# Fundamentals of Neural Networks

Cameron Kyle-Davidson

University of York

# Session Structure

This isn't a deep dive into theory!

- Goal: Equip you with enough practical knowledge to build neural networks with PyTorch.
- With a 'good enough' understanding of what's going on behind the scenes.

# This Session Structure

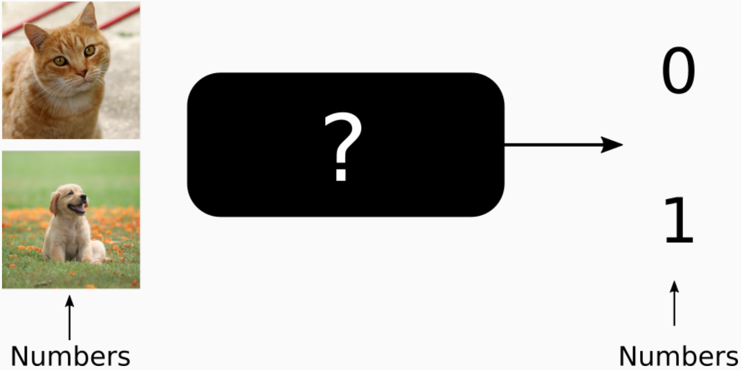
- Neural Network Fundamentals
- Practical Session (Getting started with Pytorch)

# What actually is a neural network?

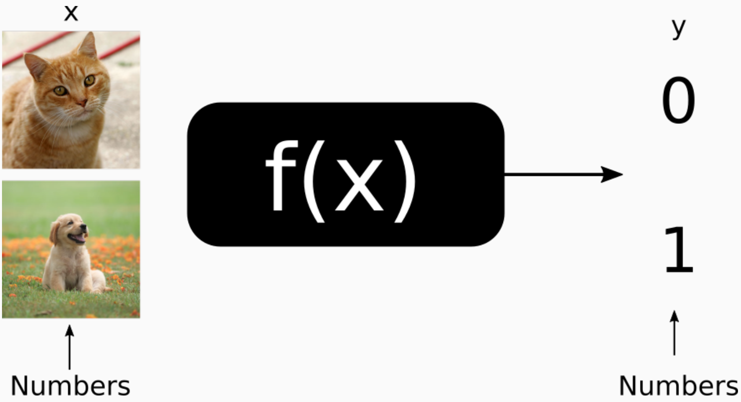
**Problem Statement**

Categorise dogs and cats?

# What actually is a neural network?

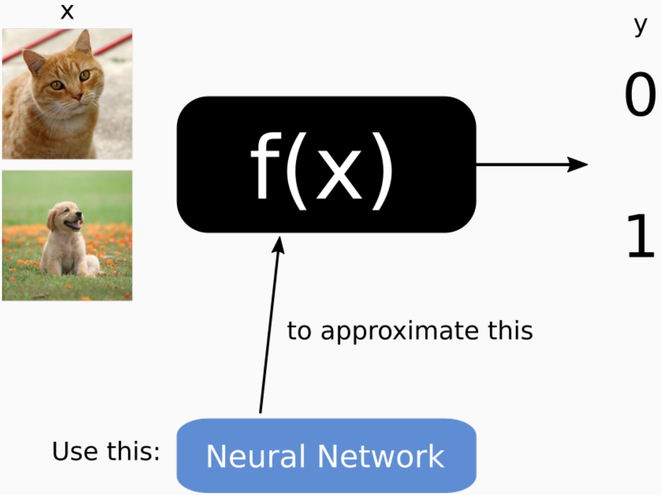


# What actually is a neural network?



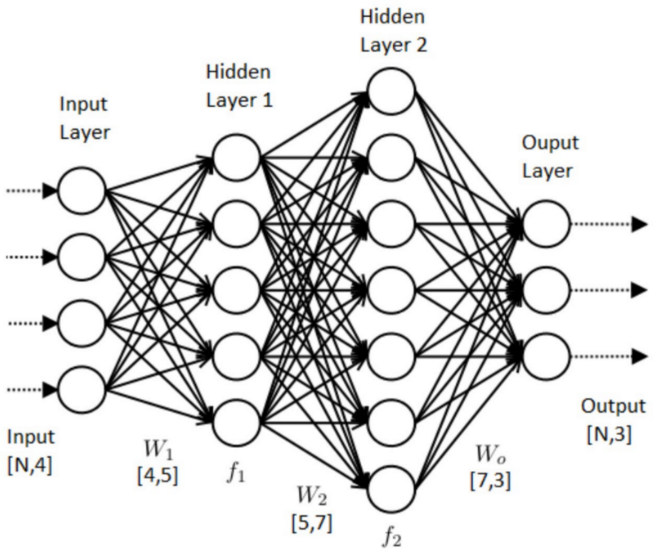
# What actually is a neural network?

- Good luck doing that by hand.

