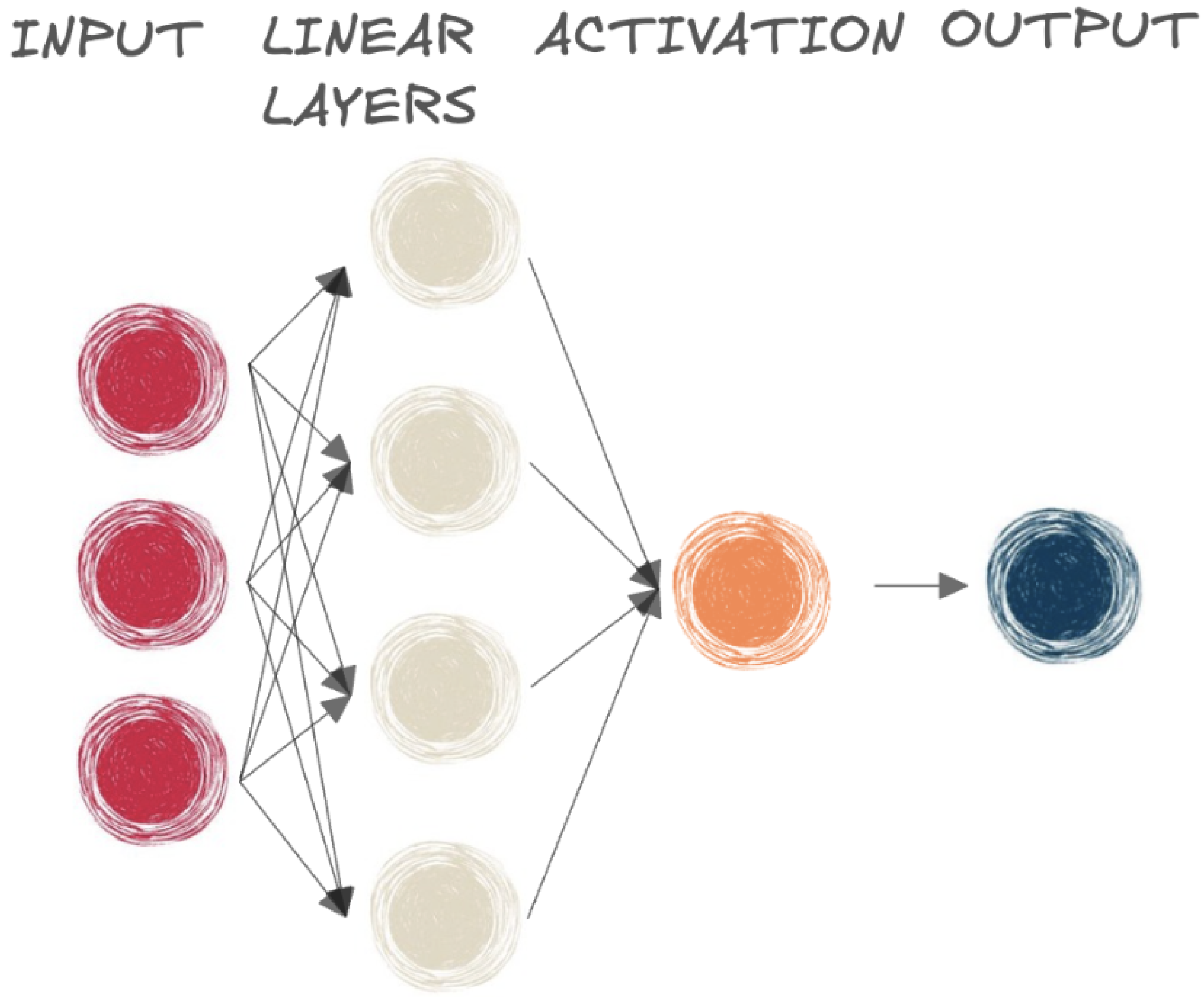
Discovering activation functions

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

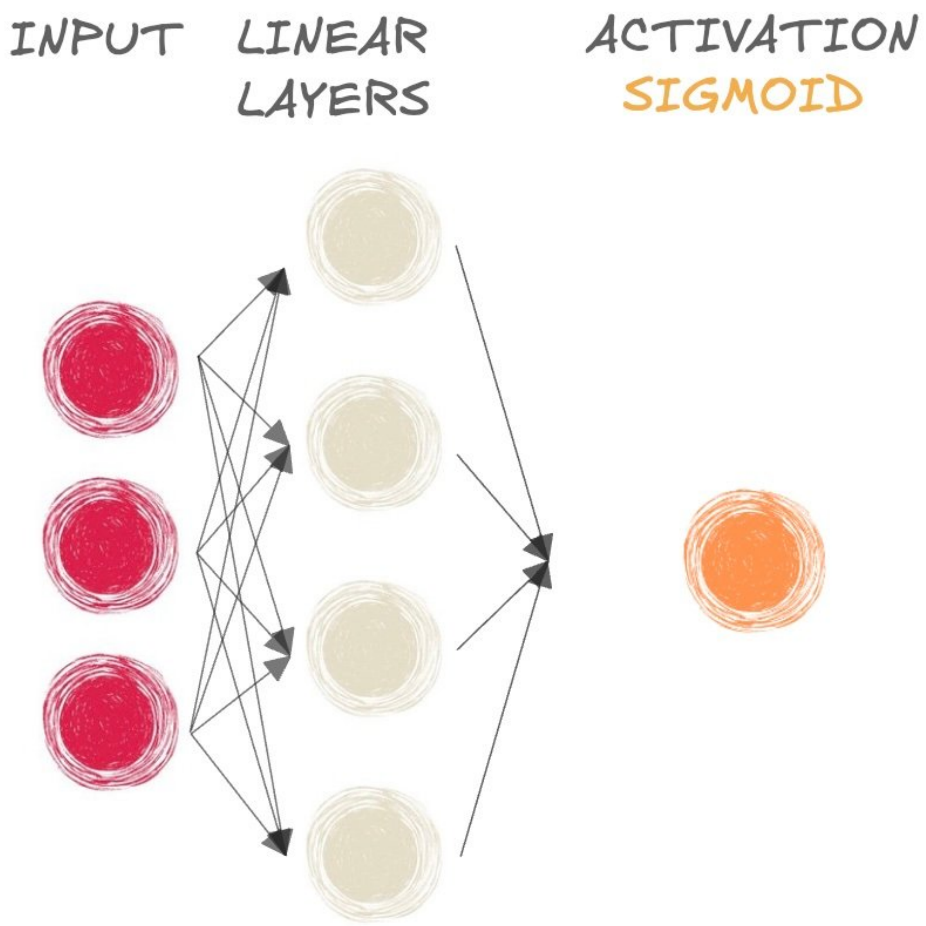
Jasmin Ludolf
Senior Data Science Content Developer, DataCamp

Activation functions

- **Activation functions** add **non-linearity** to the network
  - **Sigmoid** for binary classification
  - **Softmax** for multi-class classification
- A network can learn more **complex** relationships with non-linearity
- "Pre-activation" output passed to the activation function

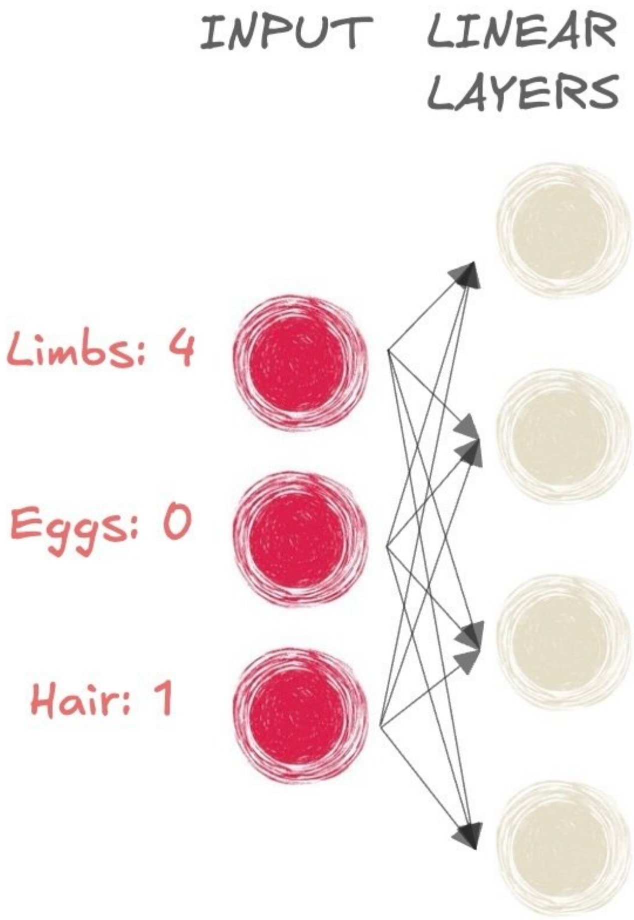


Meet the sigmoid function



- Mammal or not?

