

Introduction to deep learning with PyTorch

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



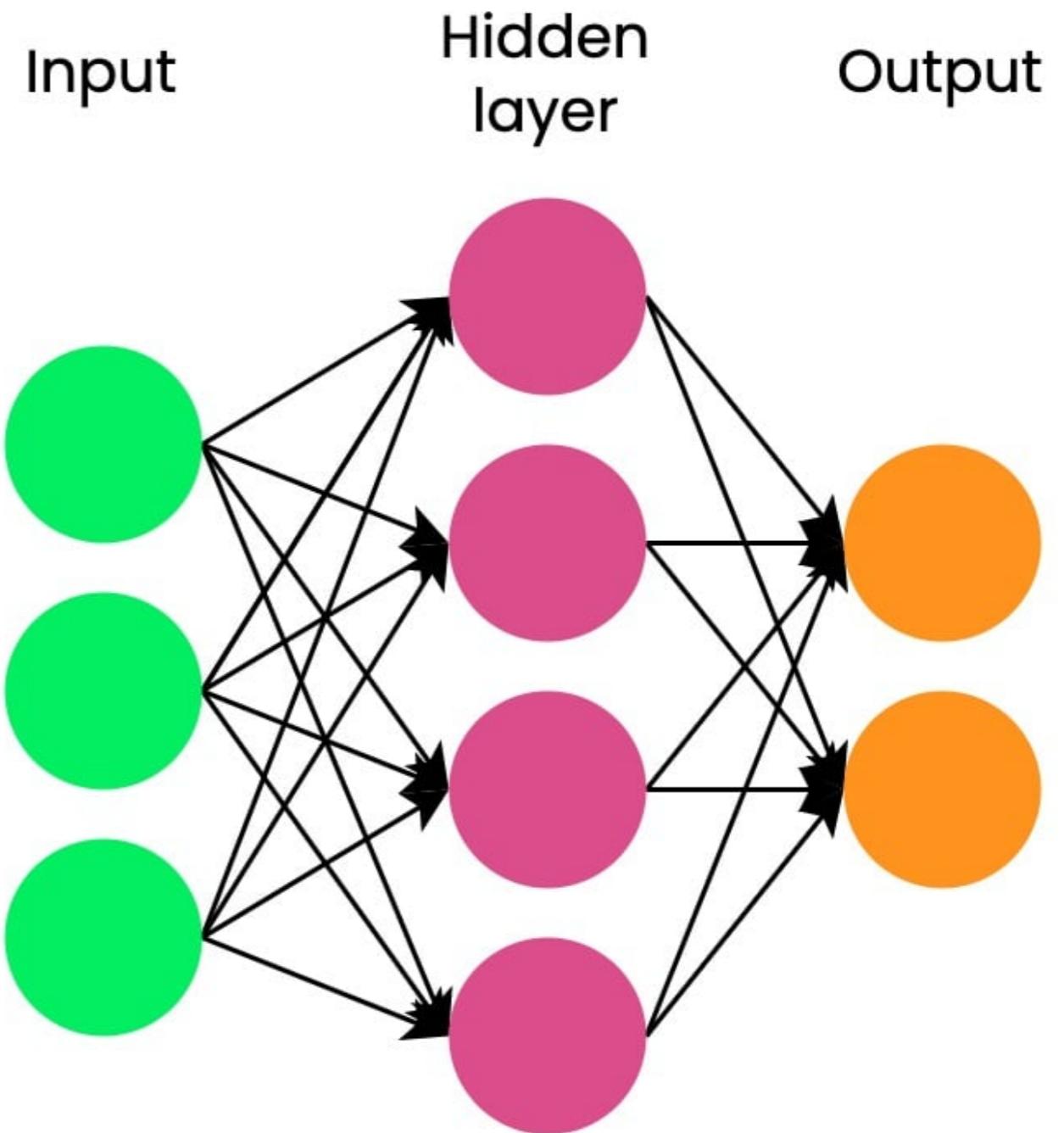
What is deep learning?

- Deep learning is everywhere:
 - Language translation
 - Self-driving cars
 - Medical diagnostics
 - Chatbots
- Used on multiple data types: **images**, **text** and **audio**
- Traditional machine learning: relies on hand-crafted **feature engineering**
