Deep Learning for Vision

Building Deep neural networks

Learning Objectives

By the end of this chapter, student should be able to:

- To understand multilayer perceptrons
- To understand and implement different types of activation functions
- To understand loss function and backpropagation algorithm
- To train a multilayer perceptrons

Demystifying the Black Box

Vectors and matrices refresher

- A *scalar* is a single number.
- A *vector* is an array of numbers.
- A *matrix* is a 2D array.
- A *tensor* is an *n*-dimensional array

Scalar	Vector	Matrix	Tensor
1	$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$	$\begin{bmatrix} [1\ 2] & [3\ 2] \\ [1\ 7] & [5\ 4] \end{bmatrix}$

- *Scalar multiplication*—Simply multiply the scalar number by all the numbers in the matrix. Note that scalar multiplications don't change the matrix dimensions:

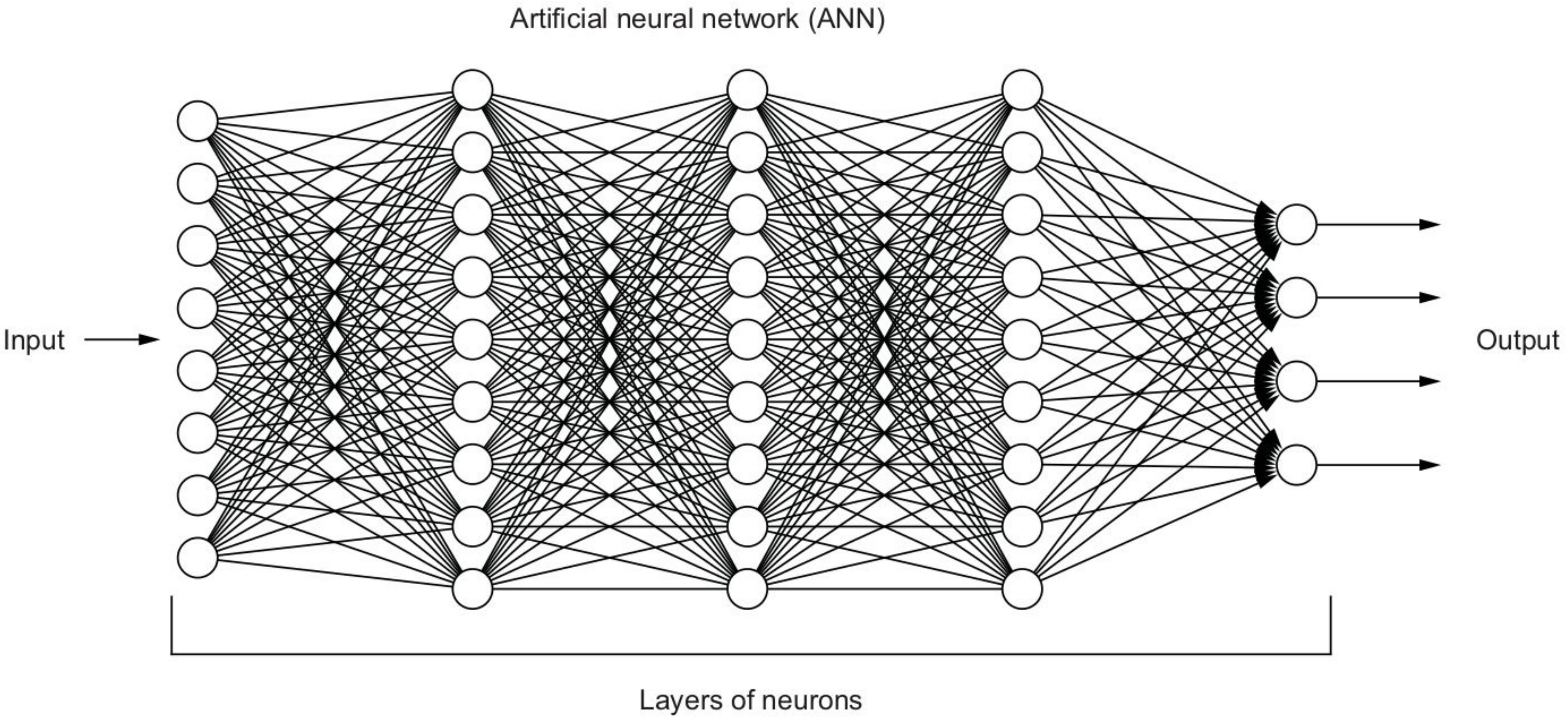
$$2 \cdot \begin{bmatrix} 10 & 6 \\ 4 & 3 \end{bmatrix} = \begin{bmatrix} 2 \cdot 10 & 2 \cdot 6 \\ 2 \cdot 4 & 2 \cdot 3 \end{bmatrix}$$

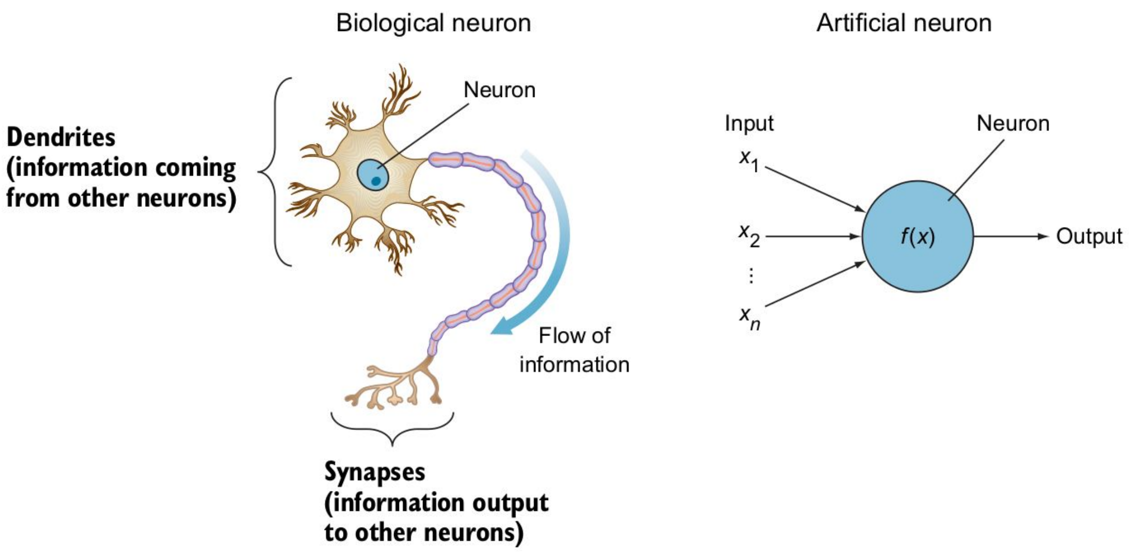
- *Matrix multiplication*—When multiplying two matrices, such as in the case of $(\text{row}_1 \times \text{column}_1) \times (\text{row}_2 \times \text{column}_2)$, column_1 and row_2 must be equal to each other, and the product will have the dimensions $(\text{row}_1 \times \text{column}_2)$. For example,

$$\begin{bmatrix} 3 & 4 & 2 \end{bmatrix} \cdot \begin{bmatrix} 13 & 9 & 7 \\ 8 & 7 & 4 \\ 6 & 4 & 0 \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix}$$

$$\begin{tikzcd} 1 \times 3 \arrow[r, leftrightarrow, bend left, "\text{Same}"] \arrow[r, leftrightarrow, bend right, "\text{Product}"'] & 3 \times 3 & 1 \times 3 \end{tikzcd}$$

where $x = 3 \cdot 13 + 4 \cdot 8 + 2 \cdot 6 = 83$, and the same for $y = 63$ and $z = 37$.