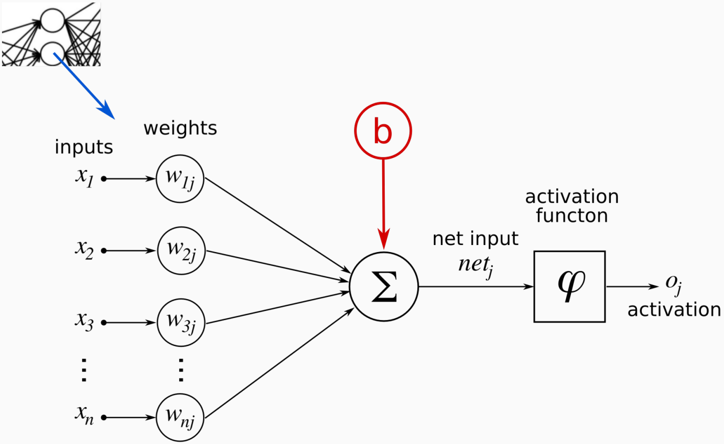



- Luckily, neural networks are **universal function approximators.**

# What actually is a neural network?

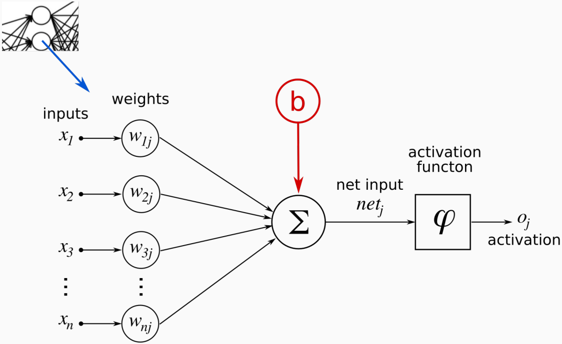


# The Artificial Neuron




$$y = \varphi(\sum_{k=0}^{m} w_{kj} x_k + b_j)$$

# The Artificial Neuron

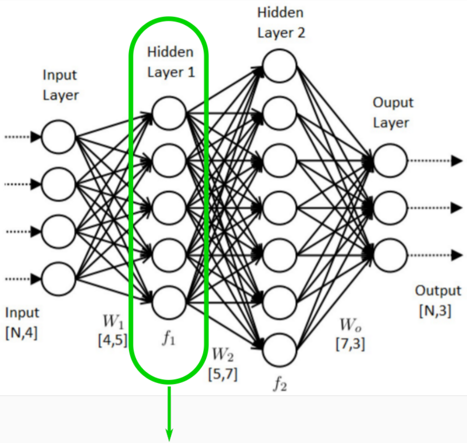



- **Weights**: A transformation applied to the input to the neuron. Most of what the neural network 'learns' is encoded in the weights.
- **Bias**: Required for the network to describe a variety of functions.

# Practical Differences

## Don't Worry

Modern ML Libraries tend to abstract away the weights & biases.




```
hidden = nn.Linear(in_features=4, out_features=5)
```

# Practical Differences

But they are there.

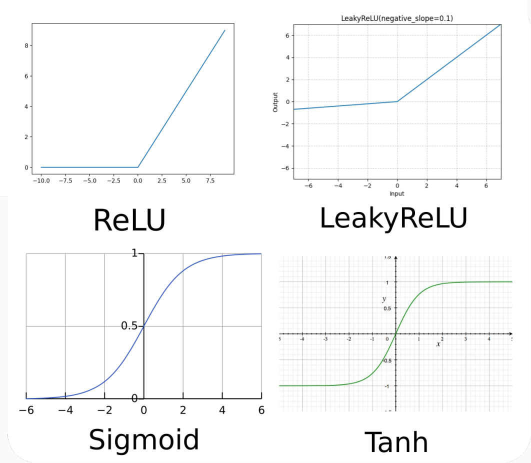
Reader Mode

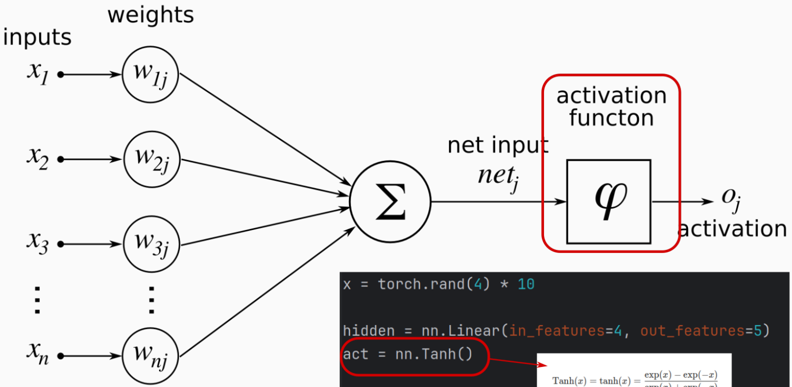
```
class Linear(Module):
    def __init__(
) -> None:
    factory_kwargs = {"device": device, "dtype": dtype}
    super().__init__()
    self.in_features = in_features
    self.out_features = out_features
    self.weight = Parameter(
        torch.empty((out_features, in_features), **factory_kwargs)
    )
    if bias:
        self.bias = Parameter(torch.empty(out_features, **factory_kwargs))
    else:
        self.register_parameter( name: "bias",  param: None)
```