- Deep learning: enables **feature learning** from raw data



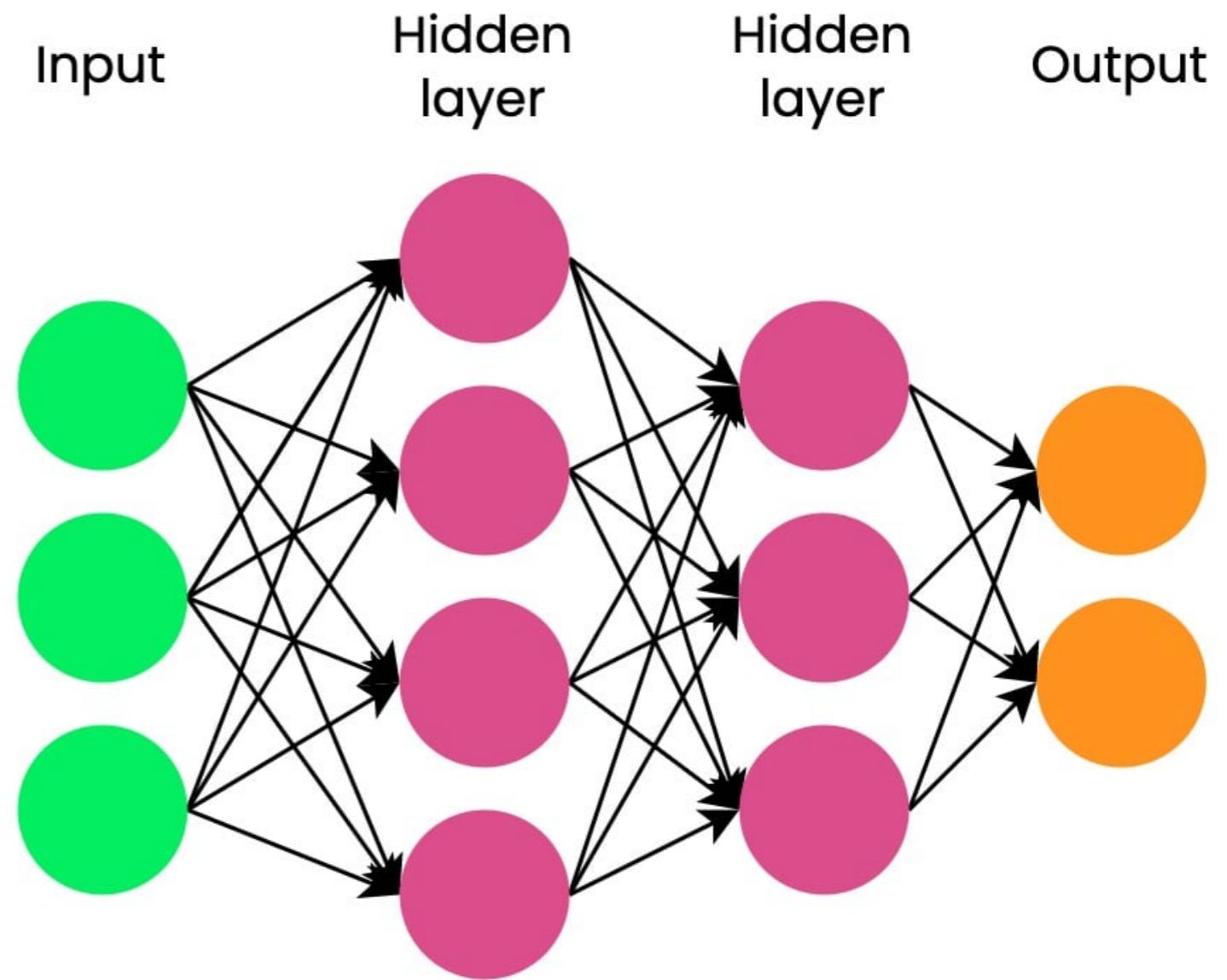
What is deep learning?

- Deep learning is a subset of machine learning



What is deep learning?

- Deep learning is a subset of machine learning
- 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
```

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

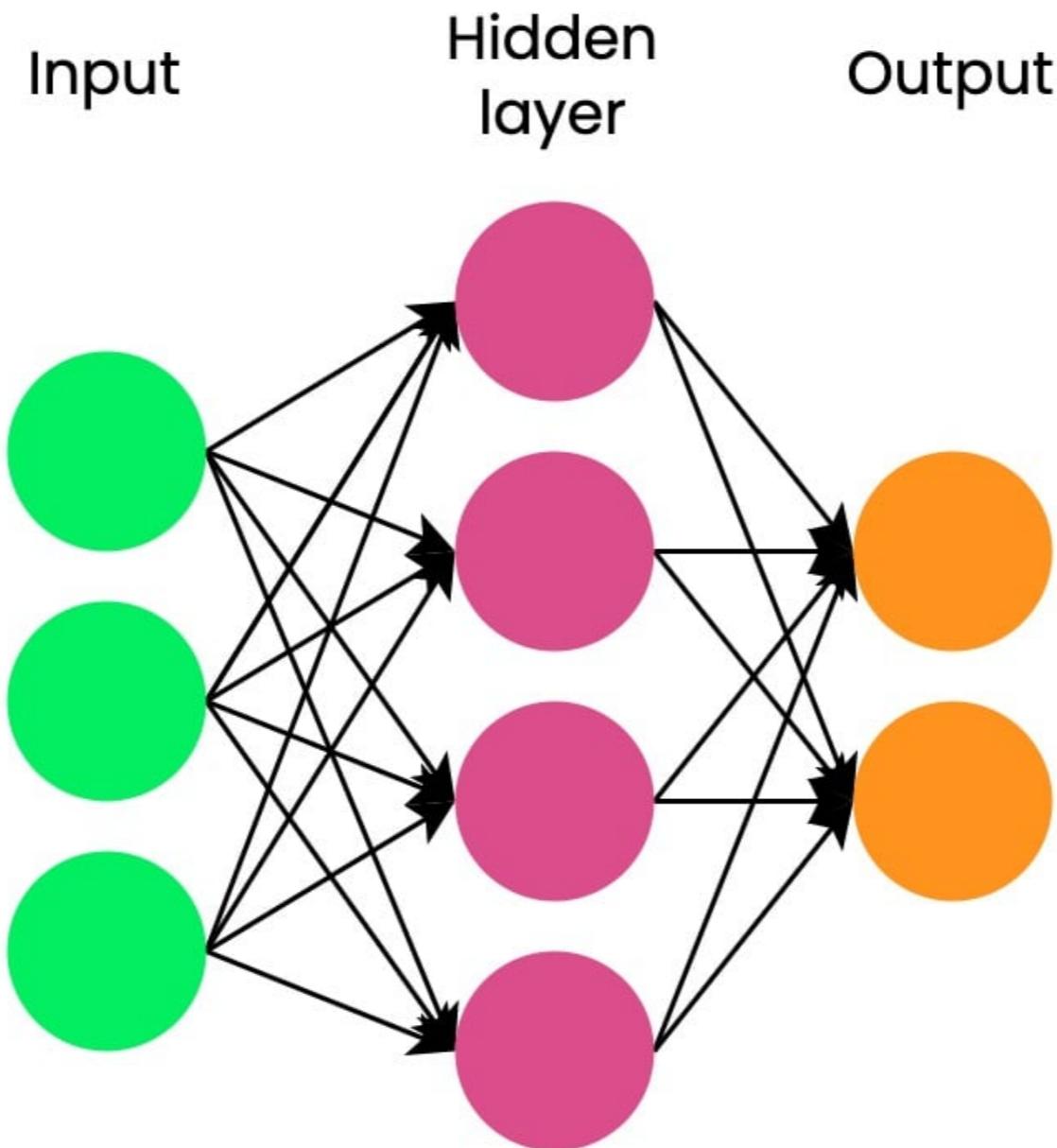
```
tensor([[2, 2],  
       [6, 6]])
```

Creating our first neural network