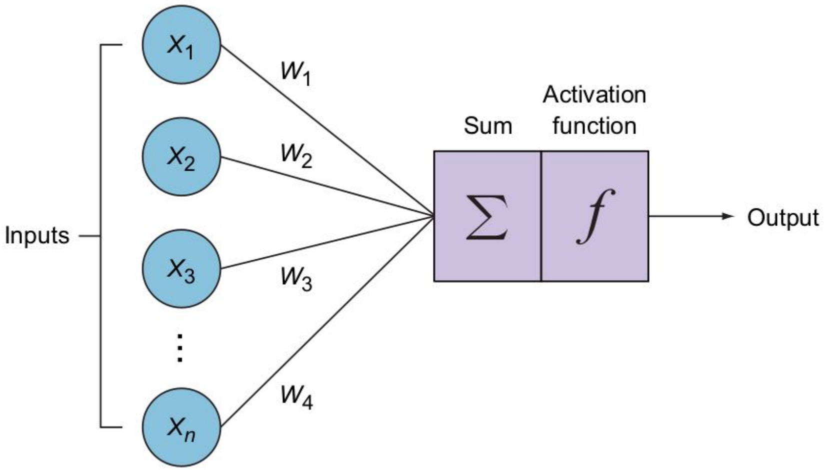

Artificial neural network (ANN)
Input
Output
Layers of neurons



- A biological neuron:
  - receives electrical signals from its dendrites
  - modulates the electrical signals in various amounts
  - fires (**activates**) an output signal through its synapses only when the total strength of the input signals exceeds a certain threshold

Perceptron

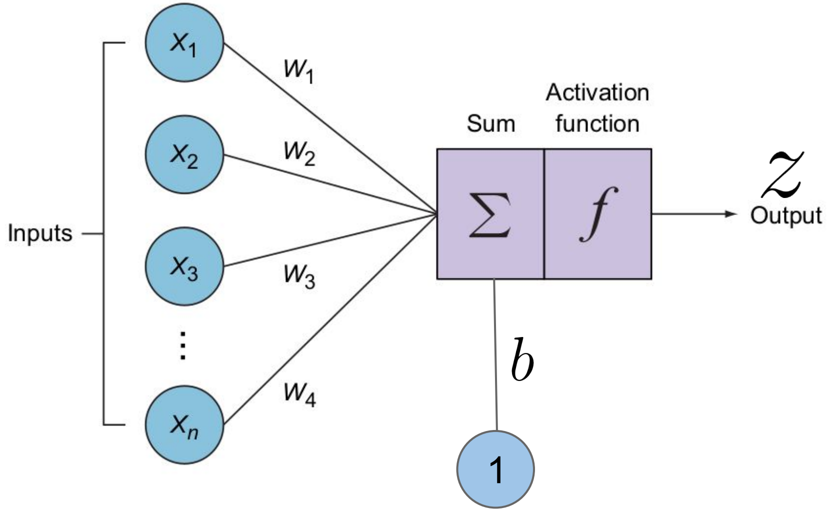
- Not all input features are equally important (Each input feature
- weight associated to a (input) signal reflects its importance in the decision-making process
- A greater absolute weight has a greater effect on the output



Perceptron

The key is learning the optimal weights for a given task

- Input vector $(x_1, x_2, x_3, \ldots, x_n)$
- Weights vector $(w_1, w_2, w_3, \ldots, w_n)$
- Neuron computation $\sum$
- Activation function f
- Output z

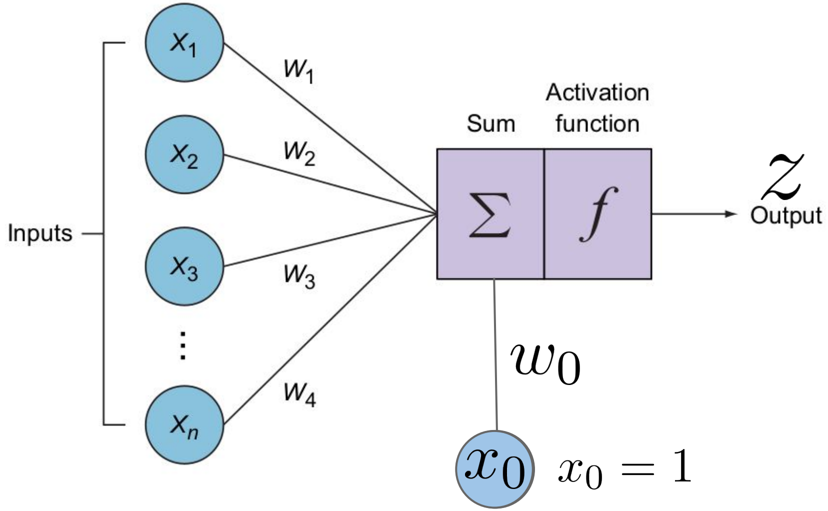


$$y = f(w_1 \times x_1 + w_2 \times x_2 + w_3 \times x_3 + \cdots + w_n \times x_n + b)$$

$$y = f(\boldsymbol{w} \times \boldsymbol{x} + b) \qquad \boldsymbol{w} = (w_1, w_2, w_3, \ldots, w_n)$$

$$\boldsymbol{x} = (x_1, x_2, x_3, \ldots, x_n)$$

- Input vector $(x_1, x_2, x_3, \ldots, x_n)$
- Weights vector $(w_1, w_2, w_3, \ldots, w_n)$
- Neuron computation $\sum$
- Activation function f
- Output z

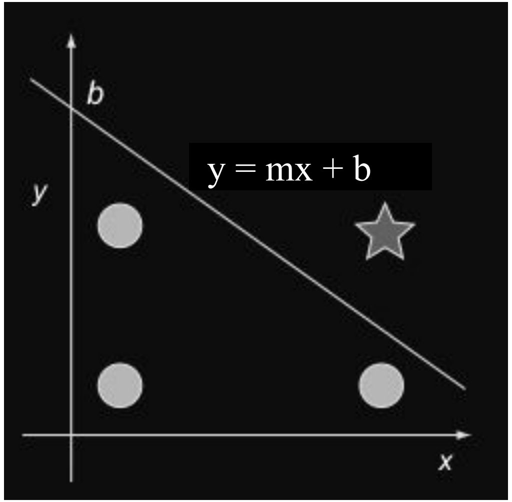


$$z = f(w_0 \times 1 + w_1 \times x_1 + w_2 \times x_2 + w_3 \times x_3 + \cdots + w_n \times x_n)$$

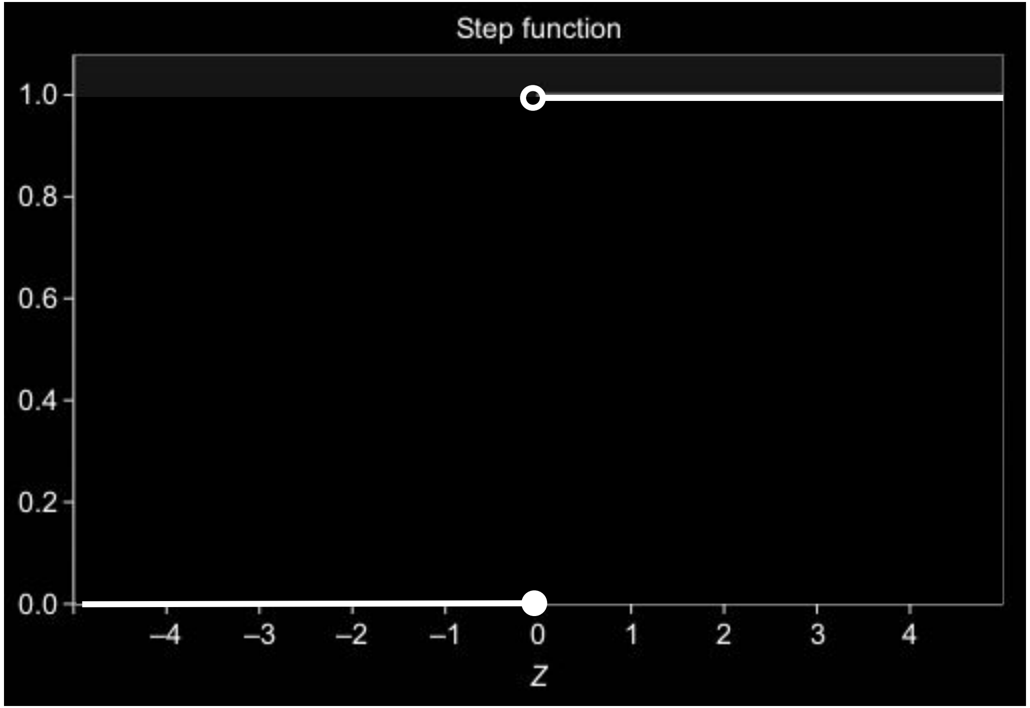
$$z = f(\boldsymbol{w} \times \boldsymbol{x})$$

