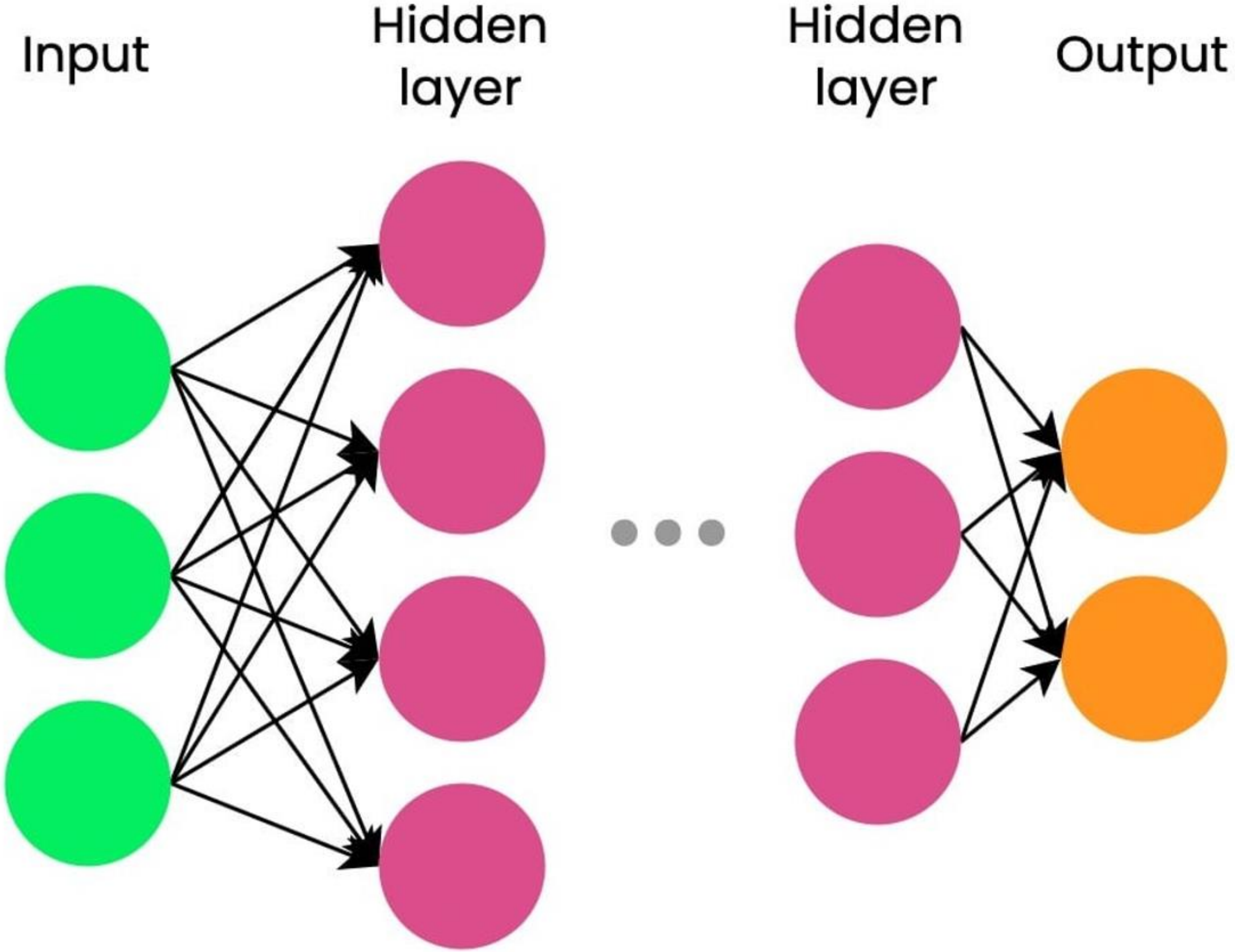
One of the Workflows(End to End)