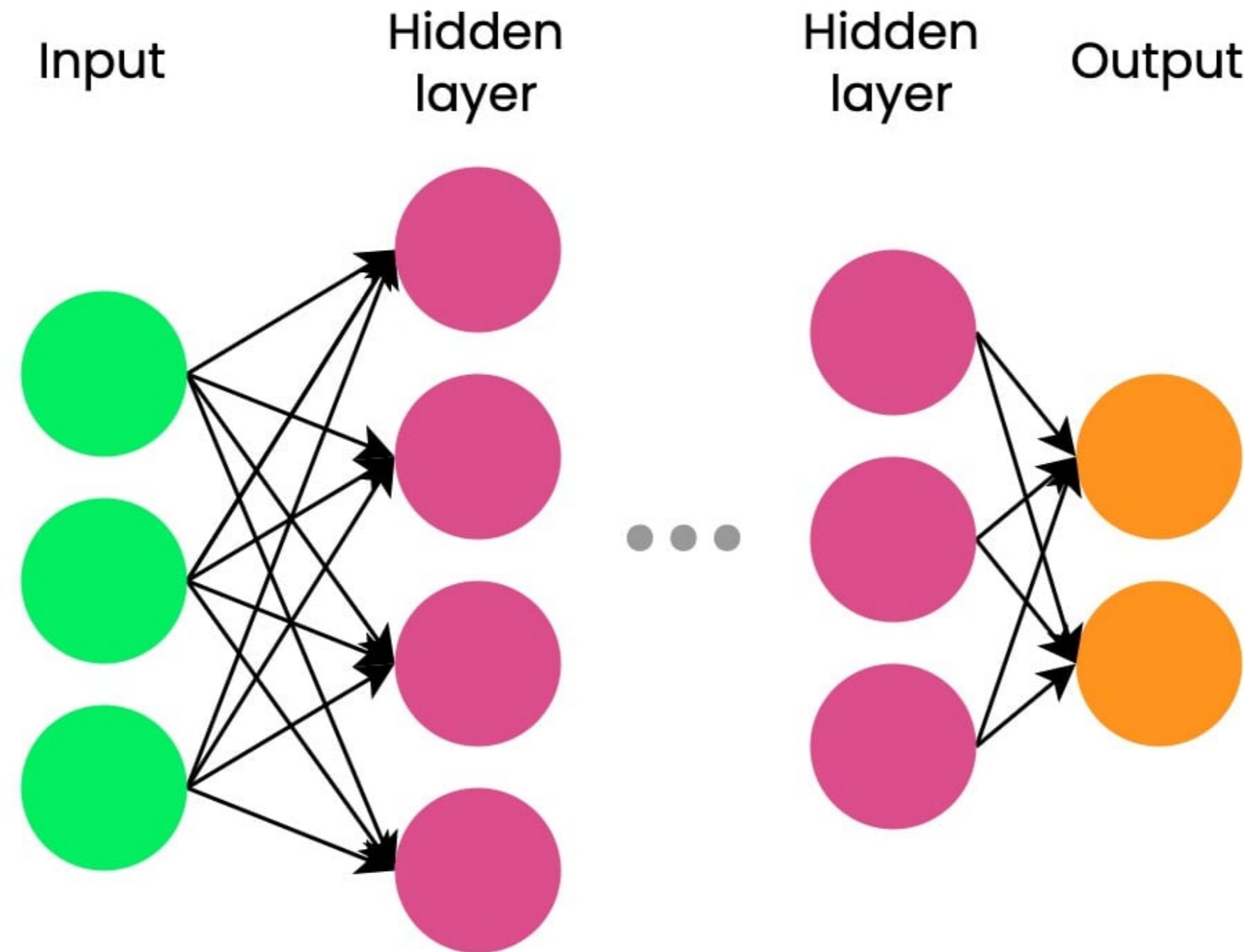
INTRODUCTION TO DEEP LEARNING WITH PYTORCH



Our first neural network



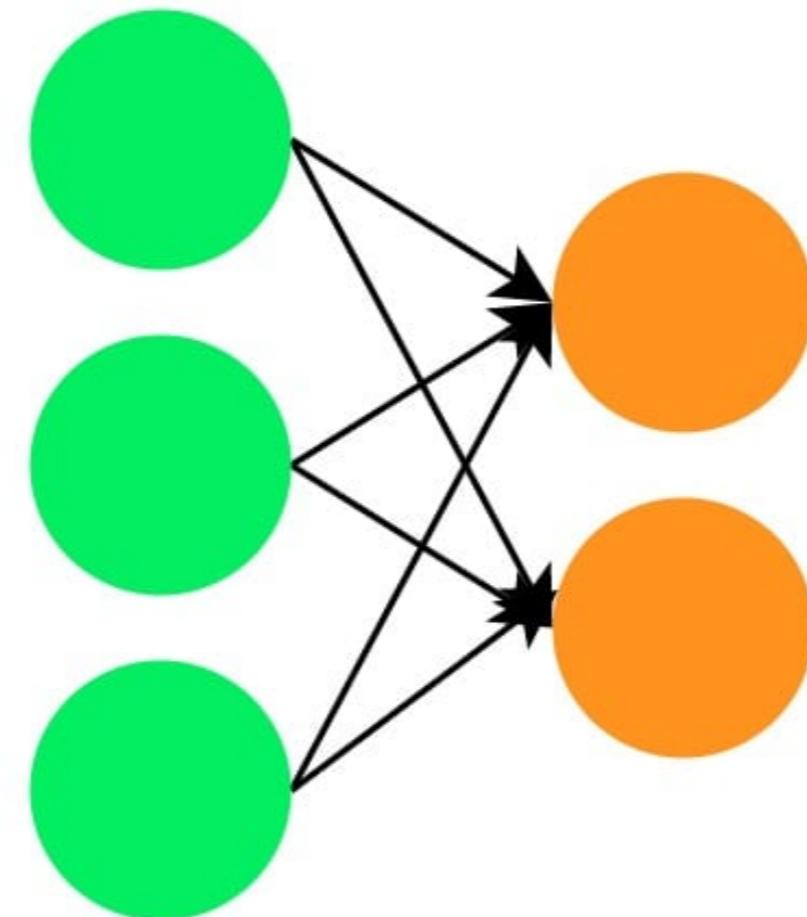
Our first neural network



Our first neural network

Input

Output



Our first neural network

Input

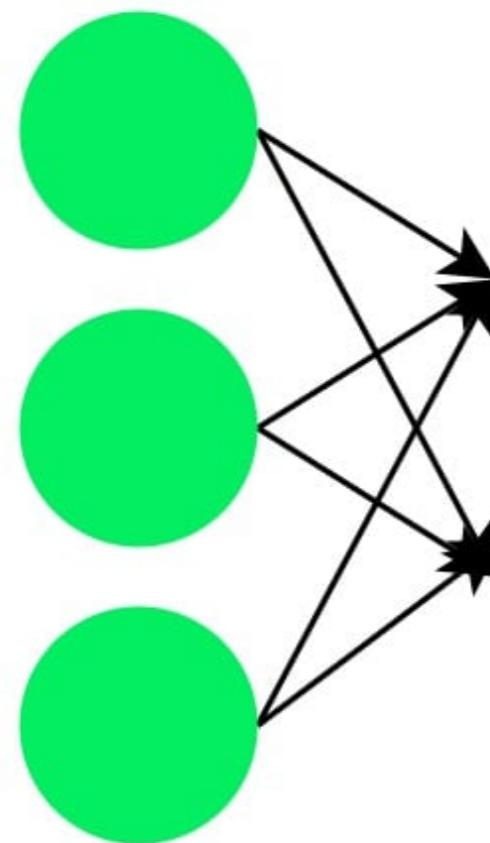


```
import torch.nn as nn
```

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

Our first neural network

Input



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