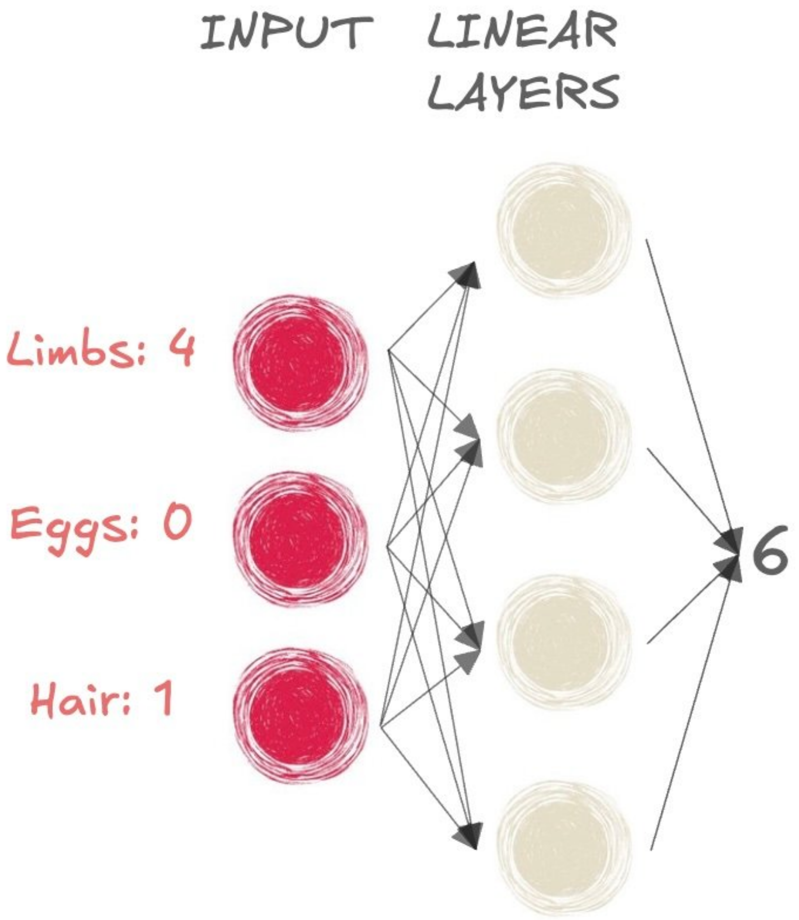
Meet the sigmoid function



- Mammal or not?

- Input:
  - Limbs: 4
  - Eggs: 0
  - Hair: 1

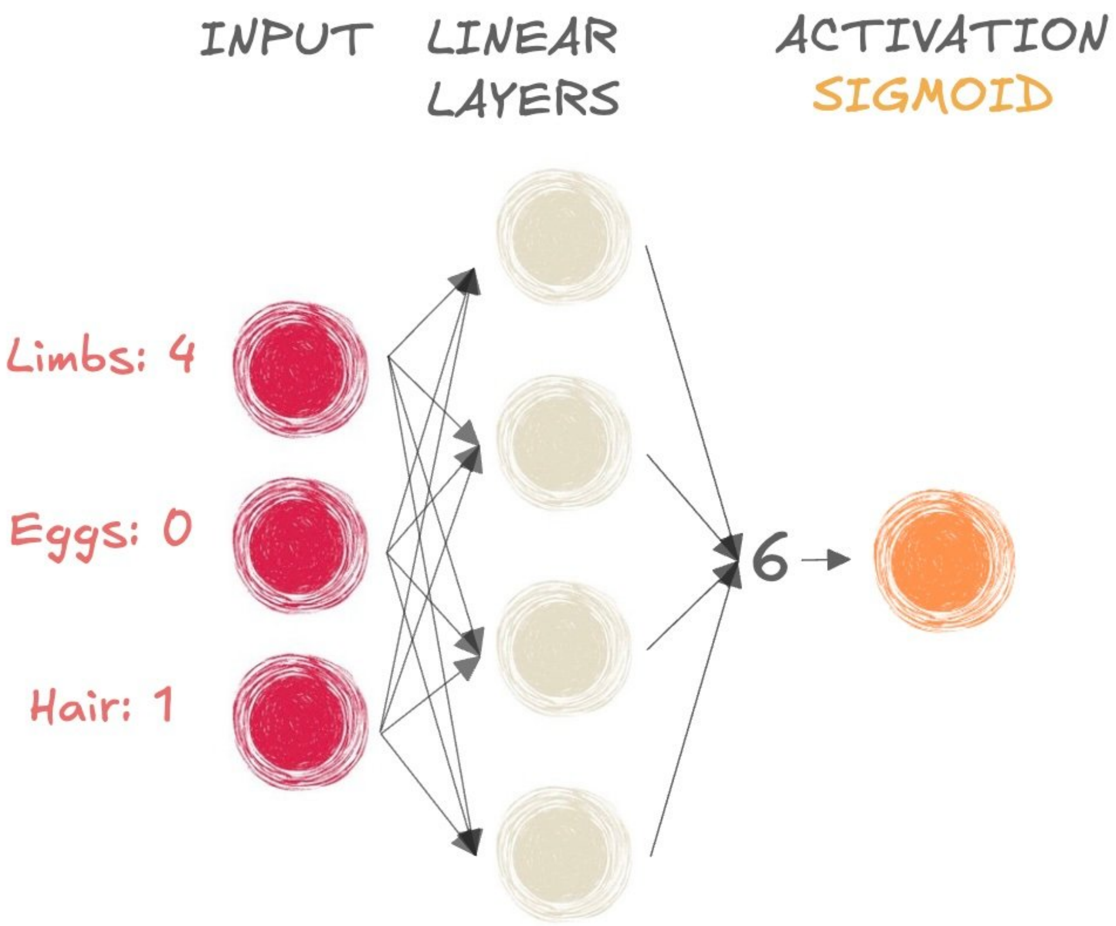
Meet the sigmoid function



- Mammal or not?

- Output to the linear layers is 6

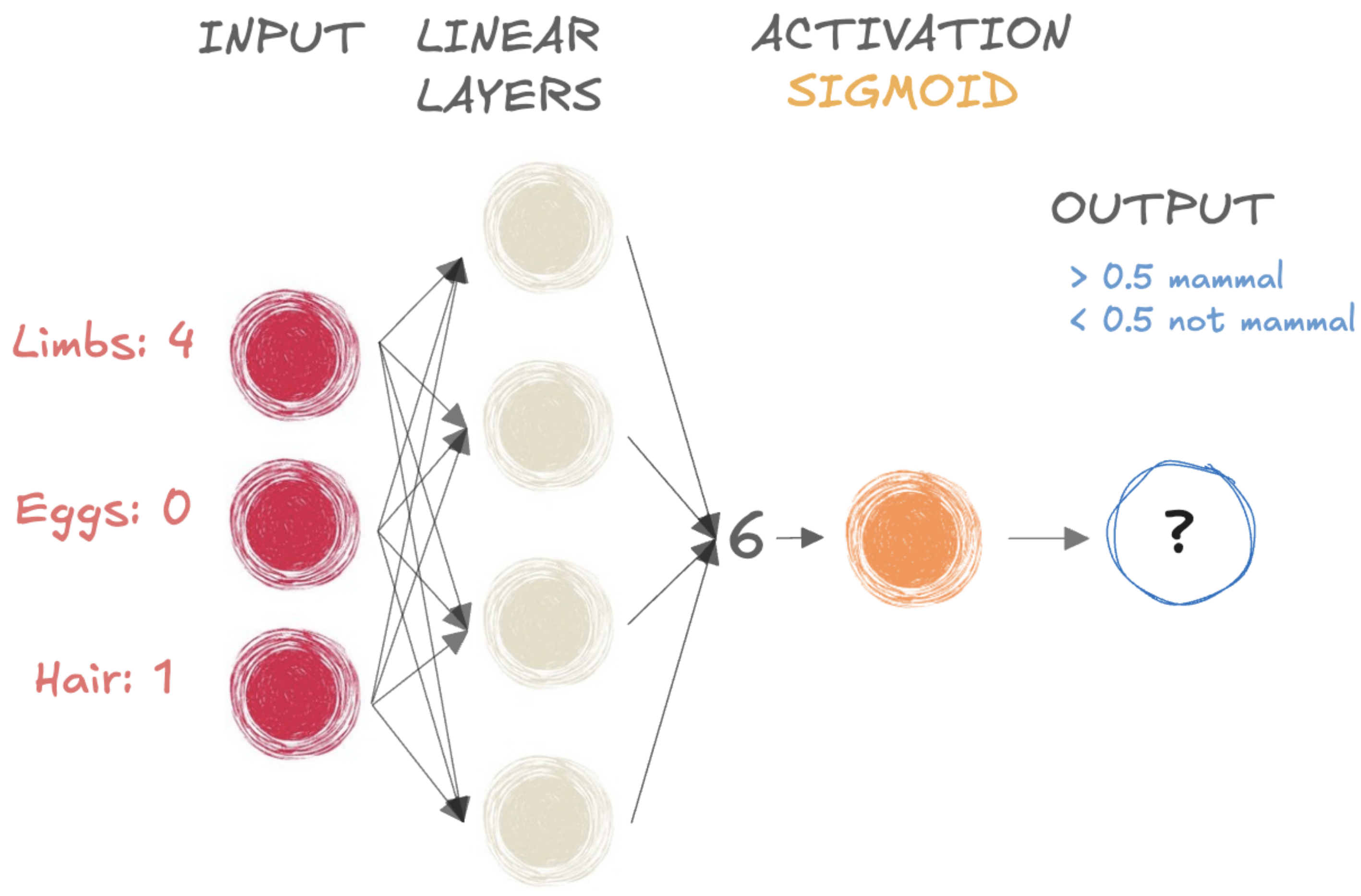
Meet the sigmoid function



- Mammal or not?