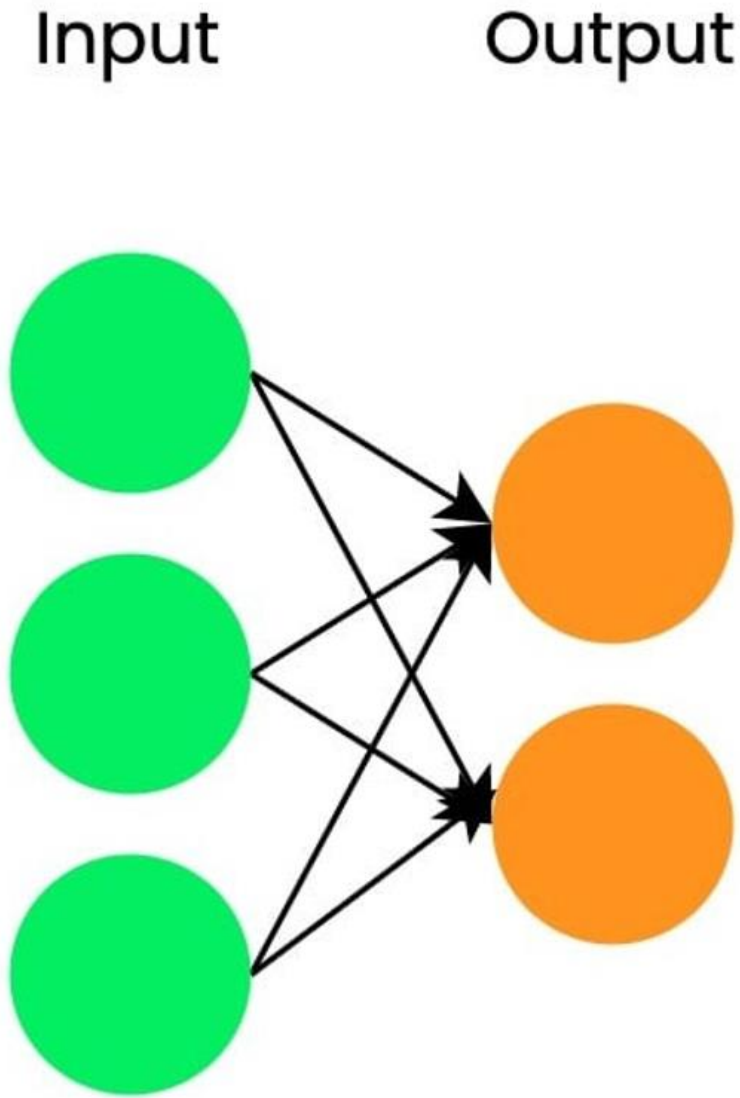
Our first neural network



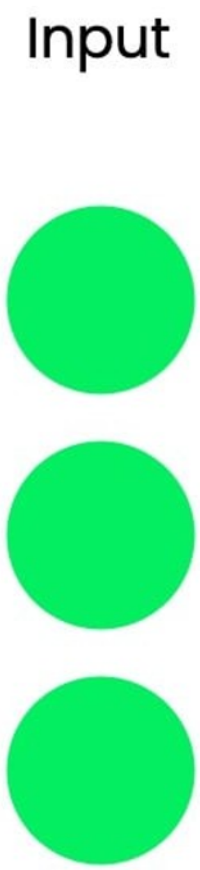
Our first neural network



Our first neural network



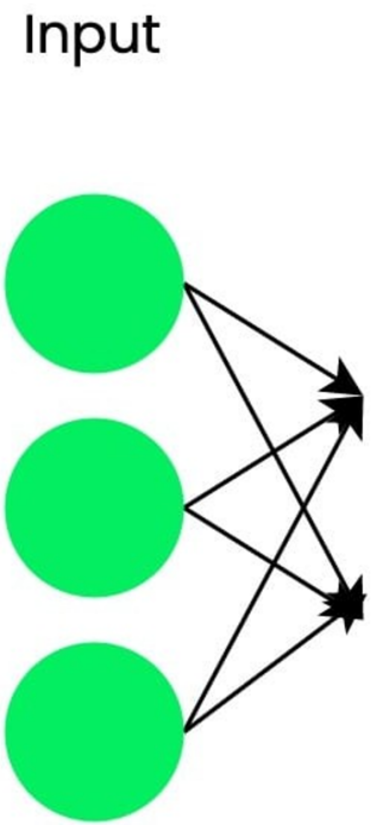
Our first neural network




```python
import torch.nn as nn
```

```python
## Create input_tensor with three features
input_tensor = torch.tensor(
    [[0.3471, 0.4547, -0.2356]]
    )
```

Our first neural network



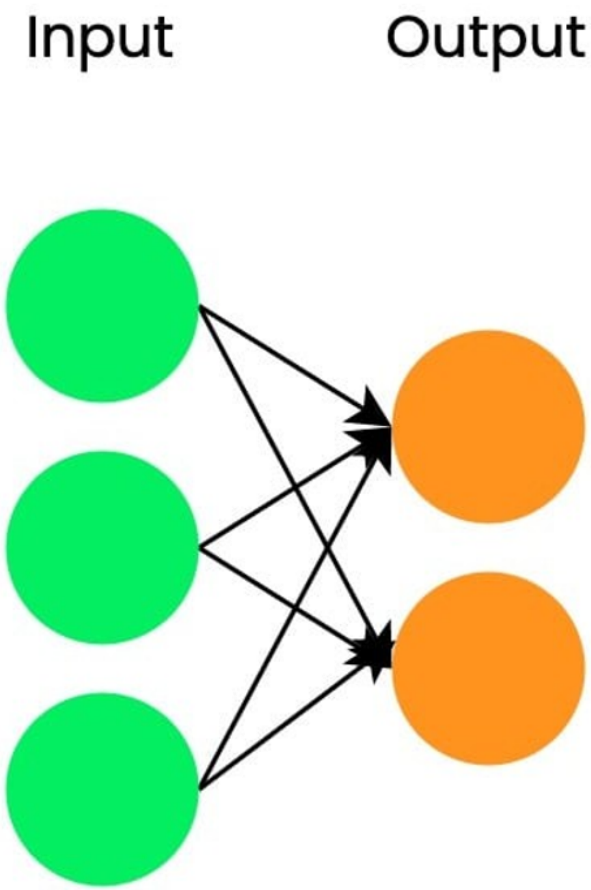

```
import torch.nn as nn
```

```
## Create input_tensor with three features
input_tensor = torch.tensor(
    [[0.3471, 0.4547, -0.2356]])
```