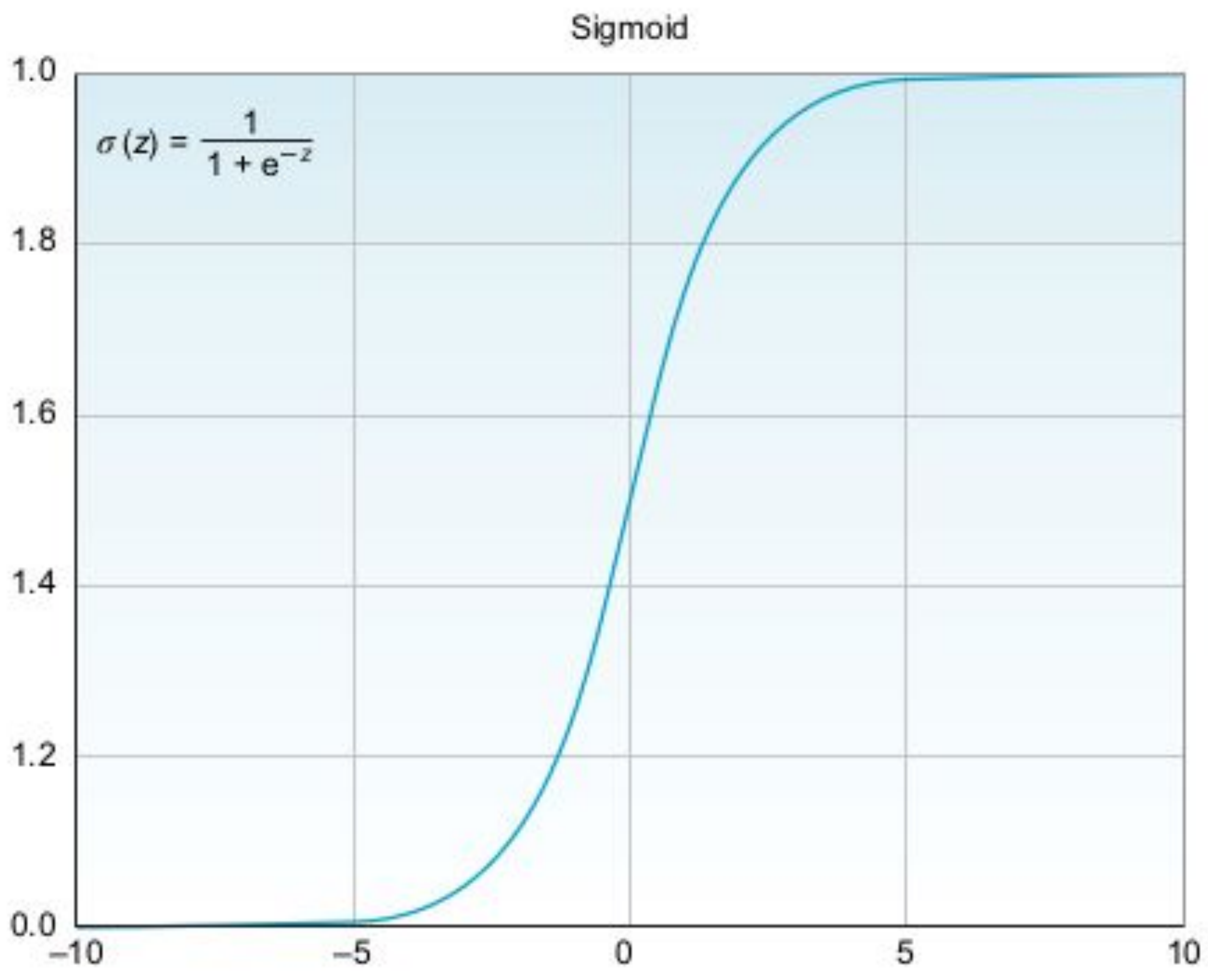
$$\boldsymbol{w} = (w_0, w_1, w_2, w_3, \ldots, w_n)$$
$$\boldsymbol{x} = (1, x_1, x_2, x_3, \ldots, x_n)$$

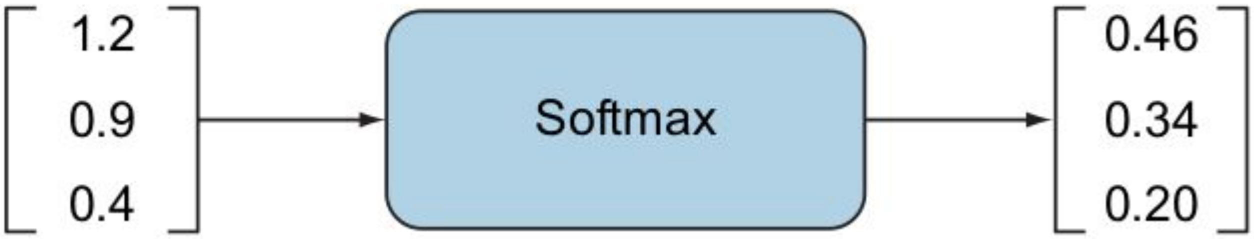
Why do we need a bias term?



$$\text{Output} = \begin{cases} 0 & \text{If } w \cdot x + b \leq 0 \\ 1 & \text{If } w \cdot x + b > 0 \end{cases}$$