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

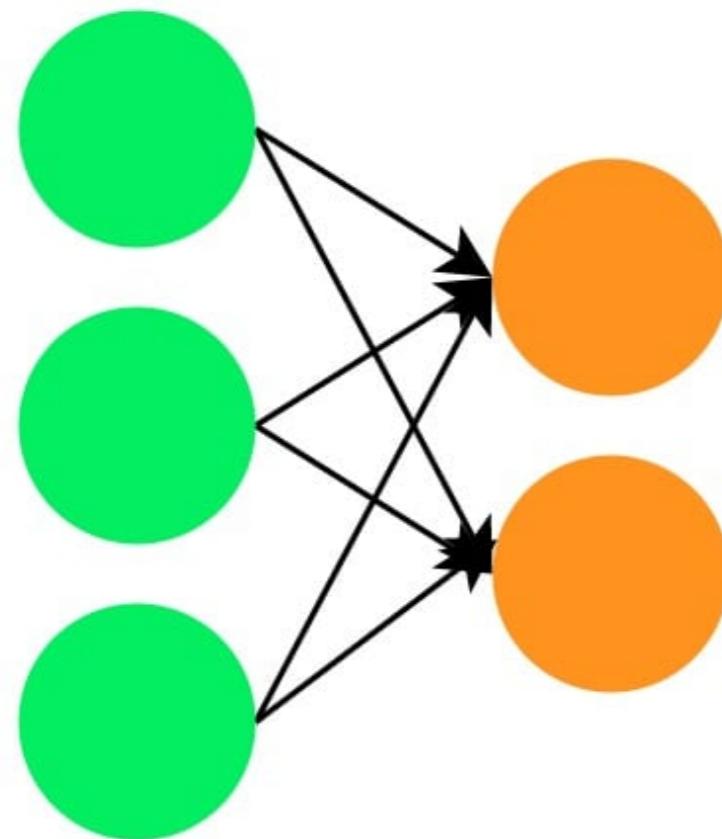
A linear layer takes an input, applies a linear function, and returns output

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

Our first neural network

Input

Output



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

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

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