A linear layer takes an input, applies a linear function, and returns output

```
# Define our first linear layer
linear_layer = nn.Linear(in_features=3, out_features=2)
```

Our first neural network




```python
import torch.nn as nn
```

```python
## Create input_tensor with three features
input_tensor = torch.tensor(
    [[0.3471, 0.4547, -0.2356]])
```

```python
# Define our first linear layer
linear_layer = nn.Linear(in_features=3, out_features=2)
```

```python
# Pass input through linear layer
output = linear_layer(input_tensor)
print(output)
```

```
tensor([[-0.2415, -0.1604]],
    grad_fn=<AddmmBackward0>)
```

Getting to know the linear layer operation

Each linear layer has a `.weight` and `.bias` property

```
linear_layer.weight
```

```
Parameter containing:
tensor([[-0.4799,  0.4996,  0.1123],
        [-0.0365, -0.1855,  0.0432]],
       requires_grad=True)
```

```
linear_layer.bias
```

```
Parameter containing:
tensor([0.0310, 0.1537], requires_grad=True)
```

Getting to know the linear layer operation

```
output = linear_layer(input_tensor)
```

For input `x`, weights `W0` and bias `b0`, the linear layer performs

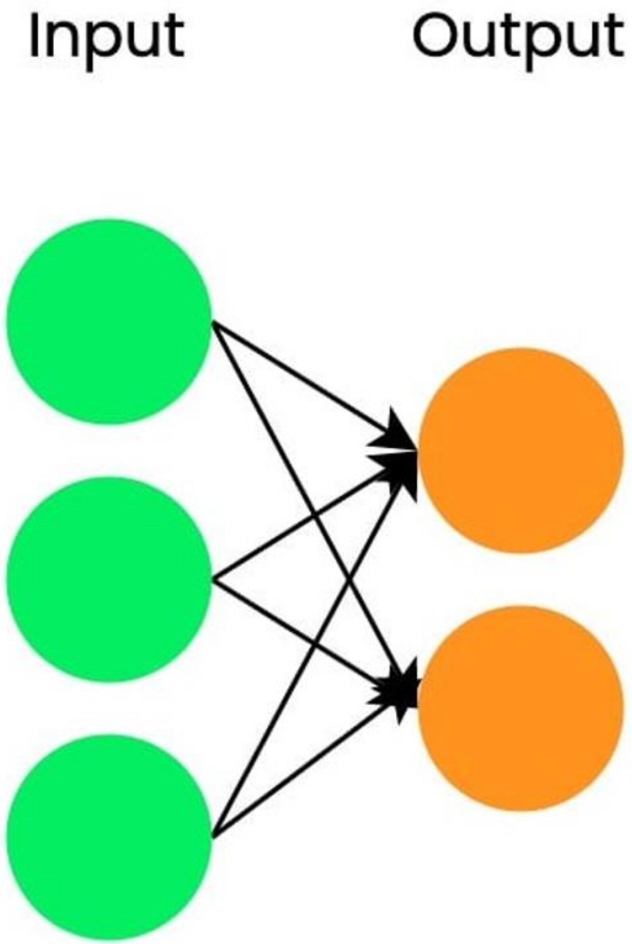
$$y_0 = W_0 \cdot X + b_0$$

In PyTorch: `output = W0 @ input + b0`

- Weights and biases are initialized randomly
- They are not useful until they are tuned

Our two-layer network summary

- Input dimensions: 1×3
- Linear layer arguments:
  - `in_features = 3`
  - `out_features = 2`
- Output dimensions: 1×2
- Networks with only linear layers are called **fully connected**
- Each neuron in one layer is connected to each neuron in the next layer



Stacking layers with nn.Sequential()

```
# Create network with three linear layers
model = nn.Sequential(
    nn.Linear(10, 18),
    nn.Linear(18, 20),
    nn.Linear(20, 5)
)
```