Sigmoid
$\sigma(z) = \frac{1}{1 + e^{-z}}$
1.0
1.8
1.6
1.4
1.2
0.0
−10
−5
0
5
10

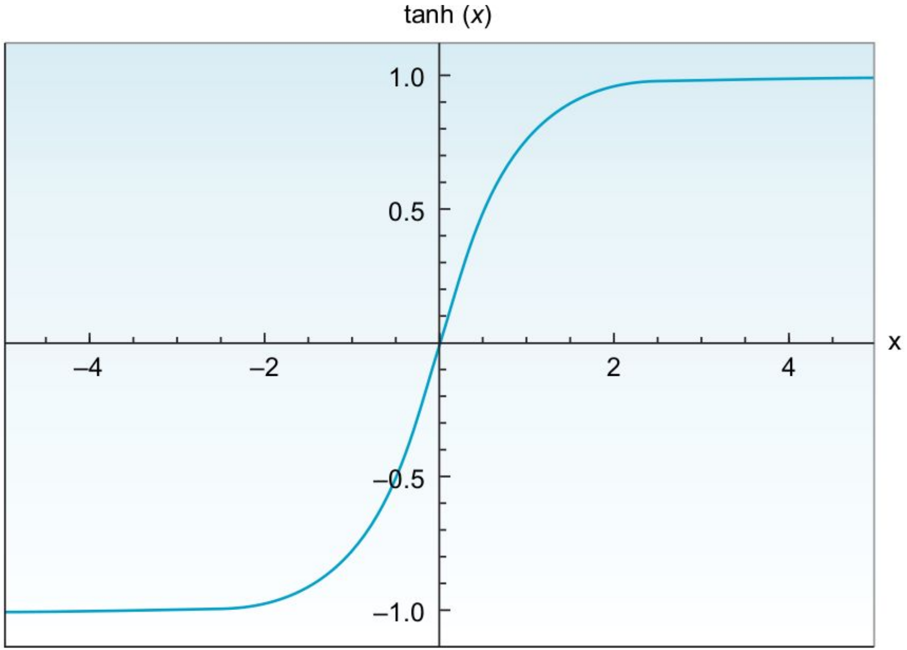


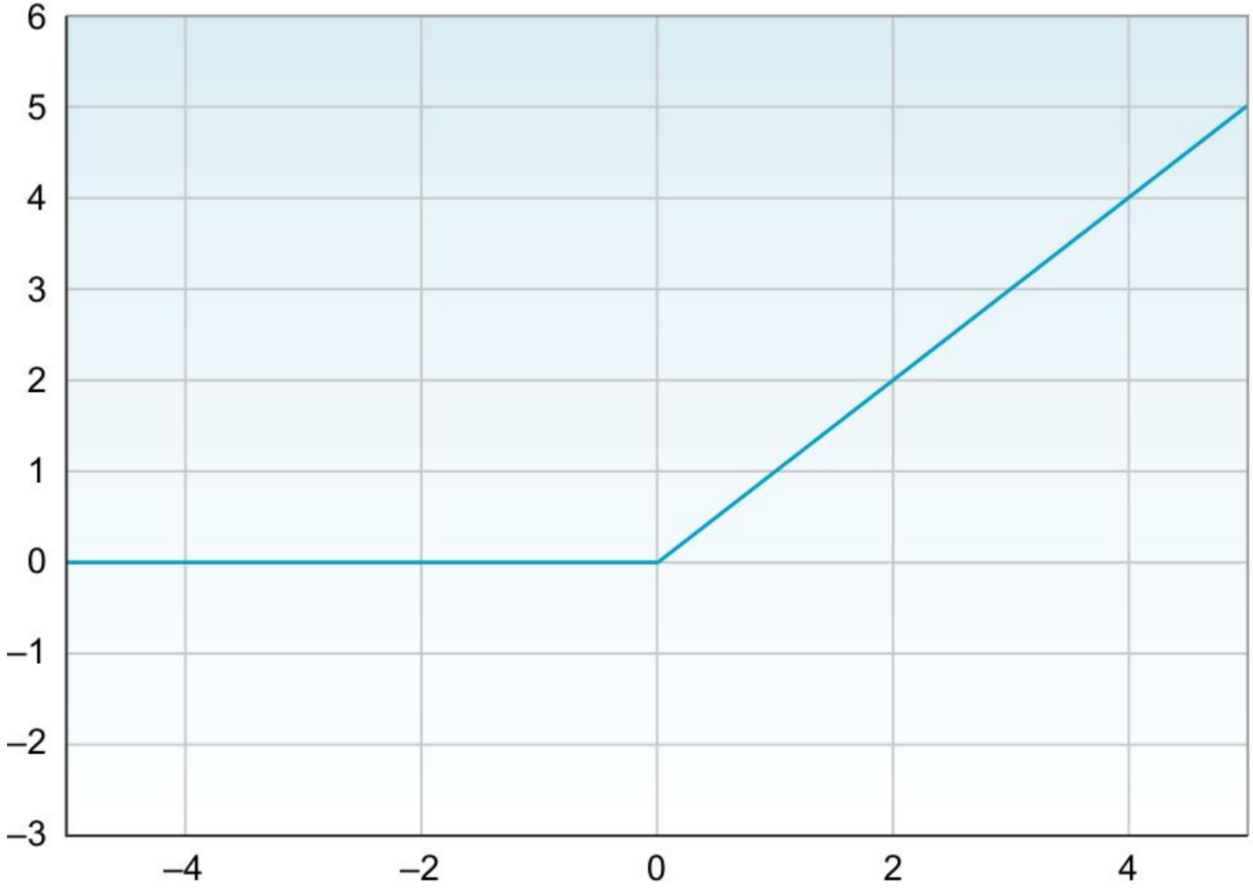
$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

```
def softmax(x):
 elements = np.exp(x)
 return elements/np.sum(elements)
```

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

```python
def tanh(x):
    ex = np.exp(x)
    e_x = np.exp(-x)
    return (ex - e_x)/(ex + e_x)
```

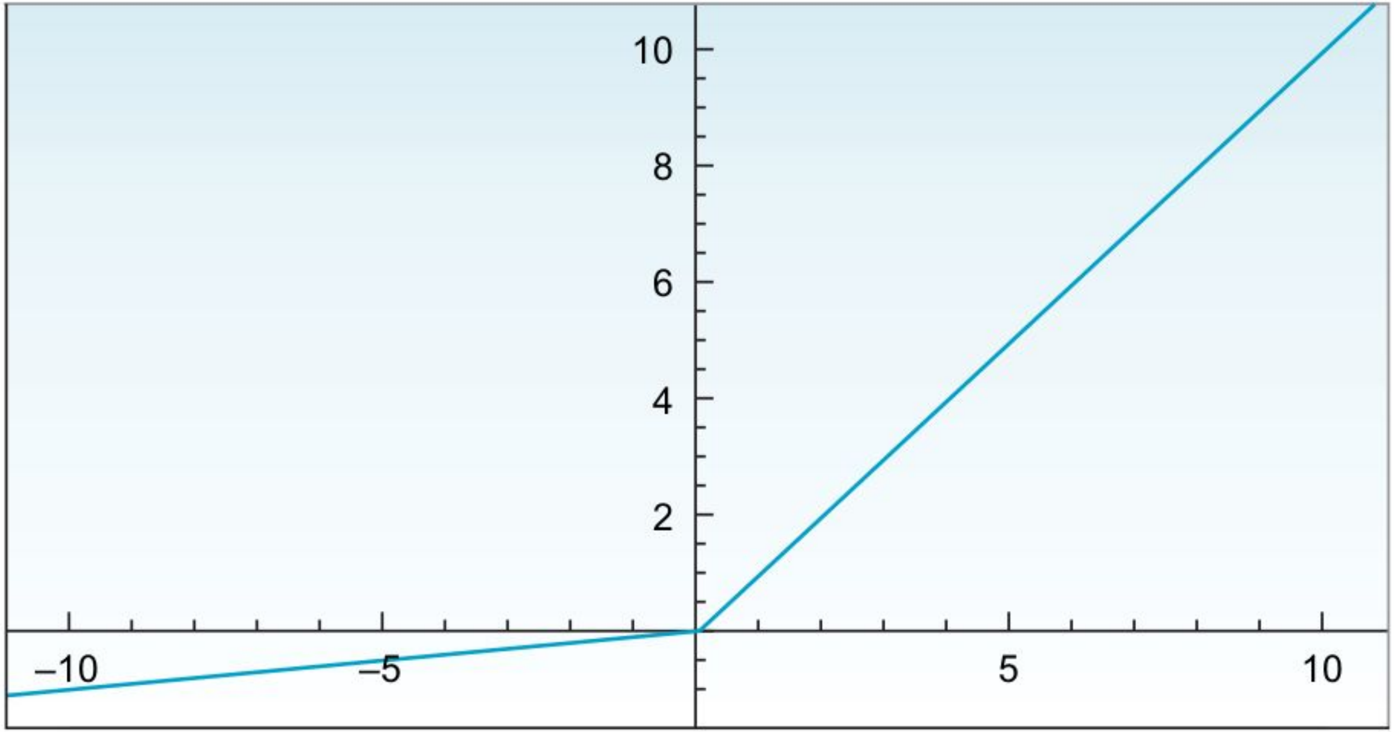




$$\mathrm{ReLU}(x) = \begin{cases} 0 \text{ if } x < 0 \\ x \text{ if } x >= 0 \end{cases}$$

```python
def relu(x):
    return np.maximum(0, x)
```

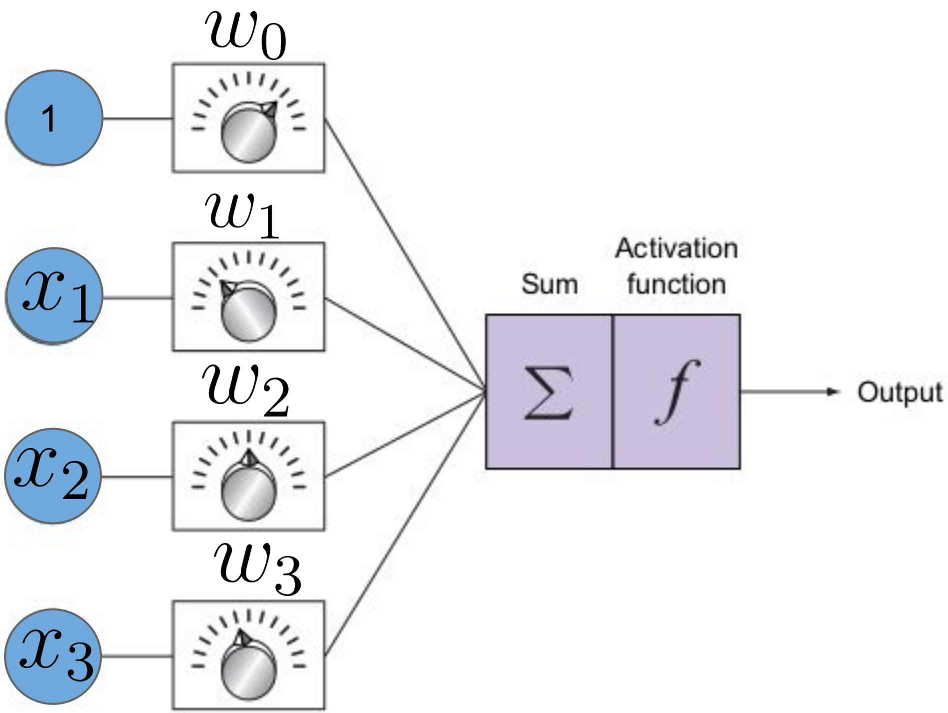
Leaky ReLU



$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x => 0 \end{cases}$$

```
def lrelu(x):
    return np.maximum(0.01*x, x)
```

How does the perceptron learn?