- We take the pre-activation output (6) and pass it to the sigmoid function

Meet the sigmoid function



- Mammal or not?

- We take the pre-activation output (6) and pass it to the sigmoid function
- Obtain a value between 0 and 1
- 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]])
sigmoid = nn.Sigmoid()
output = sigmoid(input_tensor)
print(output)
```

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

Activation 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
)
```

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

Getting acquainted with softmax

- Three classes:

Getting acquainted with softmax

- Three classes:

BIRD (0)

Getting acquainted with softmax

- Three classes:

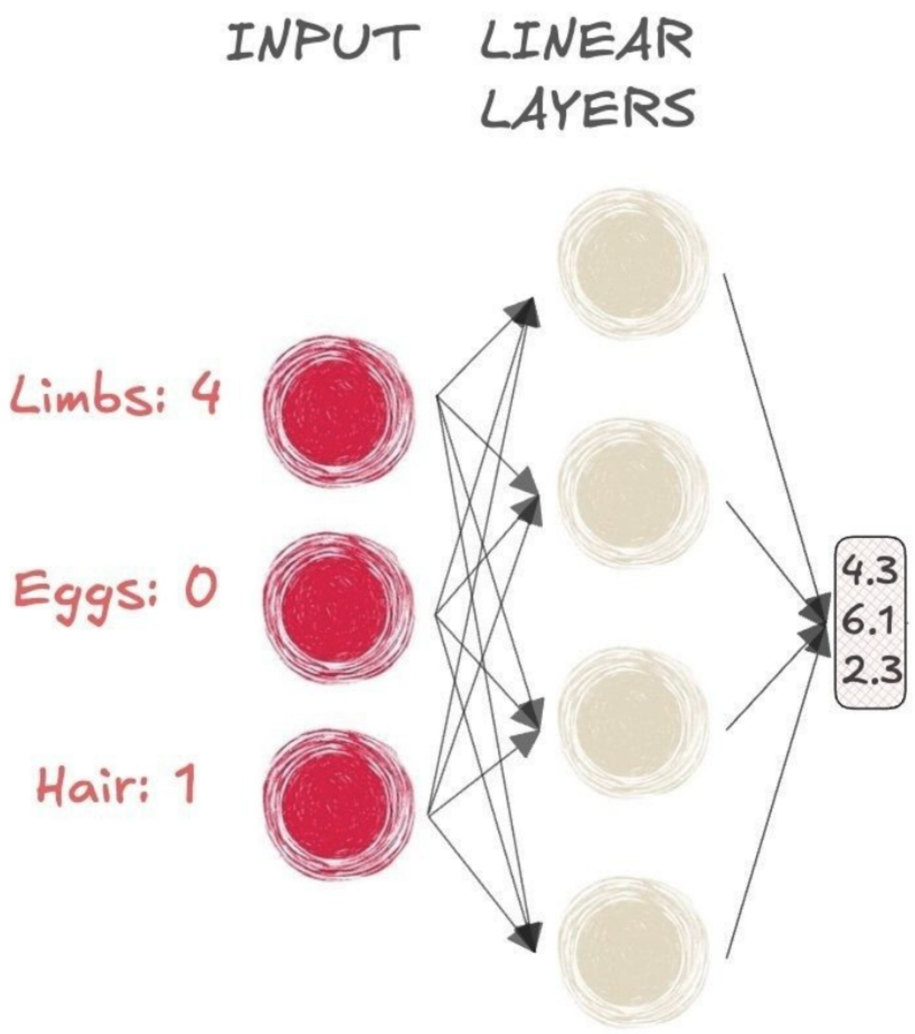
BIRD (0)

MAMMAL (1)

Getting acquainted with softmax

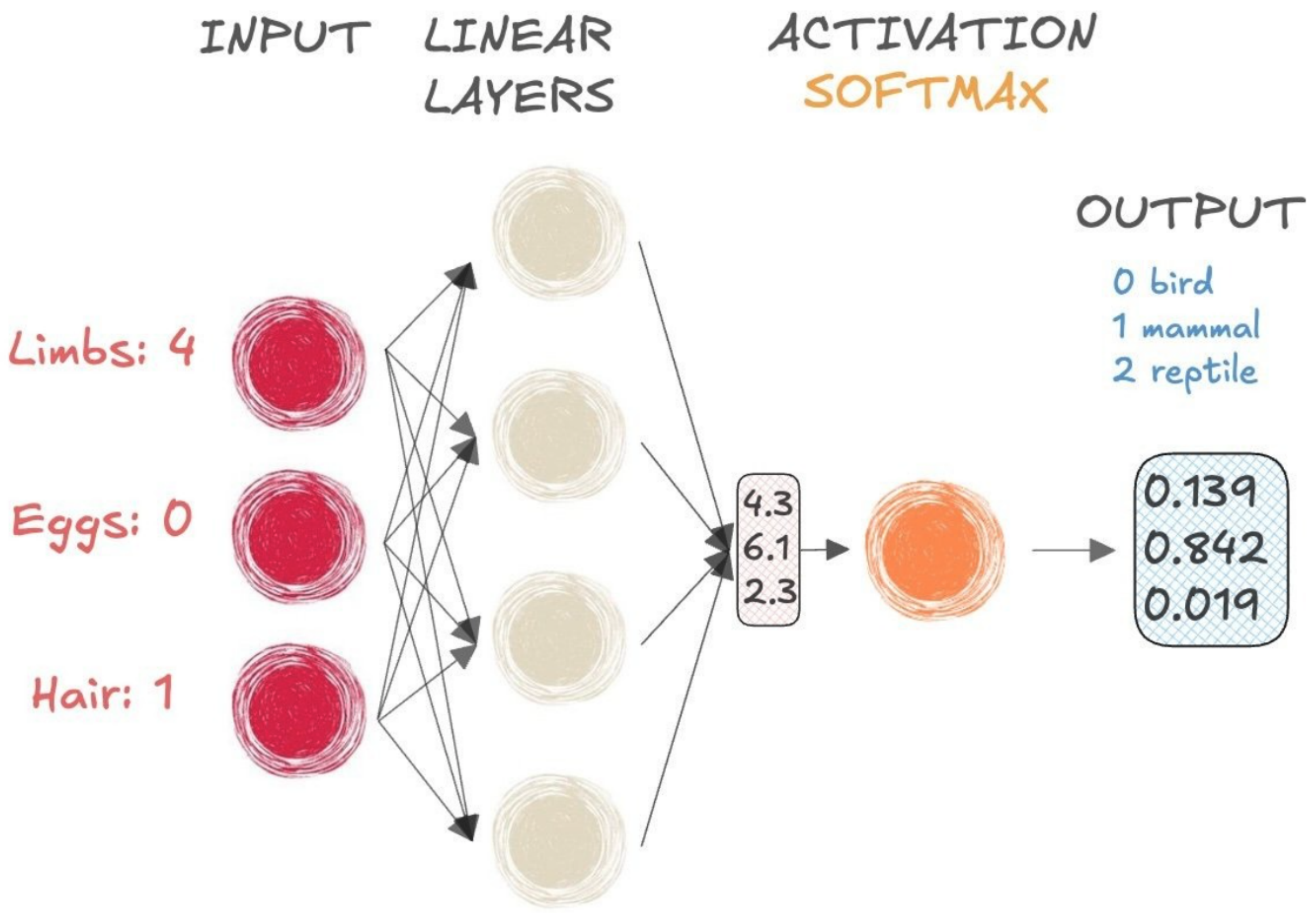
- Three classes:

Getting acquainted with softmax



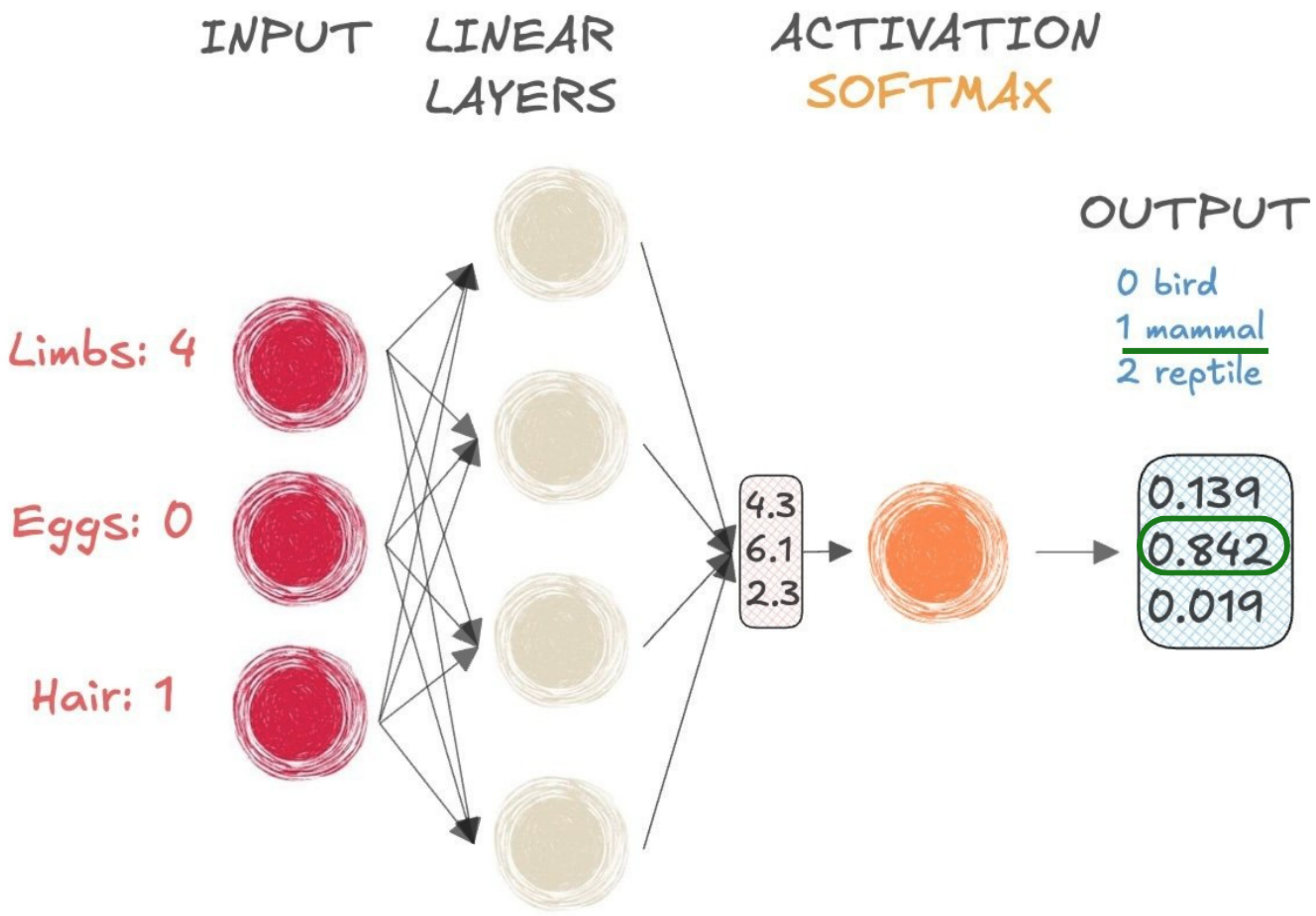
- Takes three-dimensional as input and outputs the same shape

Getting acquainted with softmax



- Takes three-dimensional as input and outputs the same shape
- 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



- Takes three-dimensional as input and outputs the same shape
- 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

```python
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()`

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Running a forward pass

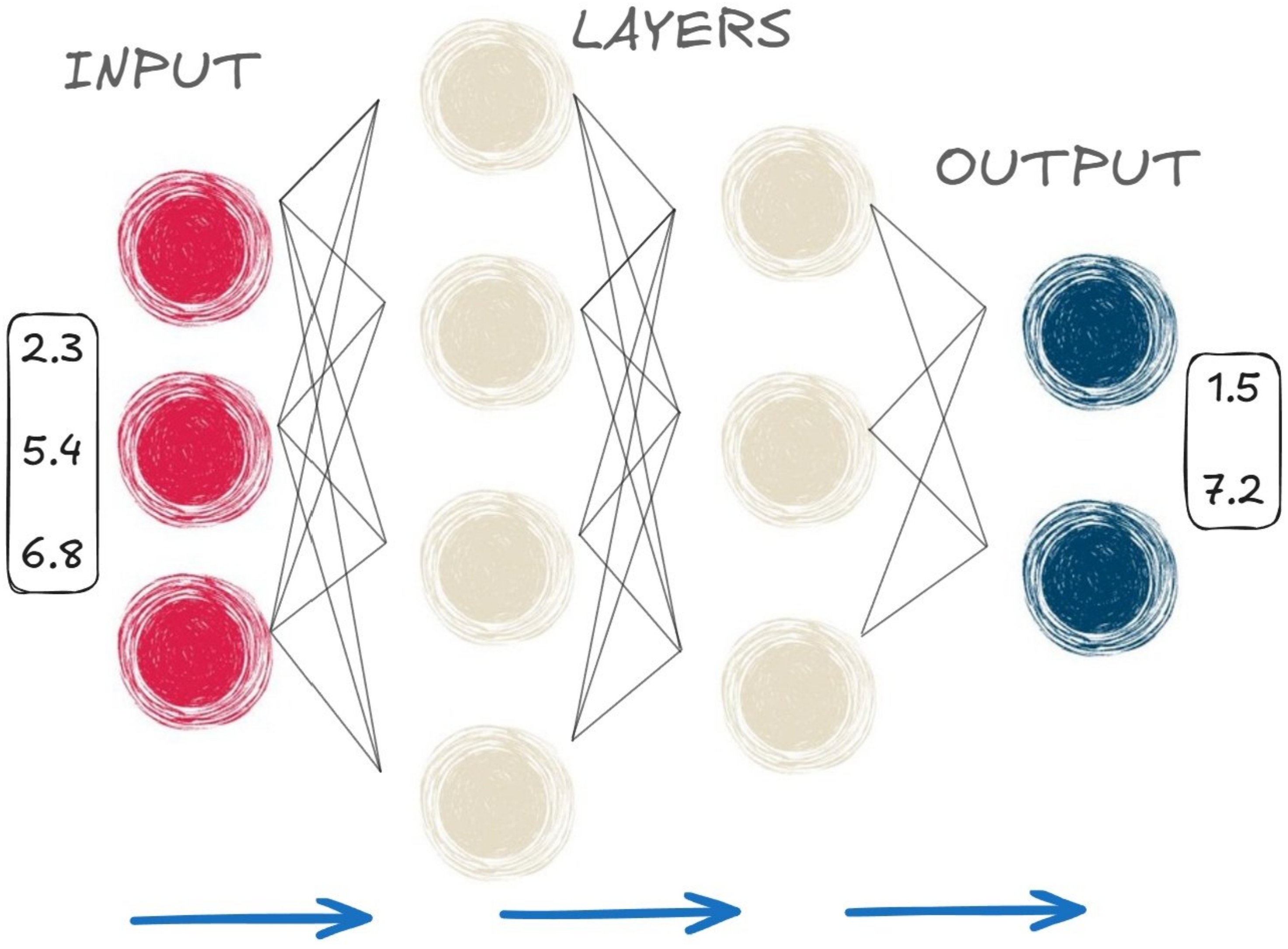
INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Jasmin Ludolf
Senior Data Science Content Developer, DataCamp

What is a forward pass?

- Input data **flows** through layers
- **Calculations** performed at each layer
- Final layer generates **outputs**

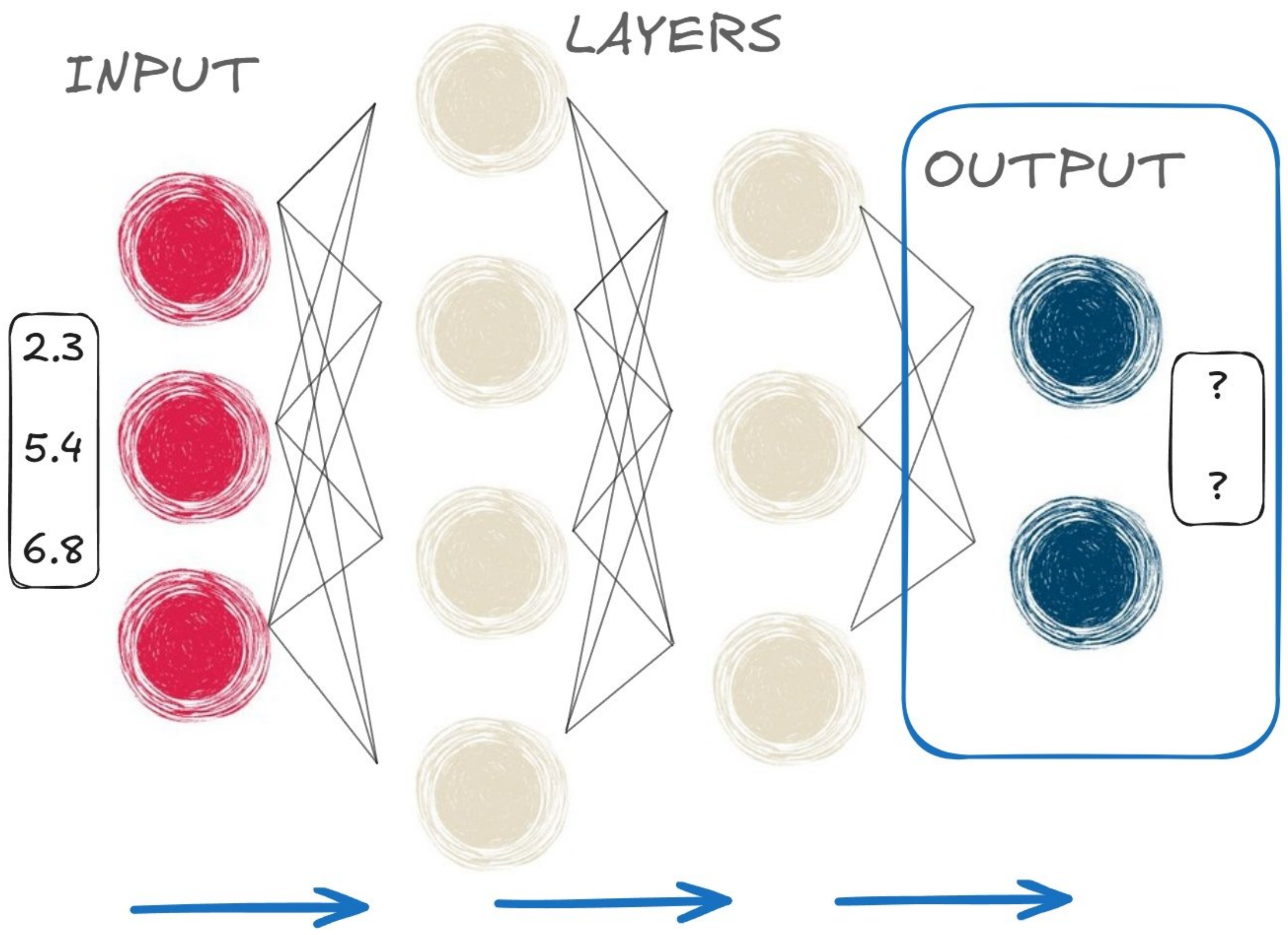
- Outputs produced based on **weights** and **biases**
- Used for **training** and making **predictions**



What is a forward pass?

Possible outputs:

- Binary classification
- Multi-class classification
- Regressions



Binary classification: forward pass