Stacking layers with nn.Sequential()

```
print(input_tensor)
```

```
tensor([[-0.0014,  0.4038,  1.0305,  0.7521, 0.7489, -0.3968,  0.0113, -1.3844, 0.8705, -0.9743]])
```

```
# Pass input_tensor to model to obtain output
output_tensor = model(input_tensor)
print(output_tensor)
```

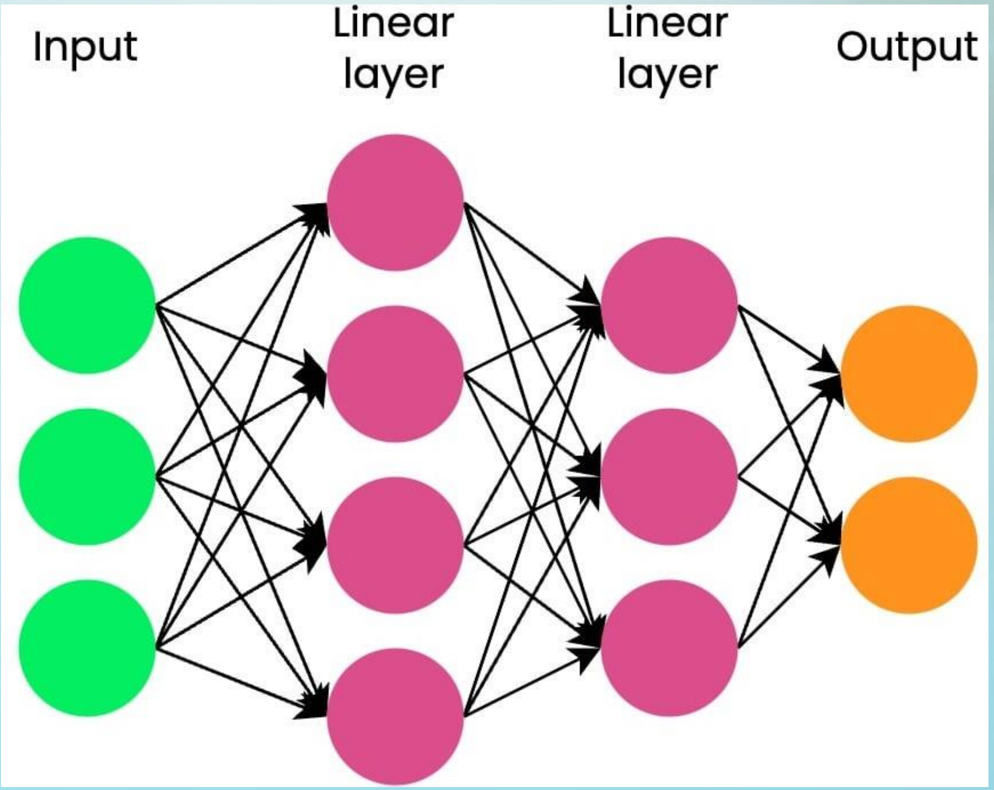
```
tensor([[-0.0254, -0.0673,  0.0763,
0.0008,  0.2561]], grad_fn=<AddmmBackward0>)
```

- We obtain output of 1×5 dimensions
- Output is still not yet meaningful

Discovering activation functions

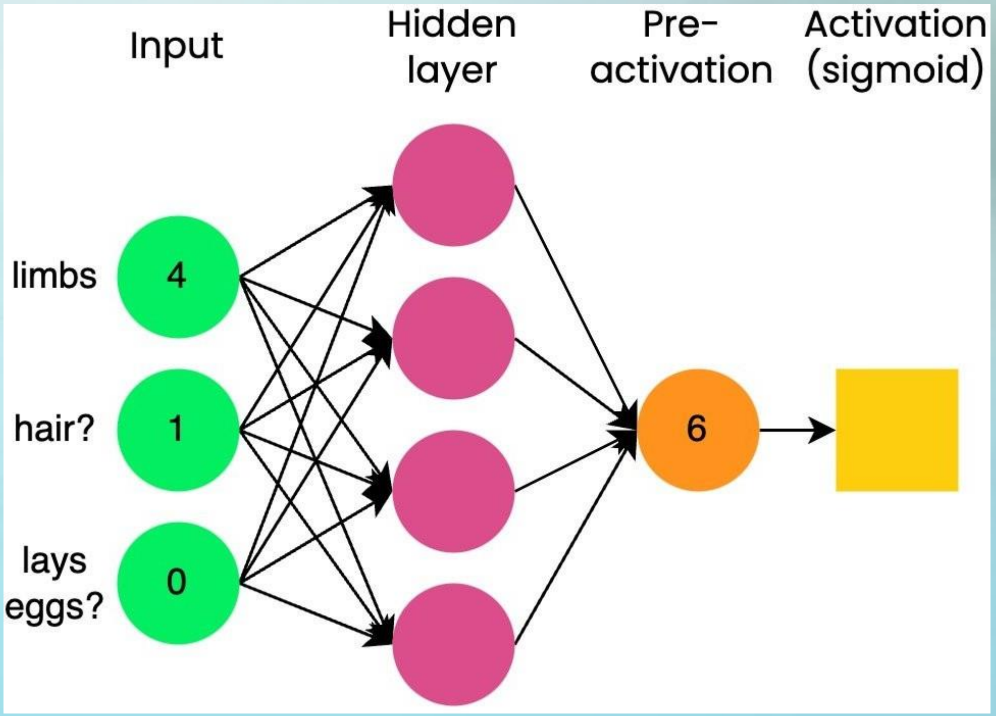
INTRODUCTION TO DEEPLEARNING WITH PYTORCH