Analogy:

If we consider weights as knobs, training would be tuning their values up and down until the network leads to desired outputs

The Perceptron Learning Logic

1. The neuron calculates the weighted sum and applies the activation function to make a prediction $\hat{y}$. This is called the feedforward process:

$$\hat{y} = \text{activation}\left(\sum x_i \cdot w_i + b\right)$$

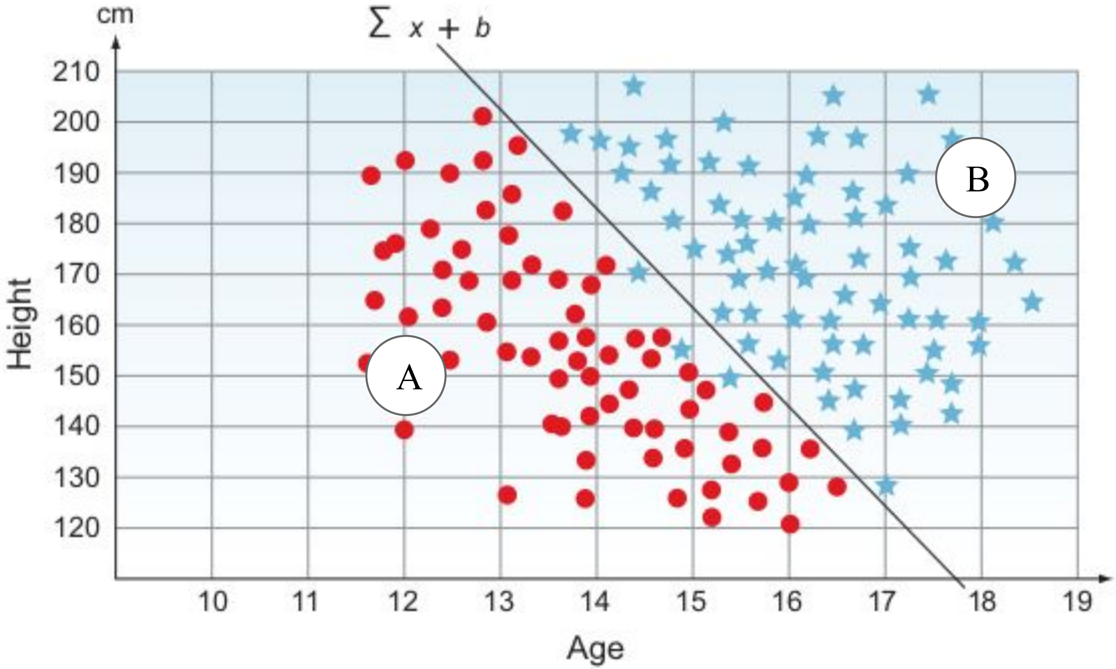
2. It compares the output prediction with the correct label to calculate the error:

$$\text{error} = y - \hat{y}$$

3. It then updates the weight. If the prediction is too high, it adjusts the weight to make a lower prediction the next time, and vice versa.
4. Repeat!

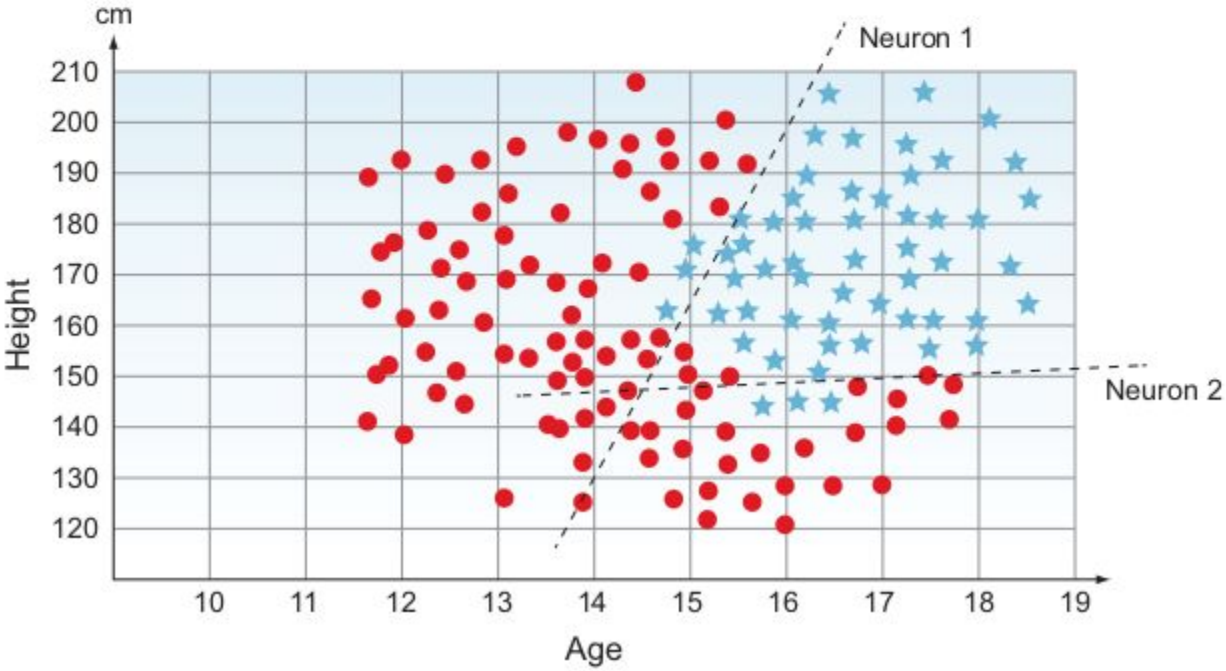
Is one neuron enough to solve complex problems?

Linearly separable data: class A (red circles) vs class B (blue stars)

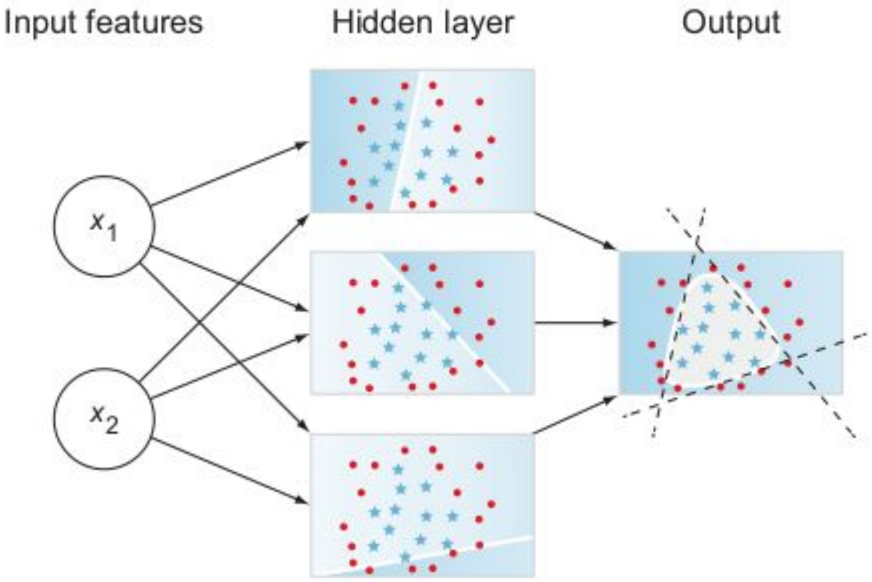


Is one neuron enough to solve complex problems?