* (You're unlikely to see this again, unless you decide to define your own NN layers)

# Activation Functions

- Most real-world 'interesting' functions aren't **linear**
- An activation function introduces non-linearity by operating on the output of each neuron.






```python
x = torch.rand(4) * 10

hidden = nn.Linear(in_features=4, out_features=5)
act = nn.Tanh()

pre_act = hidden(x)
# [ 0.3864, -2.1247,  3.0473,  2.4152, -2.6261]
post_act = act(pre_act)
# [ 0.3683, -0.9719,  0.9955,  0.9842, -0.9896]
```

$$\mathrm{Tanh}(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

Selection of output-layer activations depends on the task! E.g, Sigmoid: Binomial Classification, Softmax: Multinomial Classification, or even linear

# Training a Neural Network

# Measuring Error

Ground Truth

[ 1, 0, 1 ]

?

Predicted

[ 0, 1, 0 ]

We need a way to measure error in the NN predictions: **a loss function**.

# Loss Functions

## There's a lot of loss functions

They tend to be task specific.

`nn.CosineEmbeddingLoss` `nn.MSELoss` `nn.L1Loss` `nn.CrossEntropyLoss` `nn.NLLLoss` `nn.BCELoss`

```
y_true = torch.tensor([1.0, 0.0, 1.0])
y_pred = torch.tensor([1.0, 0.0, 1.0])
loss = nn.BCELoss() # 0.0 (perfect)

y_true = torch.tensor([1.0, 0.0, 1.0])
y_pred = torch.tensor([0.0, 1.0, 0.0])
loss = nn.BCELoss() # 100.0 (poor)
```

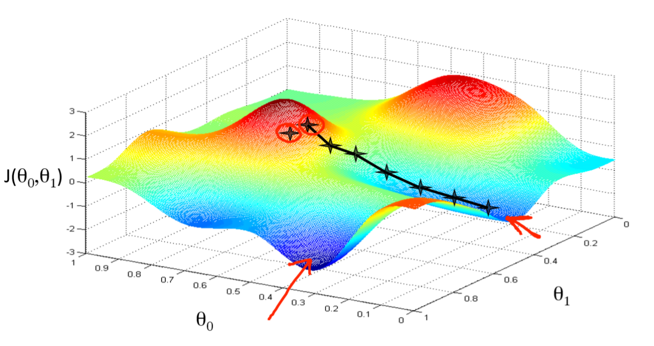
In practice: have do some research to find the best loss for your specific problem.

# Correcting for Error

**The goal of training a NN:**

Is to **minimise** the loss function.

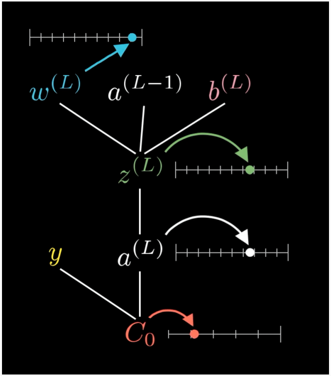
- Make the error between predictions and reality as small as possible.
- The **weights** control the loss - so we need to change the weights to reduce the loss.



# Backpropagation

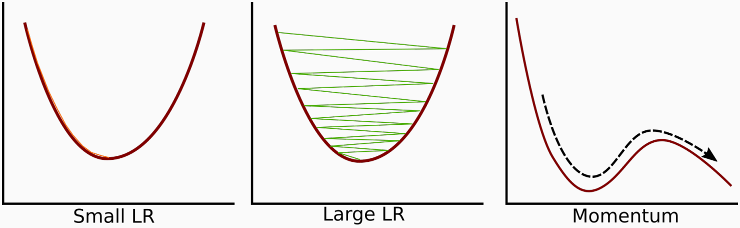
We want to go downhill - but which way is that?

- Backpropagation calculates the gradient of the loss function w.r.t the weights.
- Simply: it tells each weight in the network which direction it needs to move (up or down) to reduce the loss.



# Optimisers

- Optimisers control *how* to go downhill.
- E.g, How big a step to take




There's many available: SGD, Adam, RMSProp, Adagrad, and more, each with different dynamics.

# Practical considerations

Batches:

- Most datasets can't fit into memory
- Efficient parallelisation

Epochs:

- One pass through the entire training set

GPUs?

- Neural networks are essentially matrix multiplication - which GPUs are very good at.
- Massively parallel.

# Practical Notebook