- Stacked linear operations
- We have only seen linear layer networks
- Each linear layer multiplies its respective input with layer weights and adds biases
- Even with multiple stacked linear layers, output still has linear relationship with input



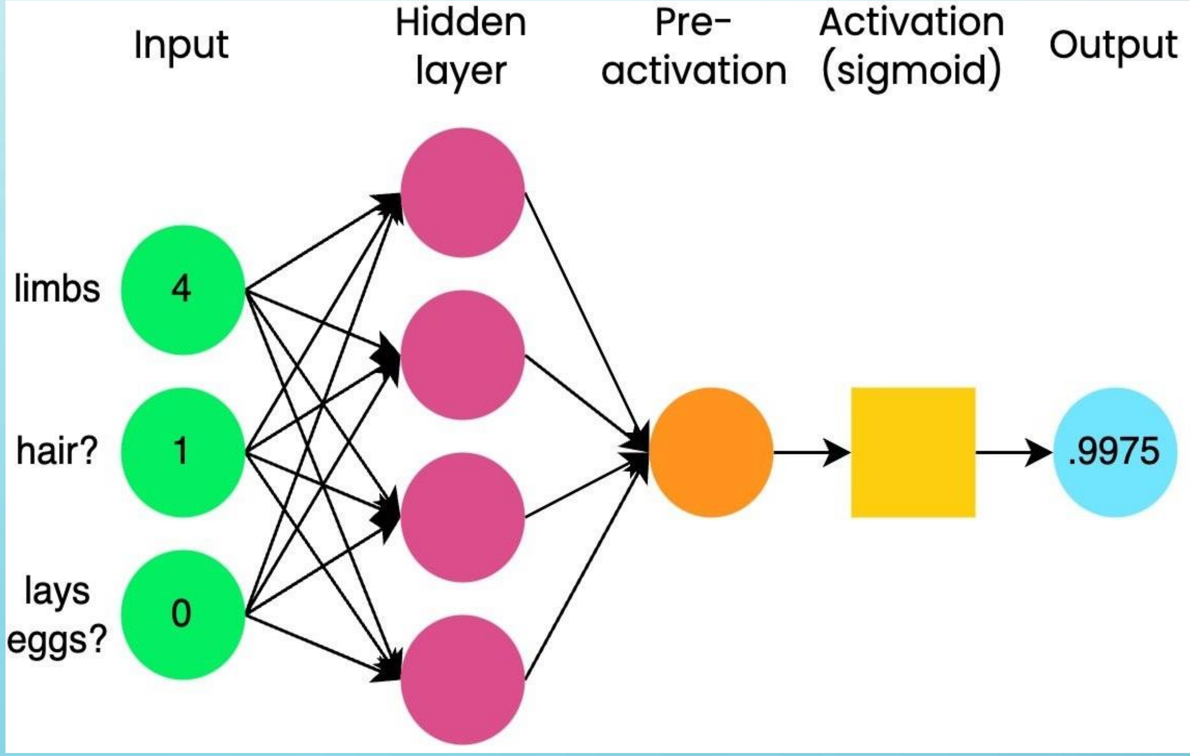
Why do we need activation functions?