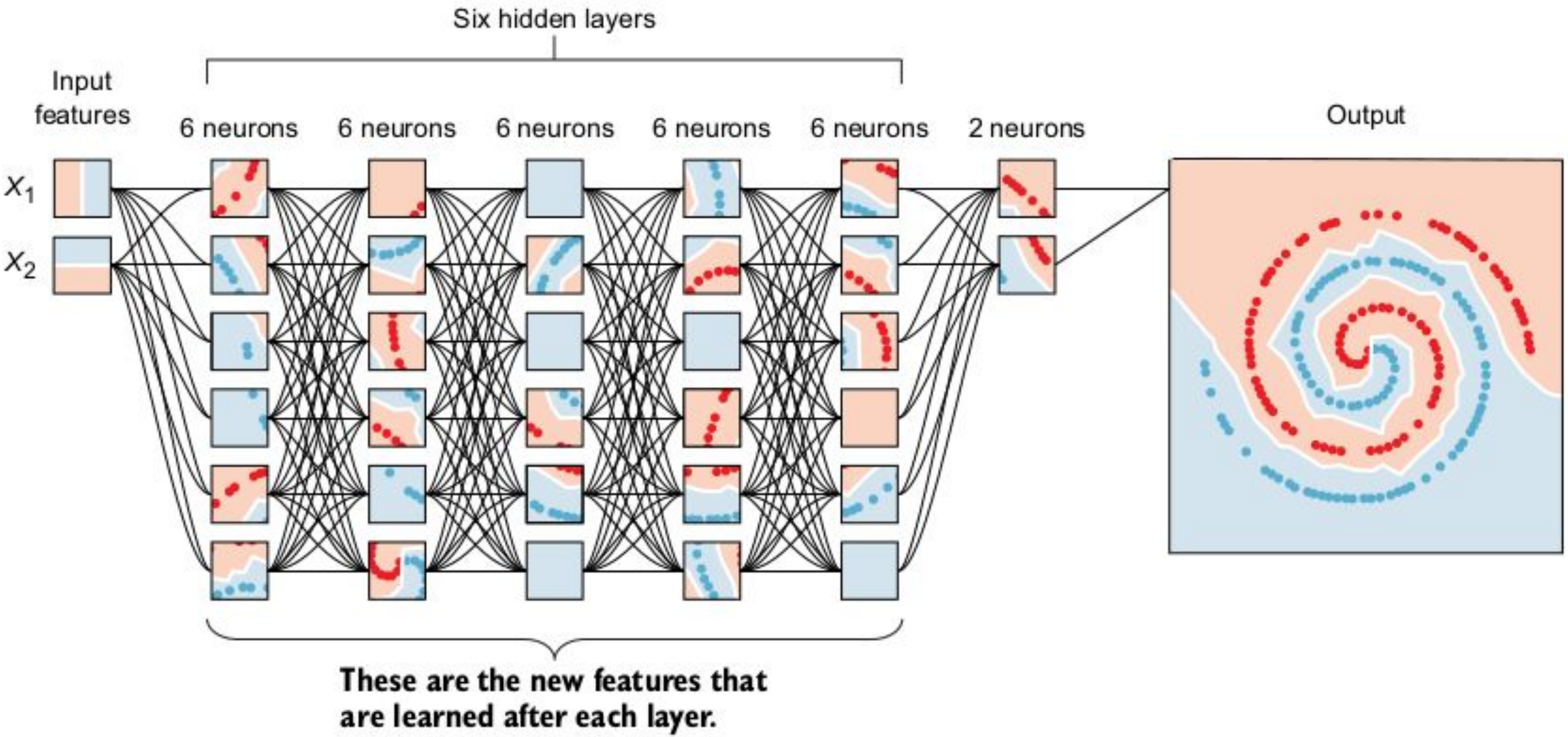
Nonlinearly separable data: class A (red circles) vs class B (blue stars)



Using more than One Neuron for Nonlinearly Separable Data

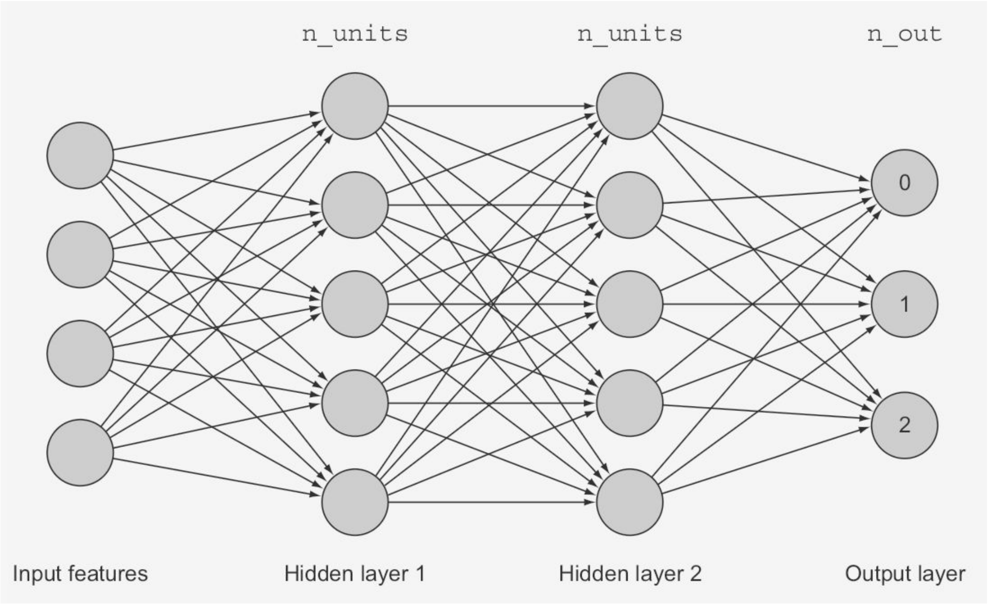


Multilayer Perceptron Architecture



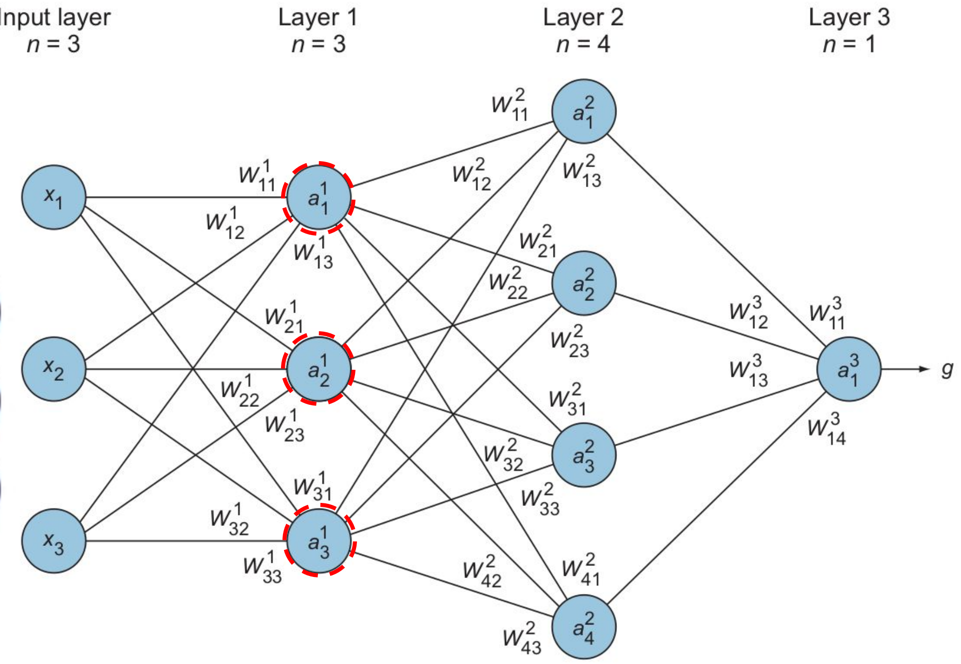
Implementing Our First Network in PyTorch

```python
import torch.nn as nn
model = torch.nn.Sequential(
    nn.Linear(4, 5, bias=True),
    nn.Linear(5, 5, bias=True),
    nn.Linear(5, 3, bias=True)
)
print(
    'Total number of model parameters:',
    sum(p.numel() for p in model.parameters())
    )
```



4x5 + 5	+	5x5 + 5	+	5x3 + 3	= **73**
Hidden layer 1		Hidden layer 2		Hidden layer 3	