```
# Create input data of shape 5x6
input_data = torch.tensor(
        [[-0.4421,  1.5207,  2.0607, -0.3647,  0.4691,  0.0946],
        [-0.9155, -0.0475, -1.3645,  0.6336, -1.9520, -0.3398],
        [ 0.7406,  1.6763, -0.8511,  0.2432,  0.1123, -0.0633],
        [-1.6630, -0.0718, -0.1285,  0.5396, -0.0288, -0.8622],
        [-0.7413,  1.7920, -0.0883, -0.6685,  0.4745, -0.4245]])
```

5 animals

6 features

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

Binary classification: forward pass

```
# Pass input data through model
output = model(input_data)
print(output)
```

```
tensor([[0.5188], [0.3761], [0.5015], [0.3718], [0.4663]],
    grad_fn=<SigmoidBackward0>)
```

- Output: five probabilities between 0 and 1, one for each animal
- Classification (0.5 threshold):
  - Class = 1 (mammal) for values ≥ 0.5 (`0.5188`, `0.5015`)
  - Class = 0 (not mammal) for values < 0.5 (`0.3761`, `0.3718`, `0.4633`)

Multi-class classification: forward pass

- Class 1 - **mammal**, class 2 - **bird**, class 3 - **reptile**

```
n_classes = 3

# Create multi-class classification model
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, n_classes), # Second linear layer
  nn.Softmax(dim=-1) # Softmax activation
)

# Pass input data through model
output = model(input_data)
print(output.shape)
```

```
torch.Size([5, 3])
```

Multi-class classification: forward pass

```
print(output)
```

```
tensor([[0.4969, 0.3606, 0.1425],
        [0.5105, 0.3262, 0.1633],
        [0.3253, 0.3174, 0.3572],
        [0.5499, 0.3361, 0.1141],
        [0.4117, 0.3366, 0.2517]], grad_fn=<SoftmaxBackward0>)
```

5 animals

probabilities for each class

- Each row sums to one
- Predicted label = class with the highest probability
- Row 1 = class 1 (mammal), row 2 = class 1 (mammal), row 3 = class 3 (reptile)

Regression: forward pass

```
# Create regression model
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, 1) # Second linear layer
)

# Pass input data through model
output = model(input_data)

# Return output
print(output)
```

```
tensor([[0.3818],
        [0.0712],
        [0.3376],
        [0.0231],
        [0.0757]],
        grad_fn=<AddmmBackward0>)
```

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

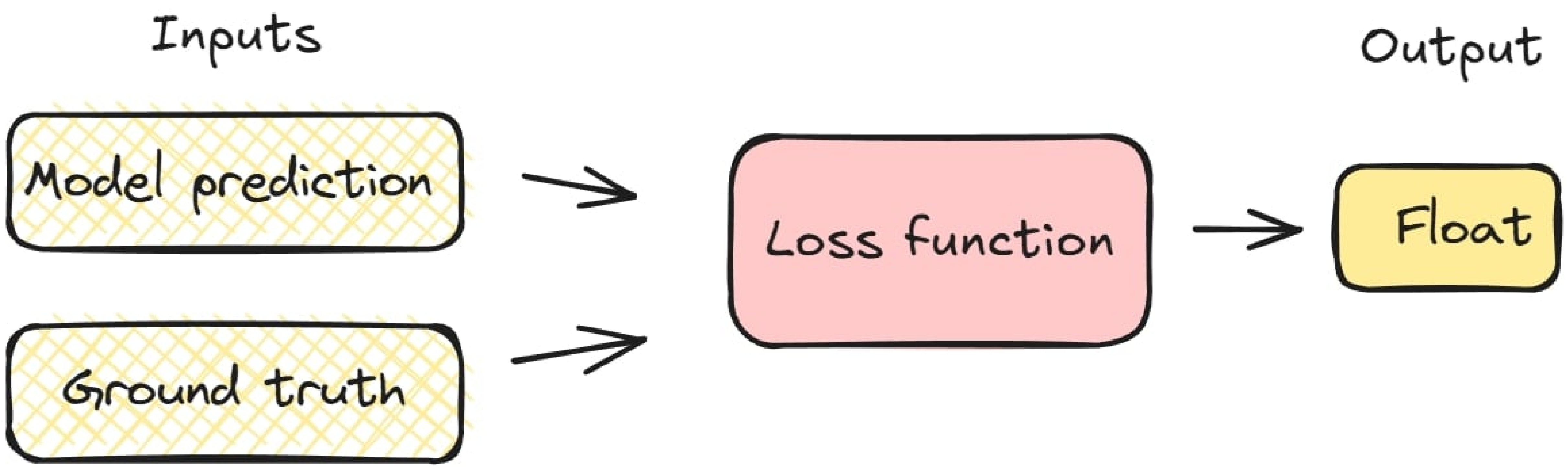
Using loss functions to assess model predictions

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Jasmin Ludolf
Senior Data Science Content Developer, DataCamp

Why do we need a loss function?

- Tells us how good our model is during training
- Takes a model **prediction** $\hat{y}$ and **ground truth** y
- Outputs a float



Why do we need a loss function?

- Class 0 - **mammal**, class 1 - **bird**, class 2 - **reptile**

Hair	Feathers	Eggs	Milk	Fins	Legs	Tail	Domestic	Catsize	Class
1	0	0	1	0	4	0	0	1	0

- Predicted class = 0 -> **correct** = low loss
- Predicted class = 1 -> **wrong** = high loss
- Predicted class = 2 -> **wrong** = high loss

- Our goal is to **minimize loss**

One-hot encoding concepts

- $loss = F(y, \hat{y})$
- y is a single **integer** (class label)
  - e.g. $y = 0$ when y is a mammal
- $\hat{y}$ is a **tensor** (prediction before softmax)
  - If N is the number of classes, e.g. N = 3
  - $\hat{y}$ is a tensor with N dimensions,
    - e.g. $\hat{y}$ = `[-5.2, 4.6, 0.8]`

One-hot encoding concepts

- Convert an **integer** y to a **tensor** of zeros and ones

```
ground truth y = 0
number of classes N = 3
```

class	**0**	**1**	**2**
one-hot encoding	1	0	0

Transforming labels with one-hot encoding

```
import torch.nn.functional as F

print(F.one_hot(torch.tensor(0), num_classes = 3))
```

```
tensor([1, 0, 0])
```

```
print(F.one_hot(torch.tensor(1), num_classes = 3))
```

```
tensor([0, 1, 0])
```

```
print(F.one_hot(torch.tensor(2), num_classes = 3))
```

```
tensor([0, 0, 1])
```

Cross entropy loss in PyTorch

```
from torch.nn import CrossEntropyLoss

scores = torch.tensor([-5.2, 4.6, 0.8])
one_hot_target = torch.tensor([1, 0, 0])

criterion = CrossEntropyLoss()
print(criterion(scores.double(), one_hot_target.double()))
```