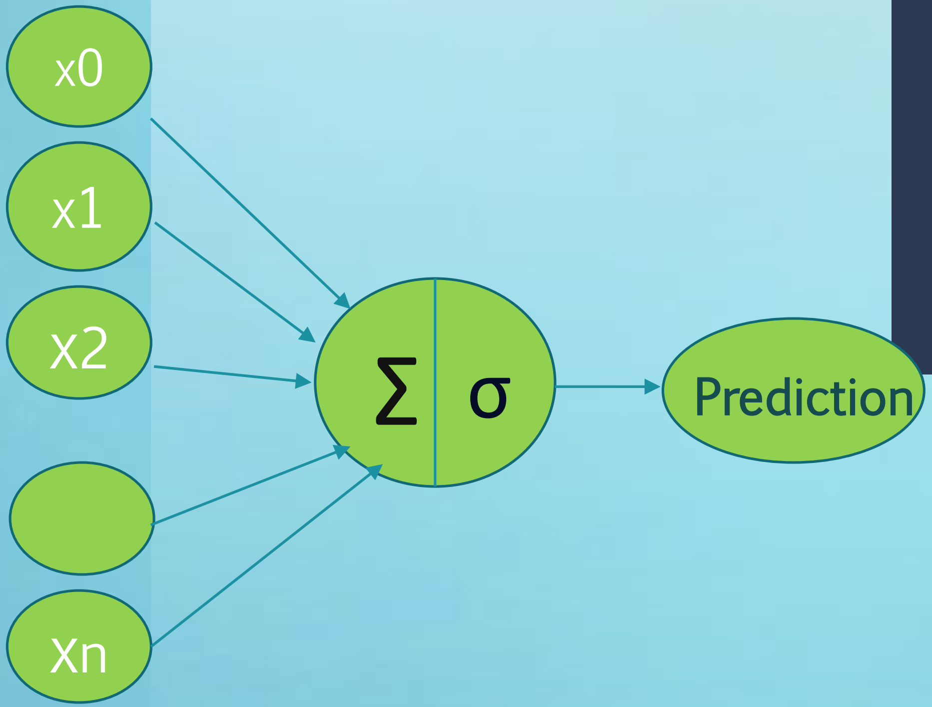
- Activation functions add non-linearity to the network
- A model can learn more complex relationships with non-linearity
- <u>SIGMOID Activation Function(</u> *first introduced with the algorithm of Logistic regression aka logistic function*<u>)</u>
- **Binary Classification**
- To predict whether animal is 1 (mammal) or 0 (not mammal),
- we take the pre-activation (6), pass it to the sigmoid