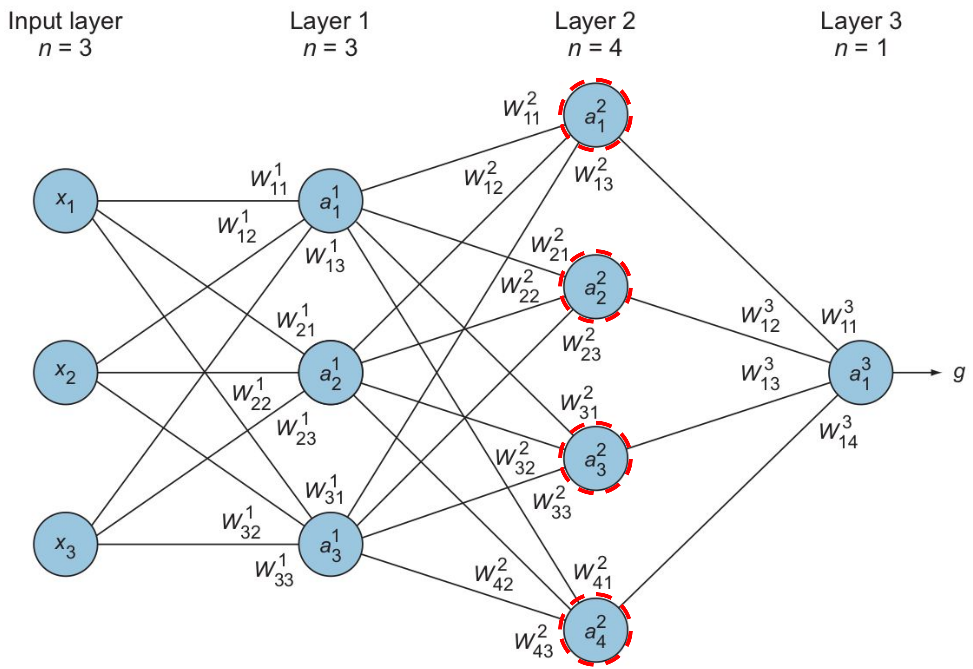
Calculation for First Hidden Layer



$$a_1^{(1)} = \sigma(w_{11}^{(1)} \times x_1 + w_{12}^{(1)} \times x_2 + w_{13}^{(1)} \times x_3)$$
$$a_2^{(1)} = \sigma(w_{21}^{(1)} \times x_1 + w_{22}^{(1)} \times x_2 + w_{23}^{(1)} \times x_3)$$
$$a_3^{(1)} = \sigma(w_{31}^{(1)} \times x_1 + w_{32}^{(1)} \times x_2 + w_{33}^{(1)} \times x_3)$$

$$W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \boldsymbol{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} \qquad \boldsymbol{a}^{(1)} = \sigma(W^{(1)} \times \boldsymbol{x})$$

$$a_1^{(2)} = \sigma(w_{11}^{(2)} \times a_1^{(1)} + w_{12}^{(2)} \times a_2^{(1)} + w_{13}^{(2)} \times a_3^{(1)})$$
$$a_2^{(2)} = \sigma(w_{21}^{(2)} \times a_1^{(1)} + w_{22}^{(2)} \times a_2^{(1)} + w_{23}^{(2)} \times a_3^{(1)})$$
$$a_3^{(2)} = \sigma(w_{31}^{(2)} \times a_1^{(1)} + w_{32}^{(2)} \times a_2^{(1)} + w_{33}^{(2)} \times a_3^{(1)})$$
$$a_4^{(2)} = \sigma(w_{41}^{(2)} \times a_1^{(1)} + w_{42}^{(2)} \times a_2^{(1)} + w_{43}^{(2)} \times a_3^{(1)})$$



$$W^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} & w_{23}^{(2)} \\ w_{31}^{(2)} & w_{32}^{(2)} & w_{33}^{(2)} \\ w_{41}^{(2)} & w_{42}^{(2)} & w_{43}^{(2)} \end{bmatrix} \qquad \boldsymbol{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix}$$

$$\boldsymbol{a}^{(2)} = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \\ a_4^{(2)} \end{bmatrix}$$

$$\boldsymbol{a}^{(2)} = \sigma(W^{(2)} \times \boldsymbol{a}^{(1)})$$

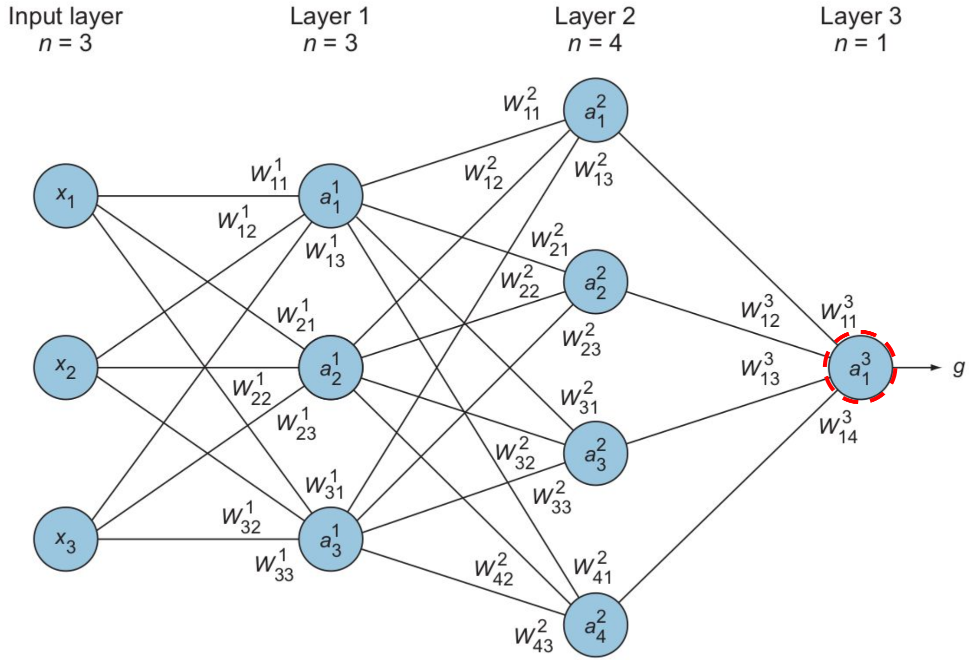
Calculation for Second Hidden Layer

$$\hat{y} = a_1^{(3)} = \sigma(w_{11}^{(3)} \times a_1^{(2)} + w_{12}^{(3)} \times a_2^{(2)} + w_{13}^{(3)} \times a_3^{(2)} + w_{14}^{(3)} \times a_4^{(2)})$$

$$W^{(3)} = \begin{bmatrix} w_{11}^{(3)} & w_{12}^{(3)} & w_{13}^{(3)} & w_{14}^{(4)} \end{bmatrix}$$

$$\boldsymbol{a}^{(2)} = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \\ a_4^{(2)} \end{bmatrix} \qquad \boldsymbol{a}^{(3)} = \begin{bmatrix} a_1^{(3)} \end{bmatrix}$$

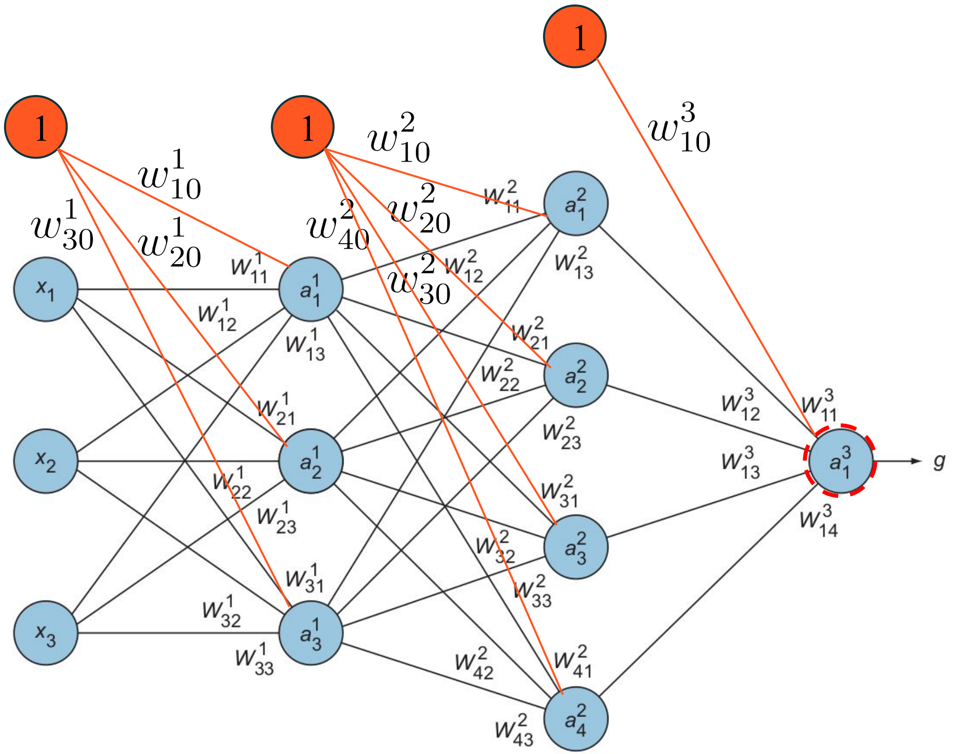
$$\hat{\boldsymbol{y}} = \boldsymbol{a}^{(3)} = \sigma(W^{(3)} \times \boldsymbol{a}^{(2)})$$