```
tensor(9.8222, dtype=torch.float64)
```

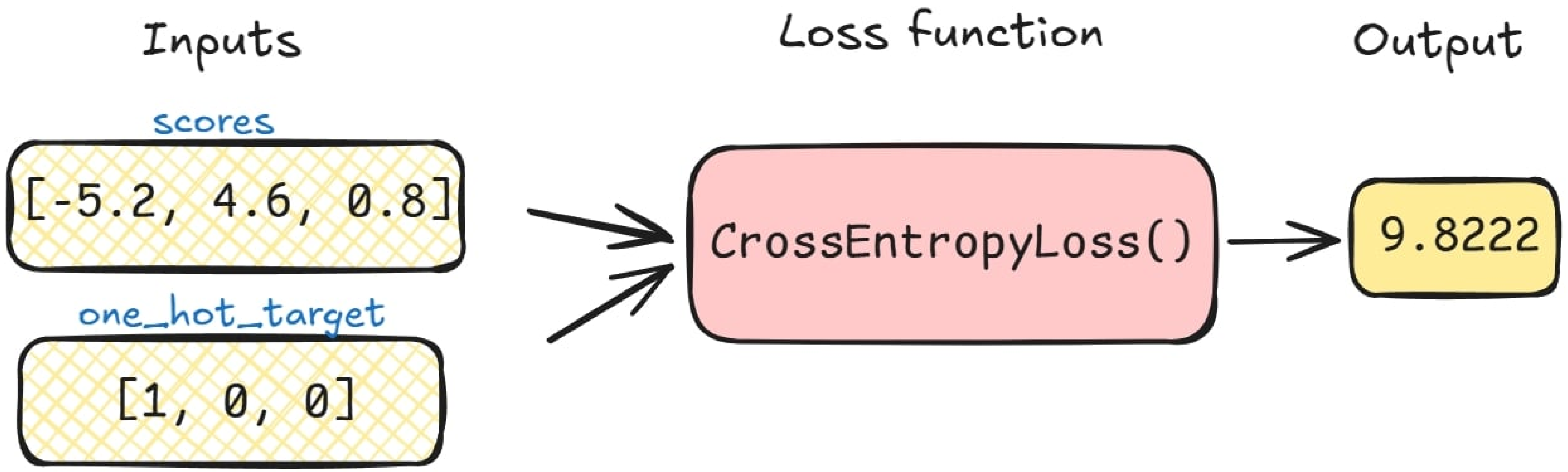
Bringing it all together

Loss function takes:

- **scores** - model predictions **before** the final softmax function
- **one_hot_target** - one hot encoded ground truth label

Loss function outputs:

- **loss** - a single **float**



Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

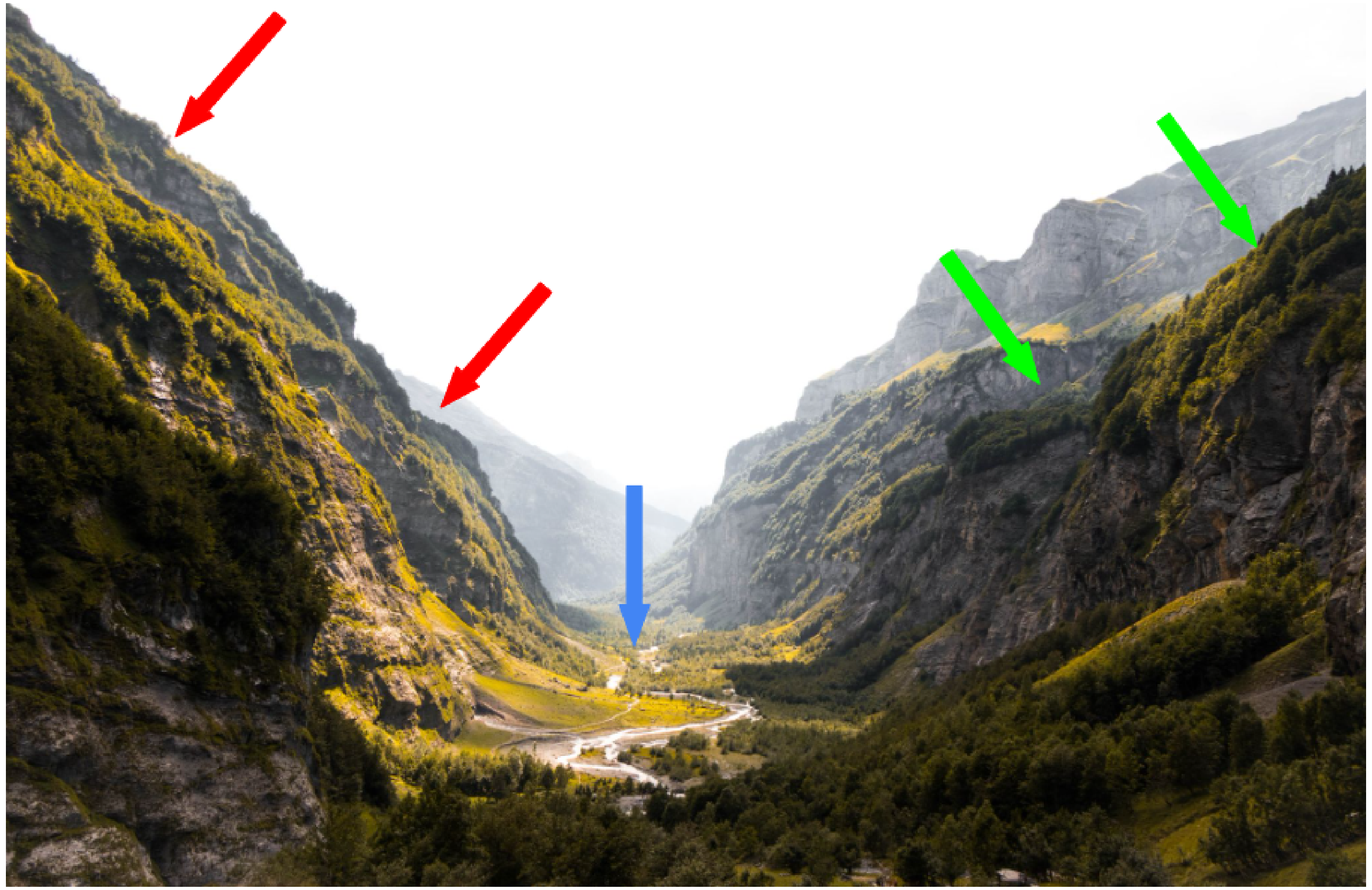
Using derivatives to update model parameters

INTRODUCTION TO DEEP LEARNING WITH PYTORCH

Jasmin Ludolf
Senior Data Science Content Developer, DataCamp

An analogy for derivatives

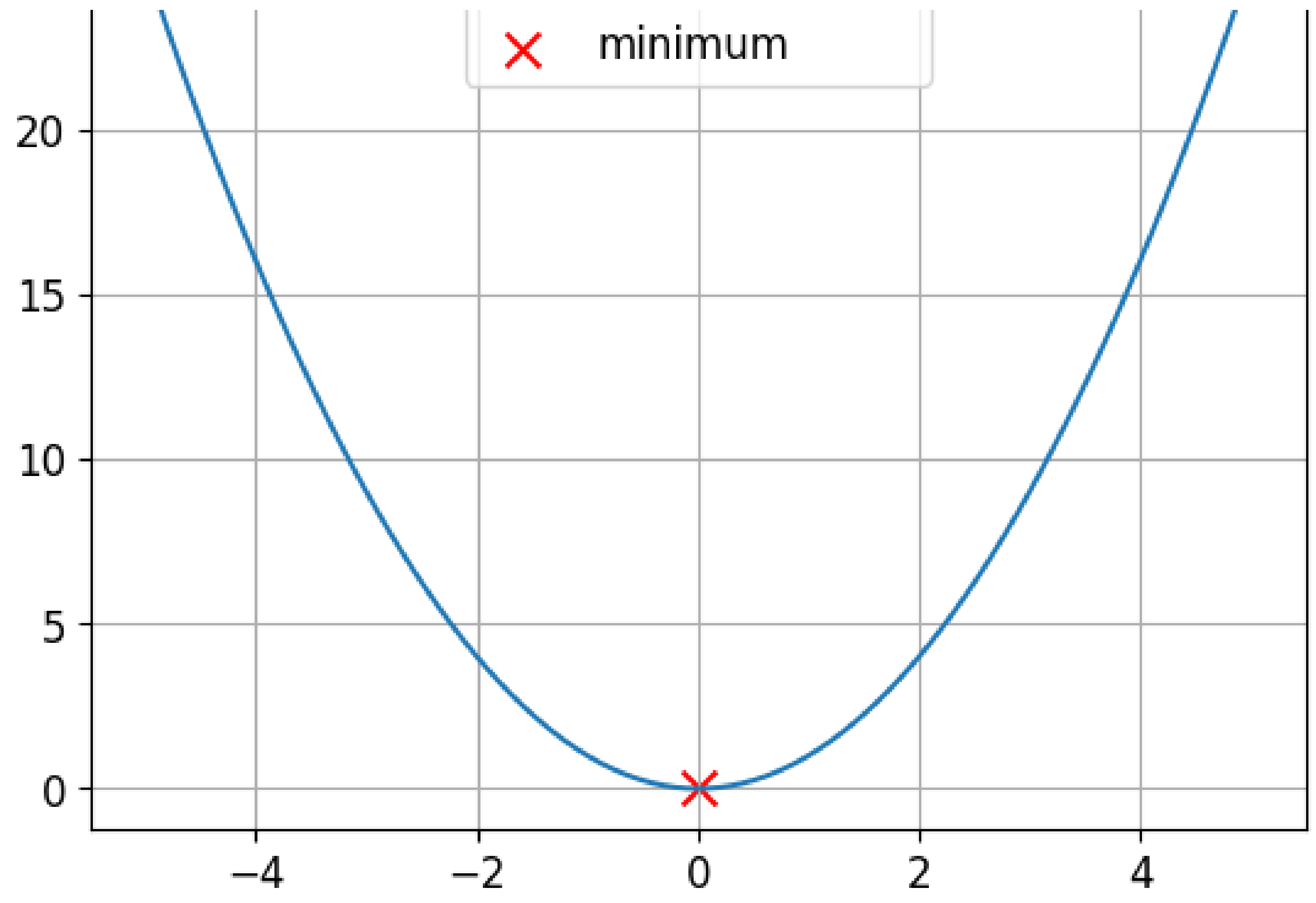
Derivative represents the **slope of the curve**



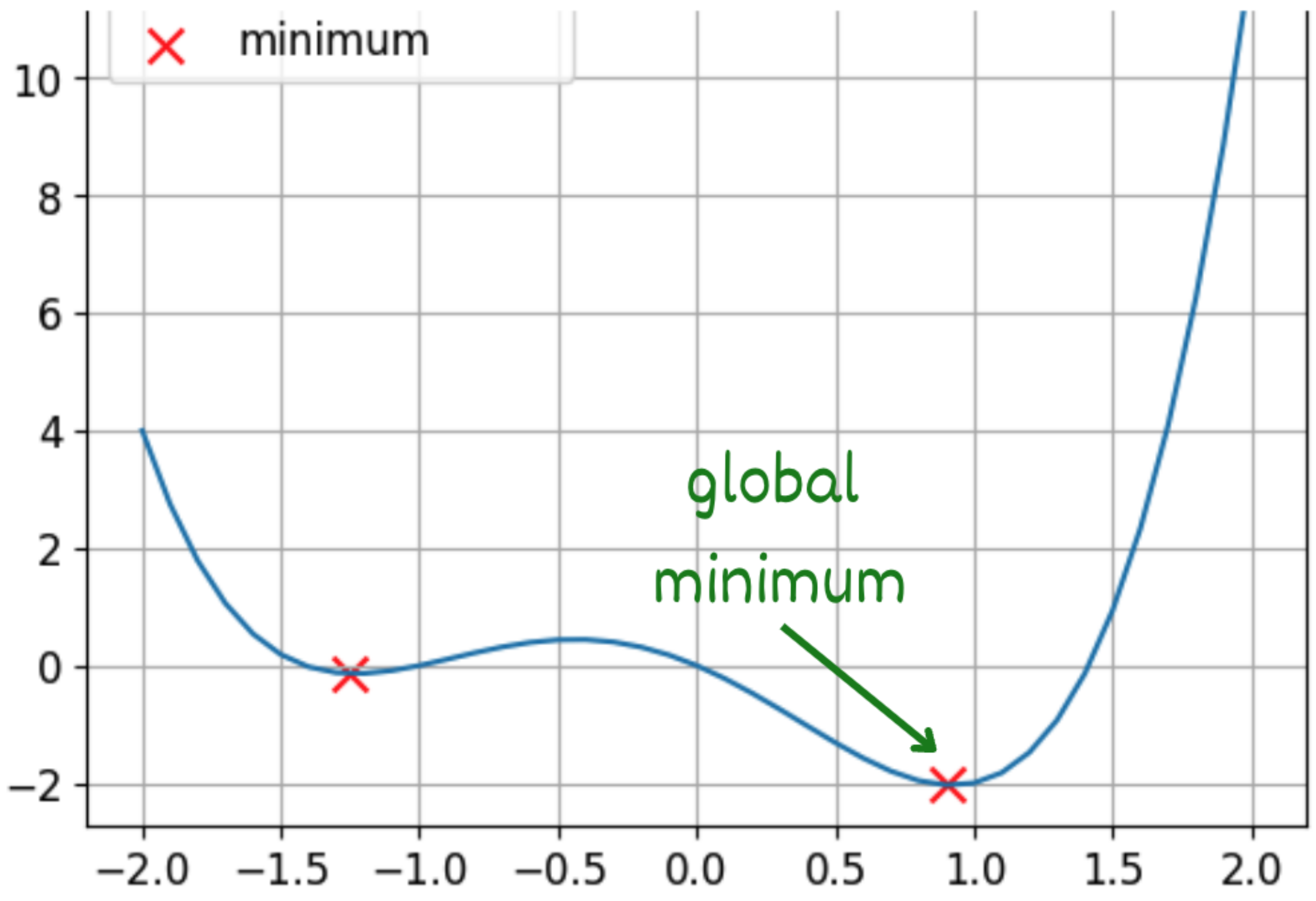
- **Steep slopes** (red arrows):
  - Large steps, **derivative is high**
- **Gentler slopes** (green arrows):
  - Small steps, **derivative is low**
- **Valley floor** (blue arrow):
  - Flat, **derivative is zero**

Convex and non-convex functions