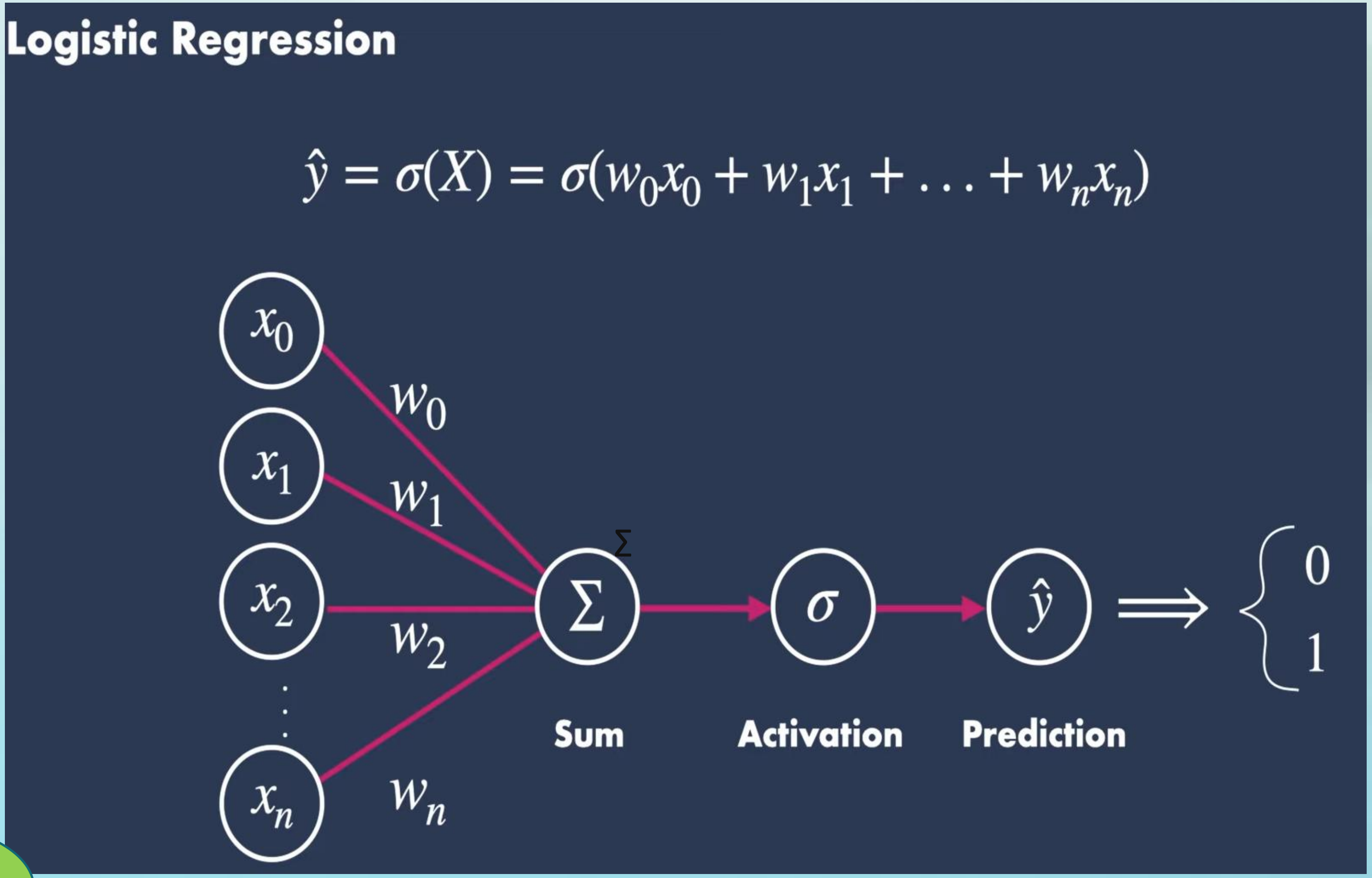


Binary Classification

- Obtain a value between 0 and 1. shows probability of x belonging to a particular class.
- Using the common threshold of 0.5:
  - If output is > 0.5, class label = 1 (mammal)
  - If output is <= 0.5, class label = 0 (not mammal)




x0
x1
x2
Xn
Σ
σ
Prediction


Logistic Regression
$\hat{y} = \sigma(X) = \sigma(w_0x_0 + w_1x_1 + \ldots + w_nx_n)$
x_0
x_1
x_2
x_n
w_0
w_1
w_2
w_n
Σ
σ
$\hat{y}$
0
1
Sum
Activation
Prediction

Meet the sigmoid function

```python
import torch
import torch.nn as nn


input_tensor = torch.tensor([[6.0]])
sigmoid = nn.Sigmoid()
output = sigmoid(input_tensor)
```

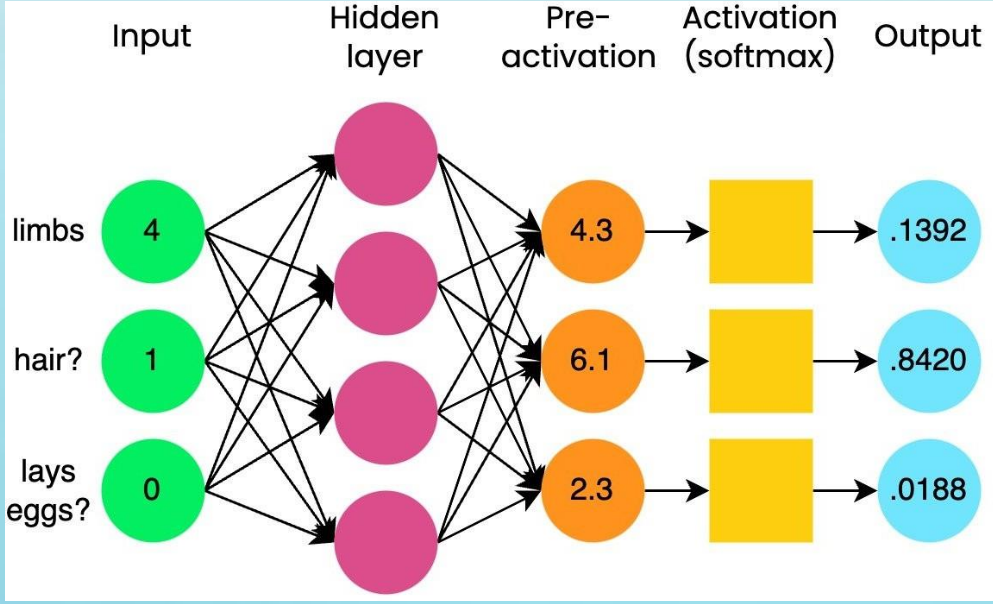
```
tensor([[0.9975]])
```

Activation function as the last layer

```
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, 1), # Second linear layer
  nn.Sigmoid() # Sigmoid activation function
)
```

Note. Sigmoid as last step in network of linear layers is **equivalent** to traditional logistic regression.

Softmax



- used for multi-class classification problems
- takes N-element vector as input and outputs vector of same size
- say N=3 classes:
- bird (0), mammal (1), reptile (2)
- output has three elements, so softmax has three elements
- outputs a probability distribution:
- each element is a probability (it's bounded between 0 and 1)
- the sum of the output vector is equal to 1