```
# 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`

```
linear_layer.weight
```

Parameter containing:

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

and `.bias` property

```
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 W_0 and bias b_0 , the linear layer performs

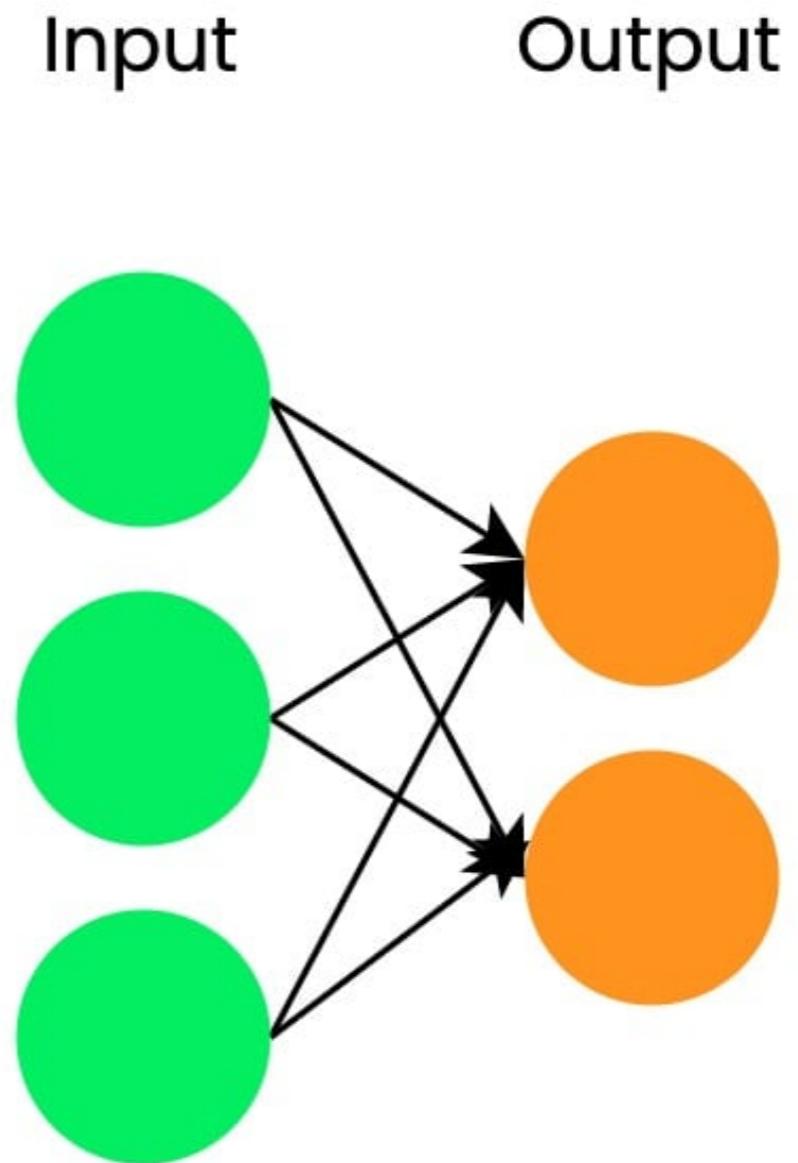
$$y_0 = W_0 \cdot X + b_0$$

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

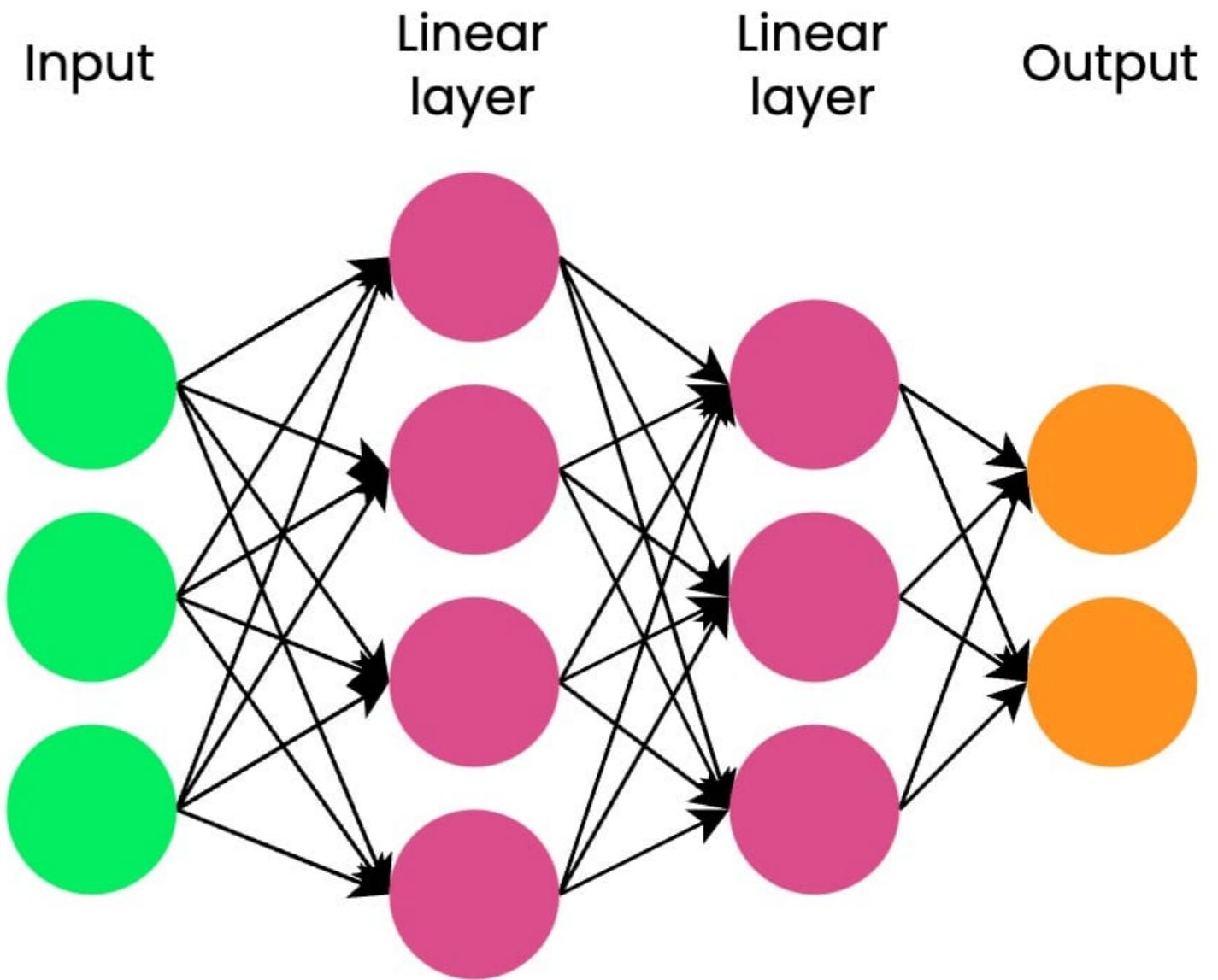