This is a **convex function**

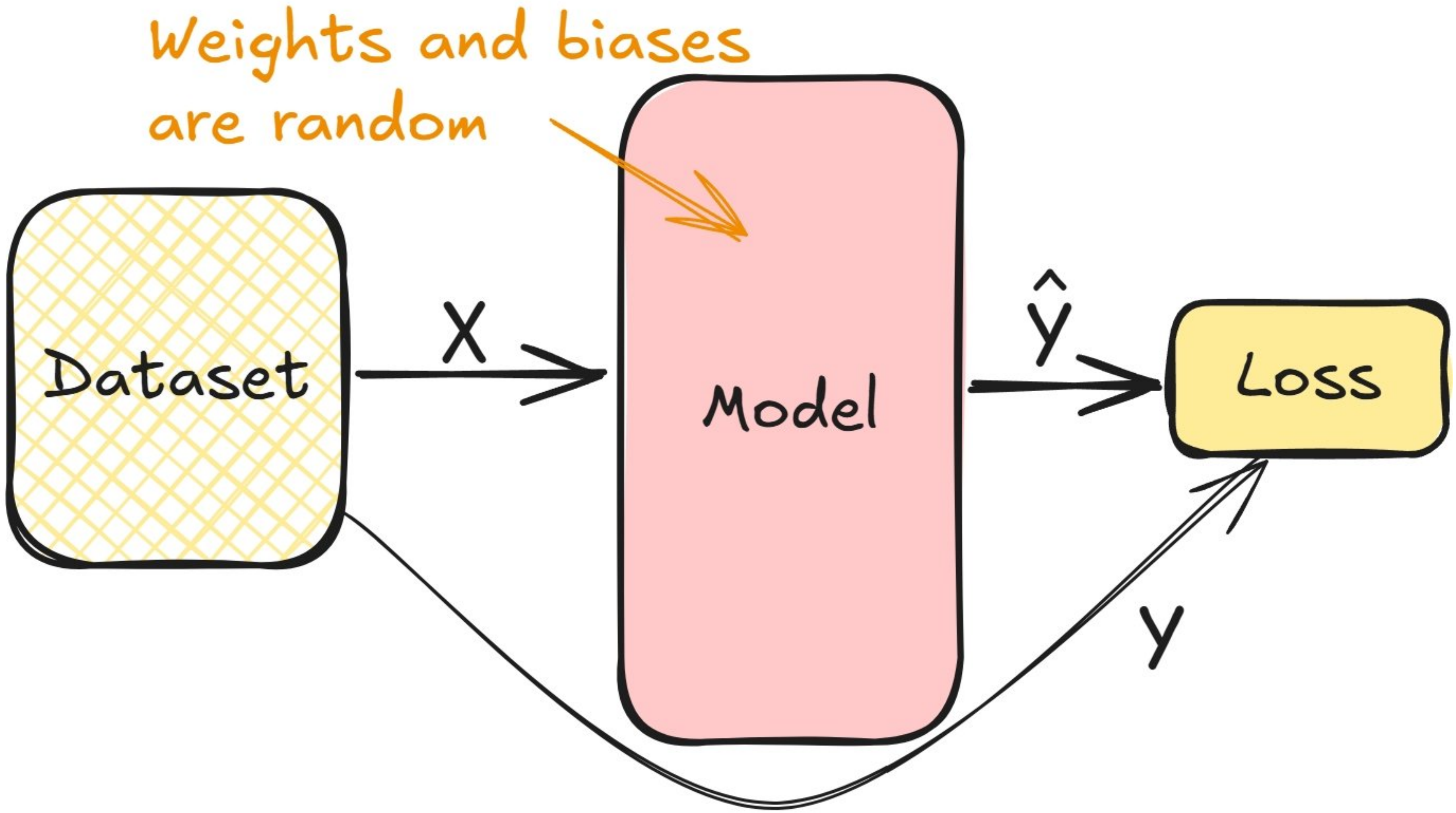


This is a **non-convex function**



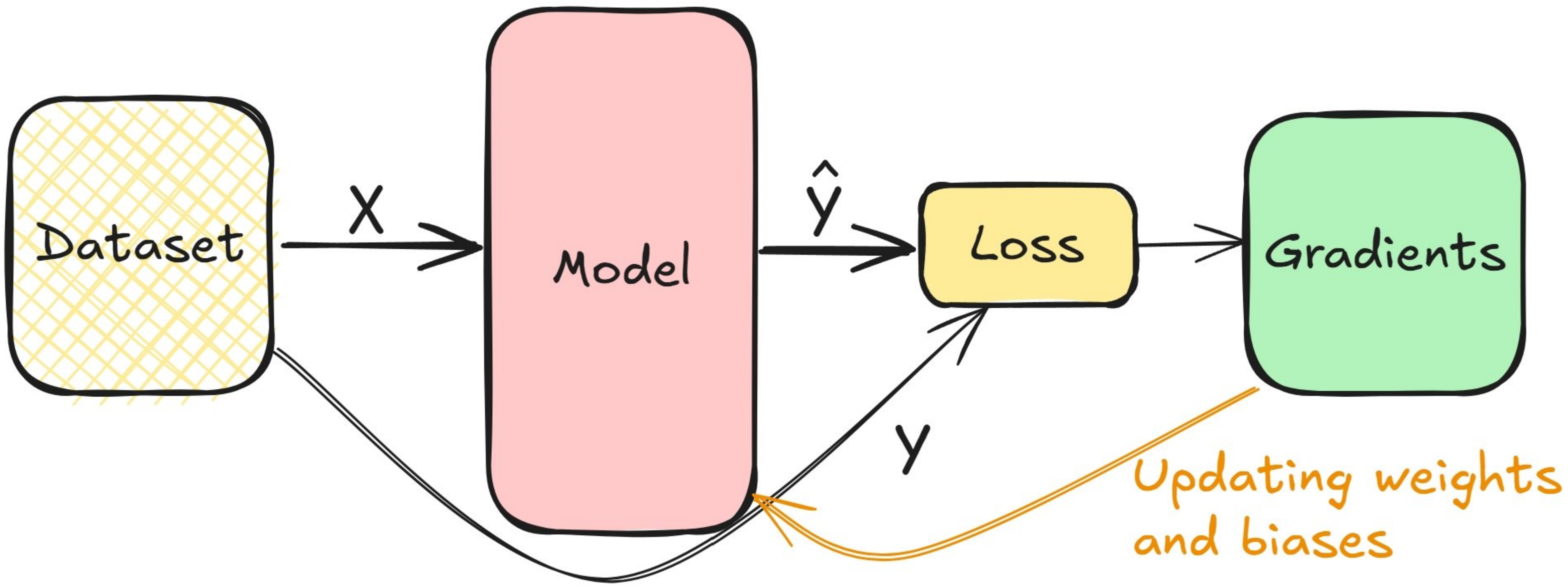
Connecting derivatives and model training

- Compute the loss in the forward pass during training



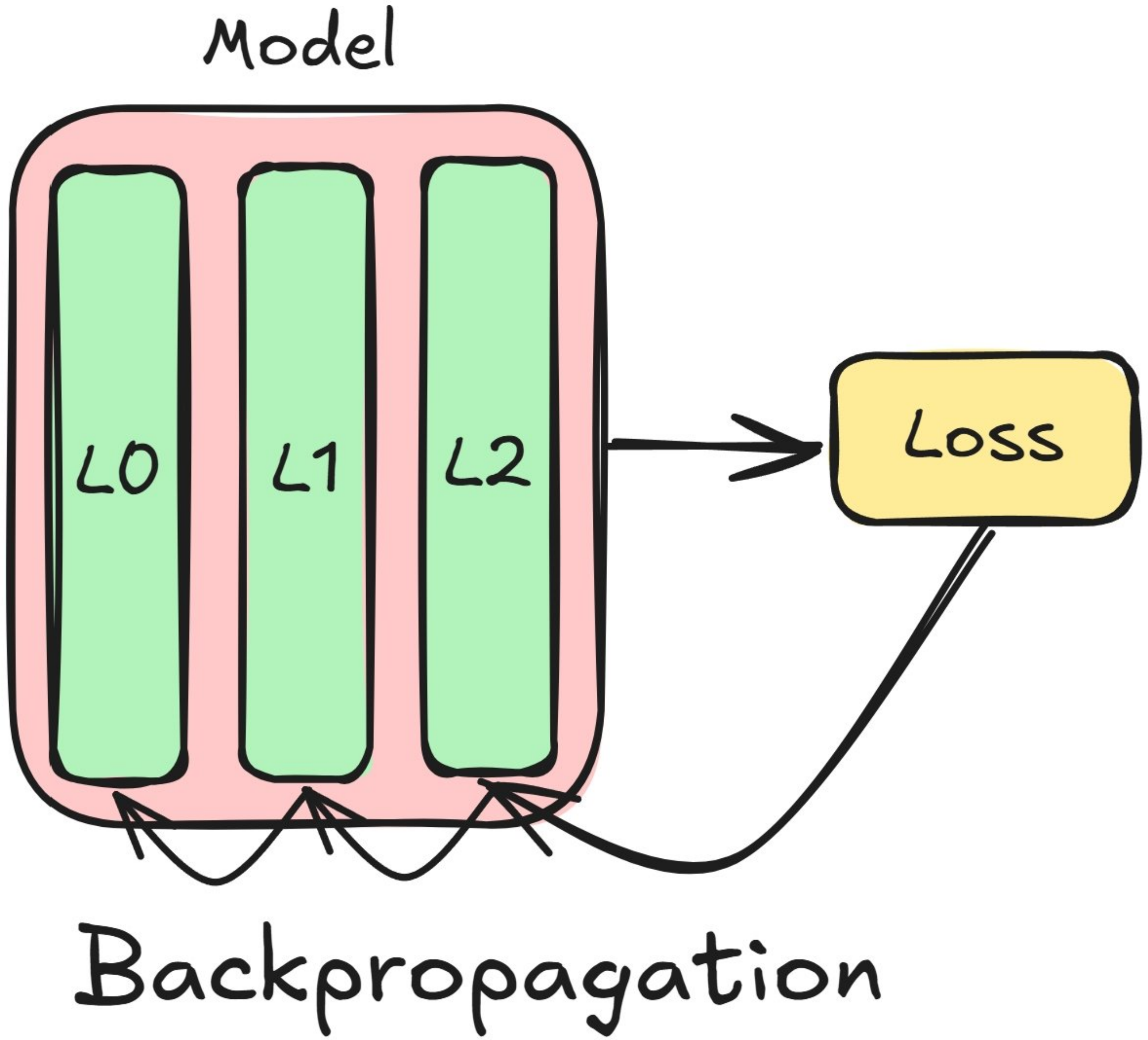
Connecting derivatives and model training

- Gradients help minimize **loss**, tune layer **weights** and **biases**
- Repeat until the layers are **tuned**



Backpropagation concepts

- Consider a network made of three layers:
  - Begin with loss gradients for $L2$
  - Use $L2$ to compute $L1$ gradients
  - Repeat for all layers ($L1$, $L0$)



Backpropagation in PyTorch

```
# Run a forward pass
model = nn.Sequential(nn.Linear(16, 8),
                      nn.Linear(8, 4),
                      nn.Linear(4, 2))
prediction = model(sample)

# Calculate the loss and gradients
criterion = CrossEntropyLoss()
loss = criterion(prediction, target)
loss.backward()
```

```
# Access each layer's gradients
model[0].weight.grad
model[0].bias.grad
model[1].weight.grad
model[1].bias.grad
model[2].weight.grad
model[2].bias.grad
```

Updating model parameters manually

```
# Learning rate is typically small
lr = 0.001

# Update the weights
weight = model[0].weight
weight_grad = model[0].weight.grad

weight = weight - lr * weight_grad

# Update the biases
bias = model[0].bias
bias_grad = model[0].bias.grad
bias = bias - lr * bias_grad
```

- Access each layer gradient
- Multiply by the learning rate
- Subtract this product from the weight

Gradient descent

- For non-convex functions, we will use **gradient descent**
- PyTorch simplifies this with **optimizers**
  - Stochastic gradient descent (SGD)

```
import torch.optim as optim

# Create the optimizer
optimizer = optim.SGD(model.parameters(), lr=0.001)

# Perform parameter updates
optimizer.step()
```

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH PYTORCH