Getting acquainted with softmax

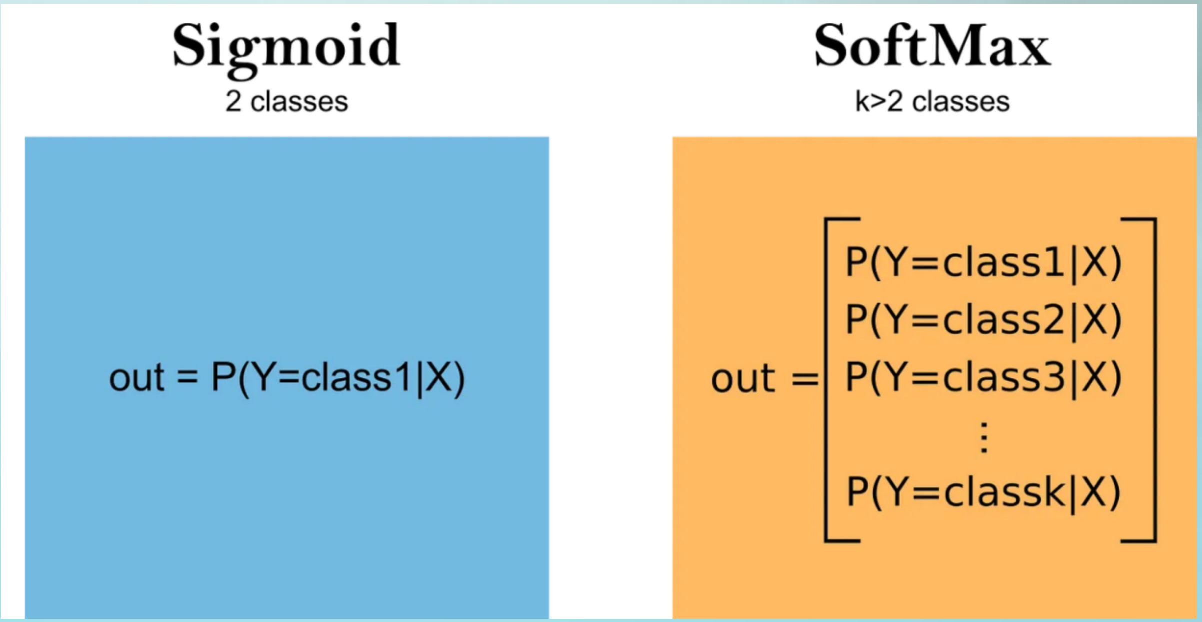
```
import torch
import torch.nn as nn

# Create an input tensor
input_tensor = torch.tensor(
    [[4.3, 6.1, 2.3]])

# Apply softmax along the last dimension
probabilities = nn.Softmax(dim=-1)
output_tensor = probabilities(input_tensor)

print(output_tensor)
```

```
tensor([[0.1392, 0.8420, 0.0188]])
```

- dim = -1 indicates softmax is applied to the input tensor's last dimension
- `nn.Softmax()` can be used as last step in `nn.Sequential()`

Sigmoid vs SoftMax

- **But if both functions map the same transformation. Difference between them?**
- Sigmoid is used for **binary classification** methods where we only have 2 classes,
- while SoftMax applies to **multiclass problems**. In fact, the SoftMax function is an extension of the Sigmoid function.



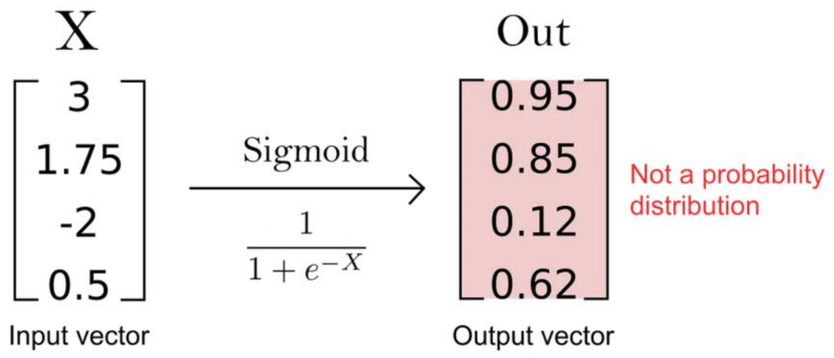
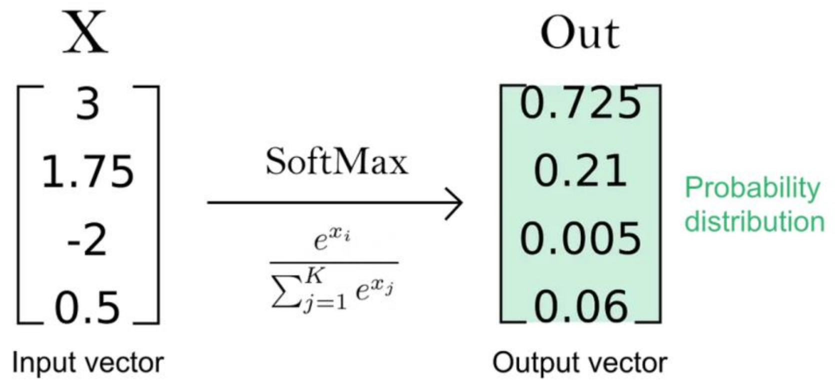


Figure 3. Why sigmoid function can't be used for multiclass classification. Notice that the output vector elements don't add up to 1. Image by author



gure 5. Using SoftMax we obtain a probability distribution over all the predicted classes. Note: The results have been approximated to 3 decimal places to facilitate reading. Image by author