Discovering activation functions

INTRODUCTION TO DEEP LEARNING WITH PYTORCH



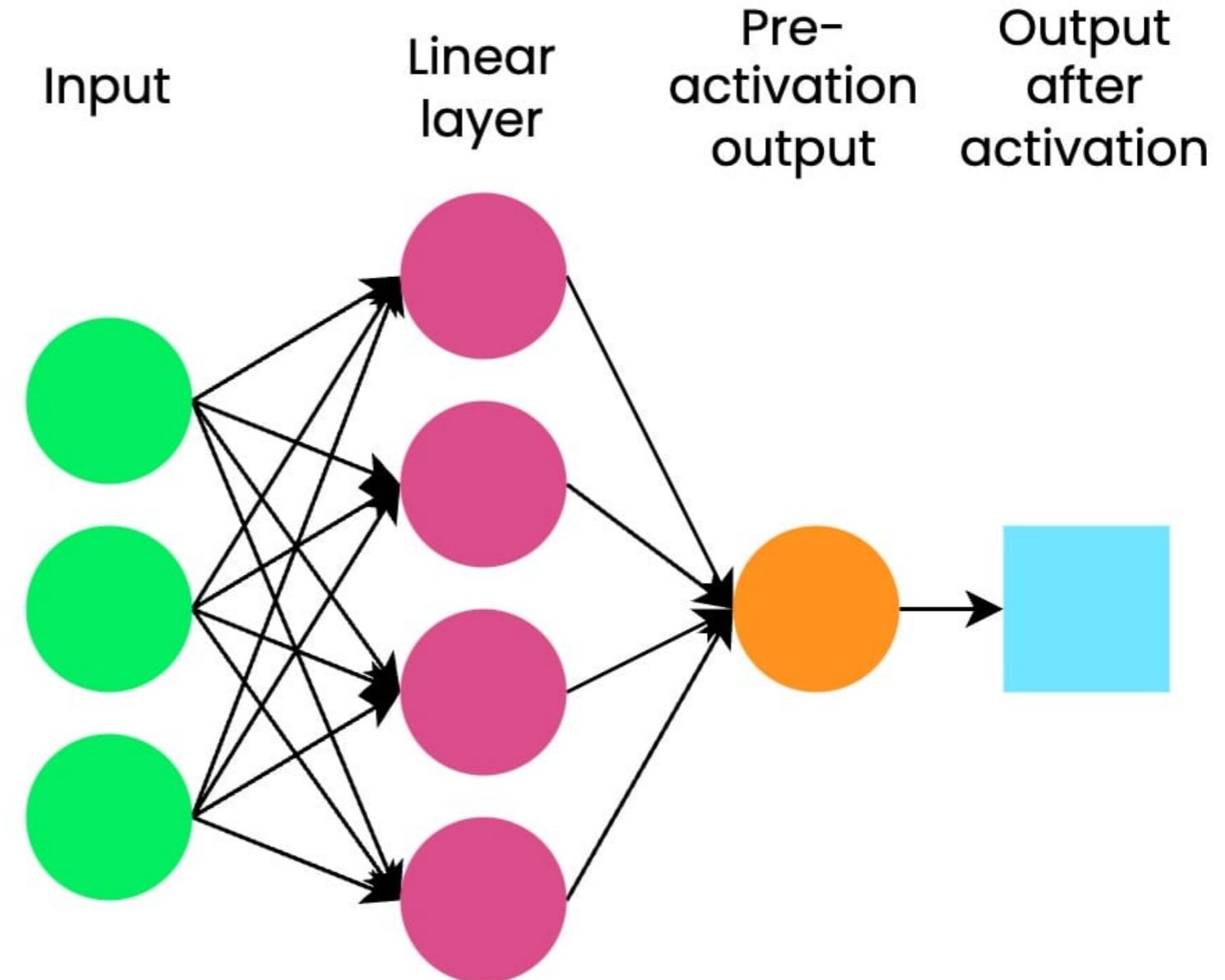
Stacked linear operations

- 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?

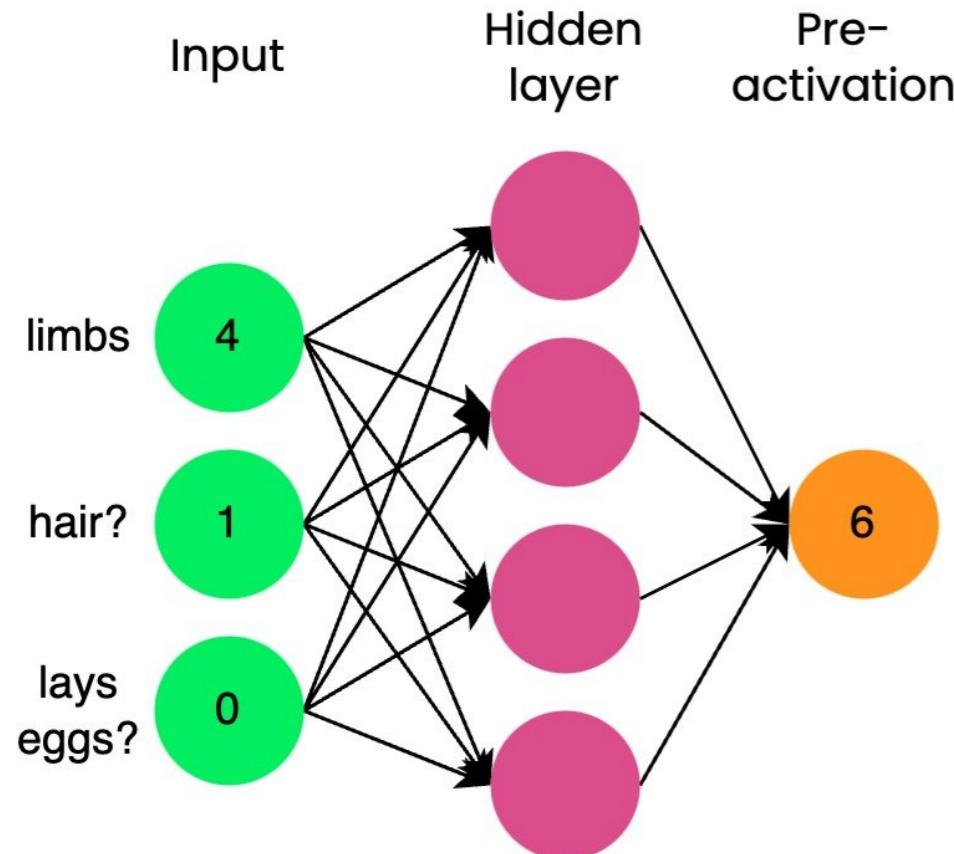
- Activation functions add **non-linearity** to the network
- A model can learn more **complex** relationships with non-linearity



Meet the sigmoid function

Binary classification task:

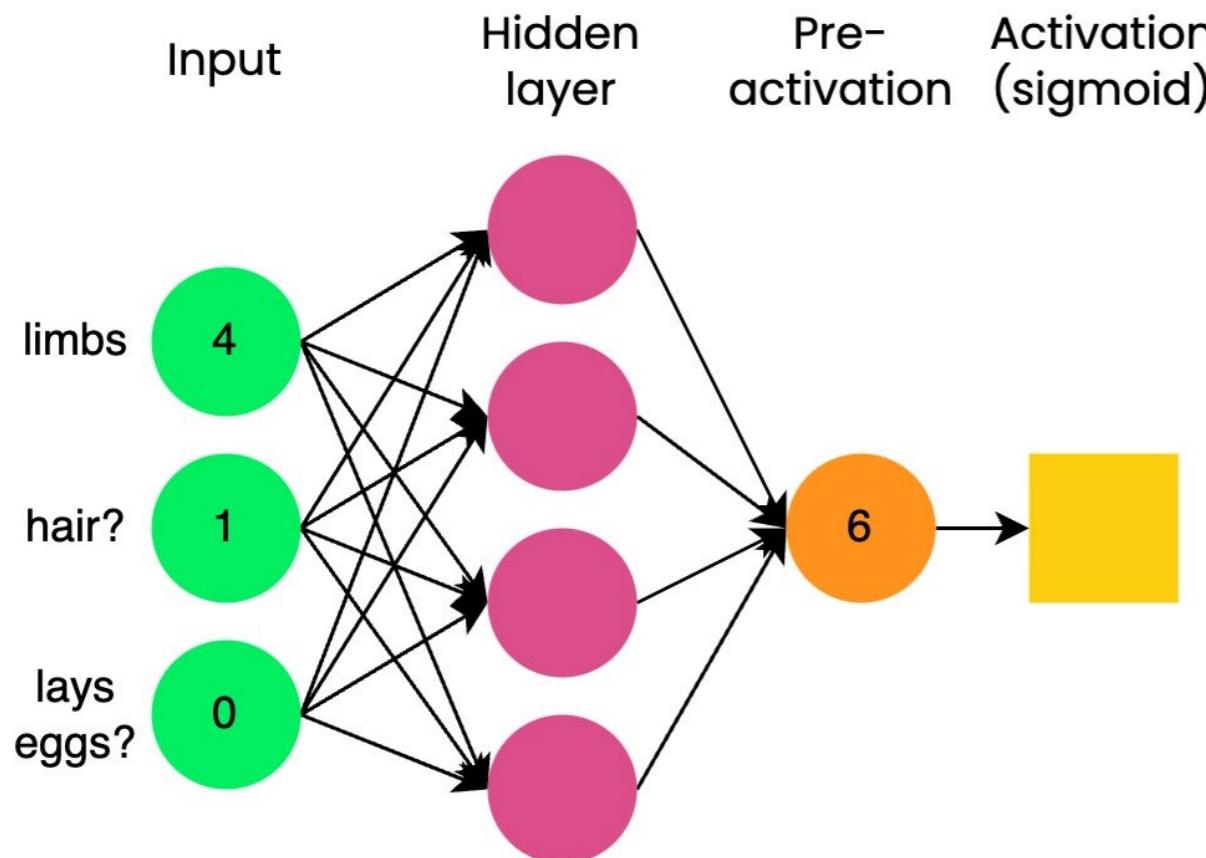
- To predict whether animal is 1 (**mammal**) or 0 (**not mammal**),



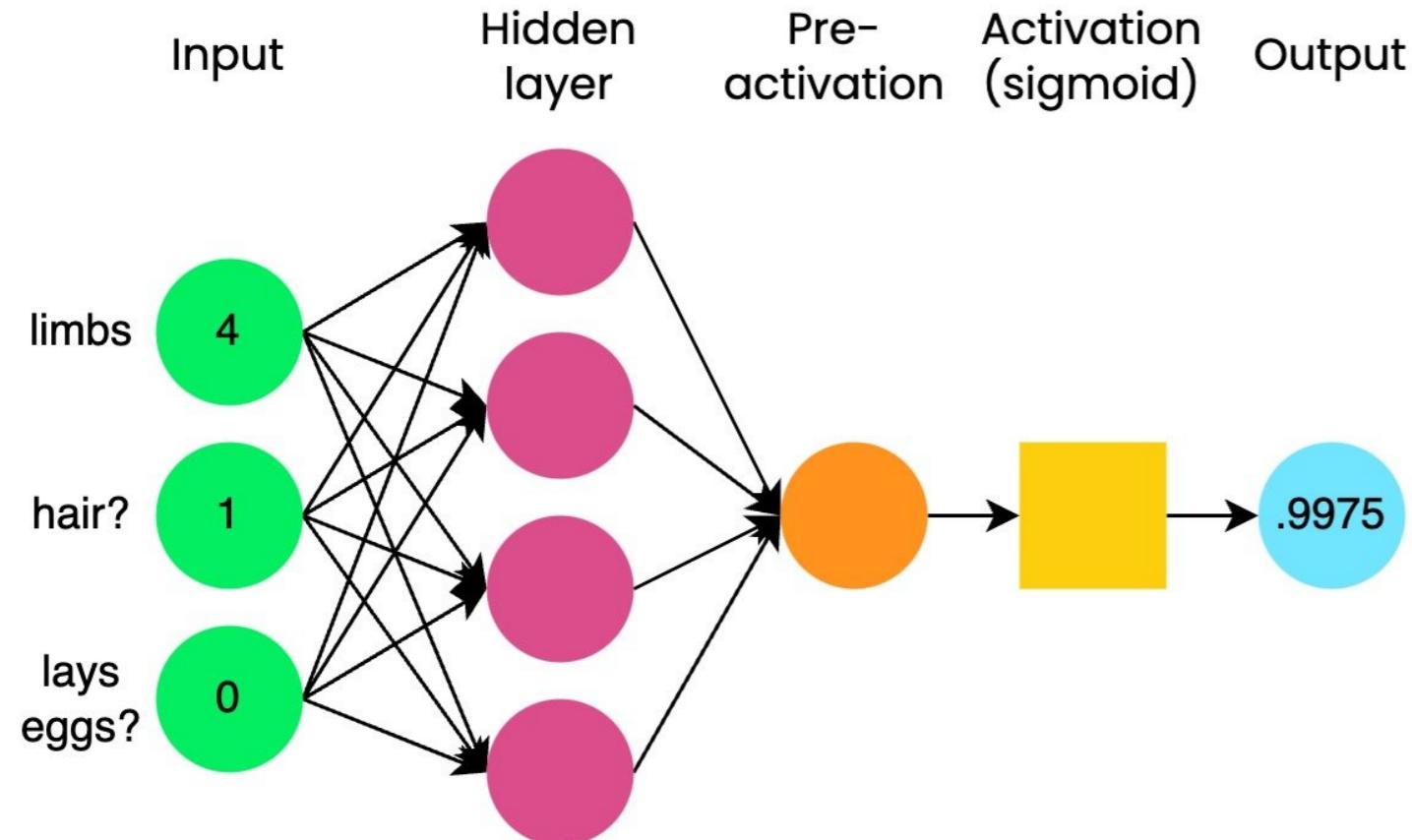
Meet the sigmoid function

Binary classification task:

- To predict whether animal is 1 (**mammal**) or 0 (**not mammal**),
- we take the pre-activation (6),
- pass it to the sigmoid,



Meet the sigmoid function



Binary classification task:

- To predict whether animal is 1 (**mammal**) or 0 (**not mammal**),
- we take the pre-activation (6),
- pass it to the sigmoid,
- and obtain a value between 0 and 1.

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**)

Meet the sigmoid function

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

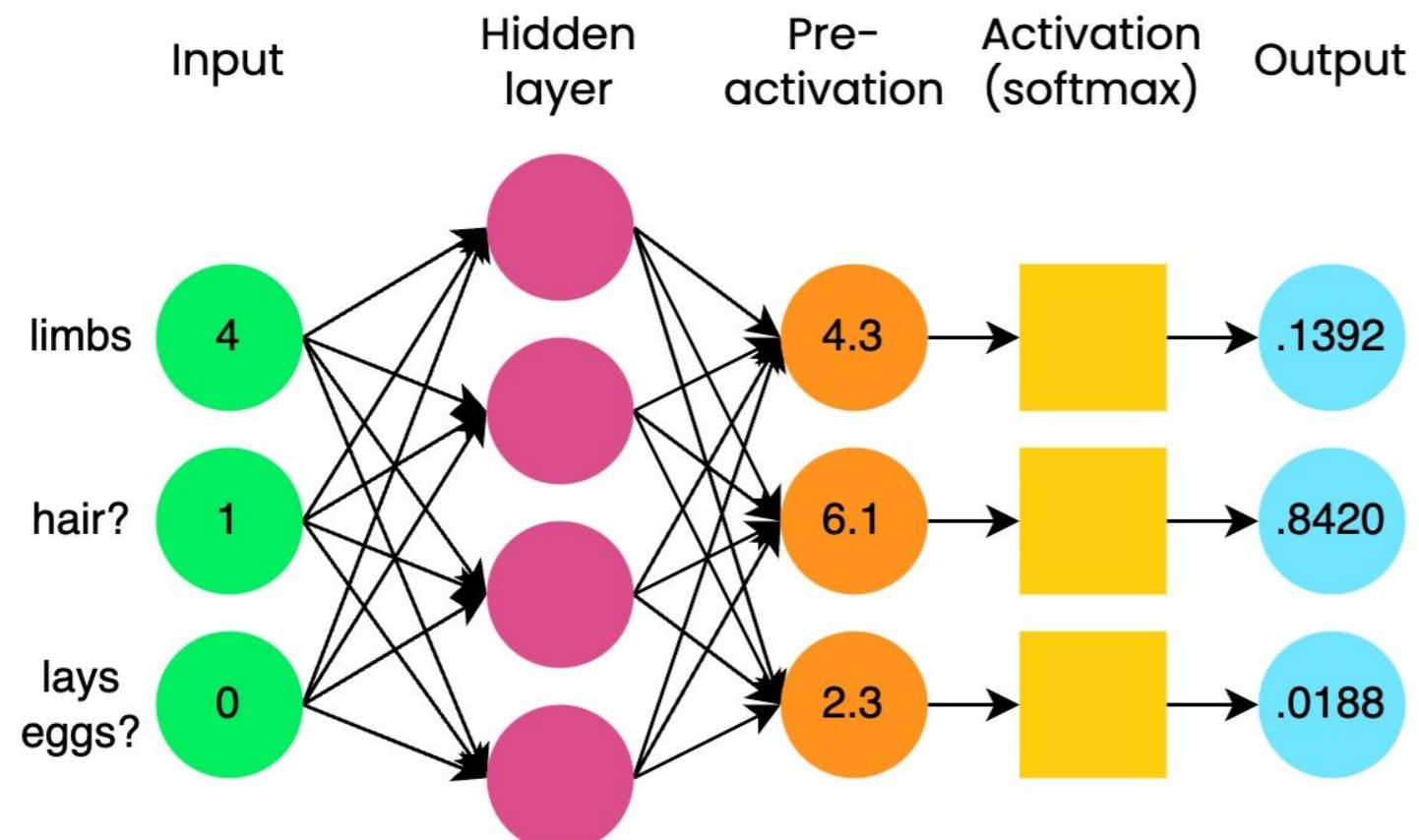
```
tensor([[0.9975]])
```

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.

Getting acquainted with 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()`