Calculating Output

$$W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

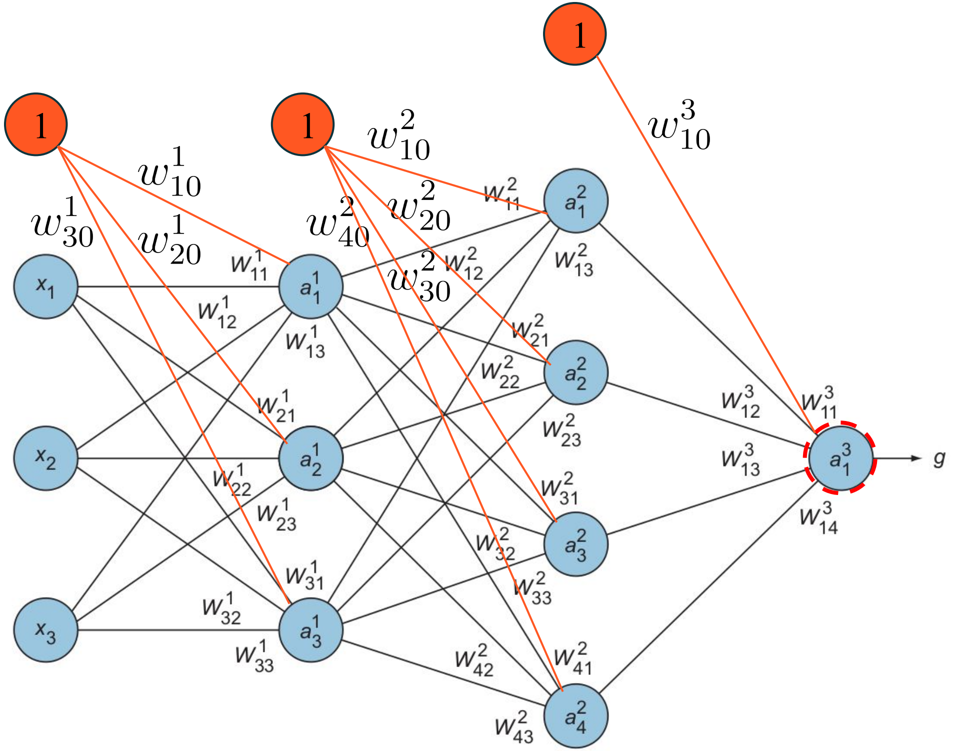
$$W^{(1)} = \begin{bmatrix} w_{10}^{(1)} & w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{20}^{(1)} & w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{30}^{(1)} & w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$



What Happened to the Bias values

$$W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$W^{(1)}x + B \qquad B = \begin{bmatrix} w_{10}^1 \\ w_{10}^2 \\ w_{10}^3 \end{bmatrix}$$



What Happened to the Bias values

$$W^{(3)} \times (W^{(2)} \times (W^{(1)} \times X + B_1) + B_2) + B_3$$

Loss functions

Mean squared error (MSE)

$$E(W, b) = \frac{1}{N}\sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

Loss functions

Cross-entropy

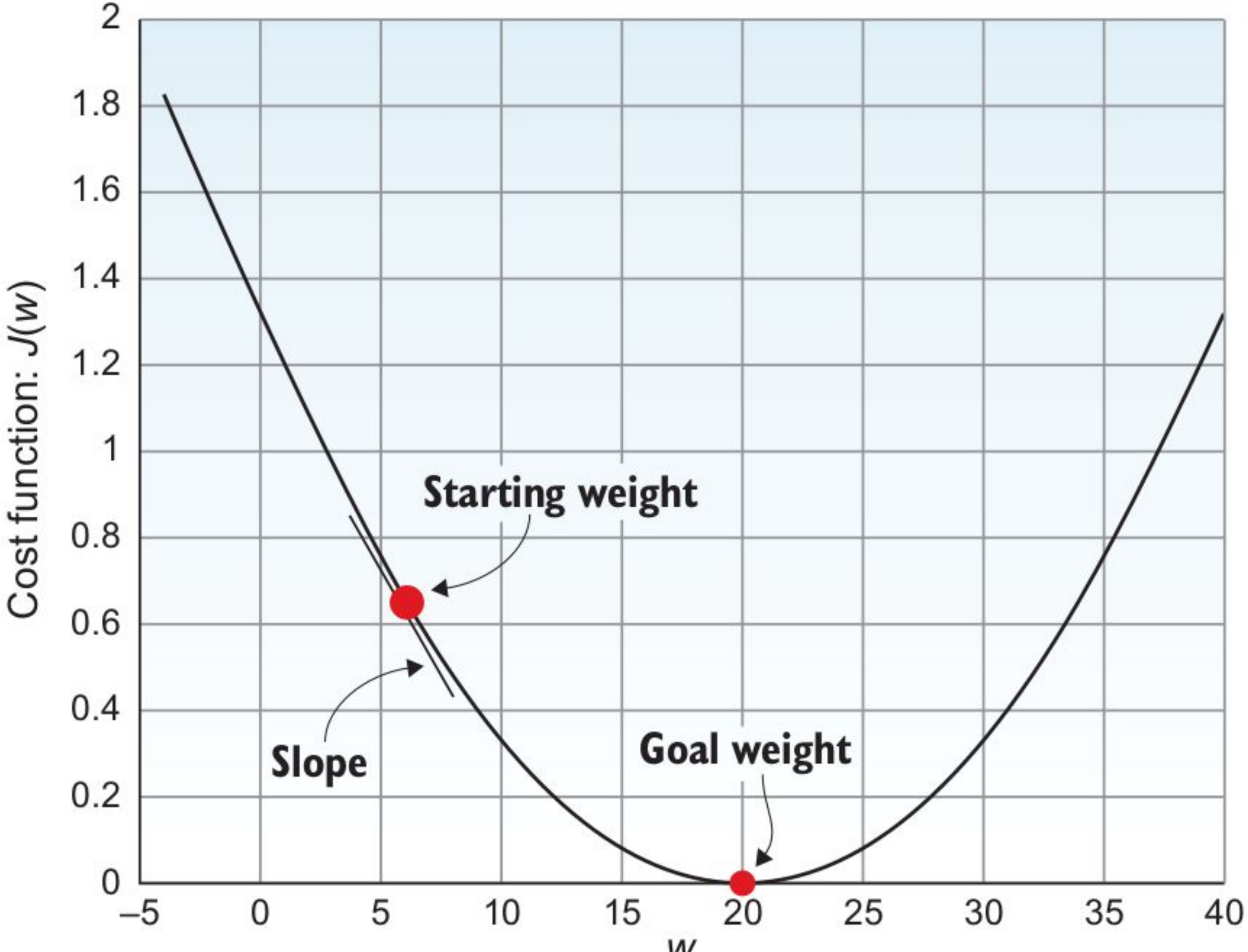
$$E(W, b) = -\sum_{i=1}^{m} y_i \log(p_i)$$

$$E(W, b) = -\sum_{i=1}^{n}\sum_{i=1}^{m} \hat{y}_{ij} \log(p_{ij})$$

Probability(cat)	P(dog)	P(fish)
0.0	1.0	0.0

Probability(cat)	P(dog)	P(fish)
0.2	0.3	0.5

```
E = - (0.0 * log(0.2) + 1.0 * log(0.3) + 0.0 * log(0.5)) = 1.2
```


Cost function: J(w)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
–5
0
5
10
15
20
25
30
35
40
w
Starting weight
Slope
Goal weight

Partial derivative

$$\frac{d}{dw}E(x) \quad \text{or just} \quad \frac{dE(x)}{dw}$$

Constant Rule: $\frac{d}{dx}(c) = 0$	Difference Rule: $\frac{d}{dx}[f(x) - g(x)] - f'(x) - g'(x)$
Constant Multiple Rule: $\frac{d}{dx}[cf(x)] = cf'(x)$	Product Rule: $\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$
Power Rule: $\frac{d}{dx}(x^n) = x^{n-1}$	Quotient Rule: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$
Sum Rule: $\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x)$	Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$

Constant Rule: $\frac{d}{dx}(c) = 0$	Difference Rule: $\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x)$
Constant Multiple Rule: $\frac{d}{dx}[cf(x)] = cf'(x)$	Product Rule: $\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$
Power Rule: $\frac{d}{dx}(x^n) = x^{n-1}$	Quotient Rule: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$
Sum Rule: $\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x)$	Chain Rule: $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$

$$f(x) = 10x^5 + 4x^7 + 12x$$

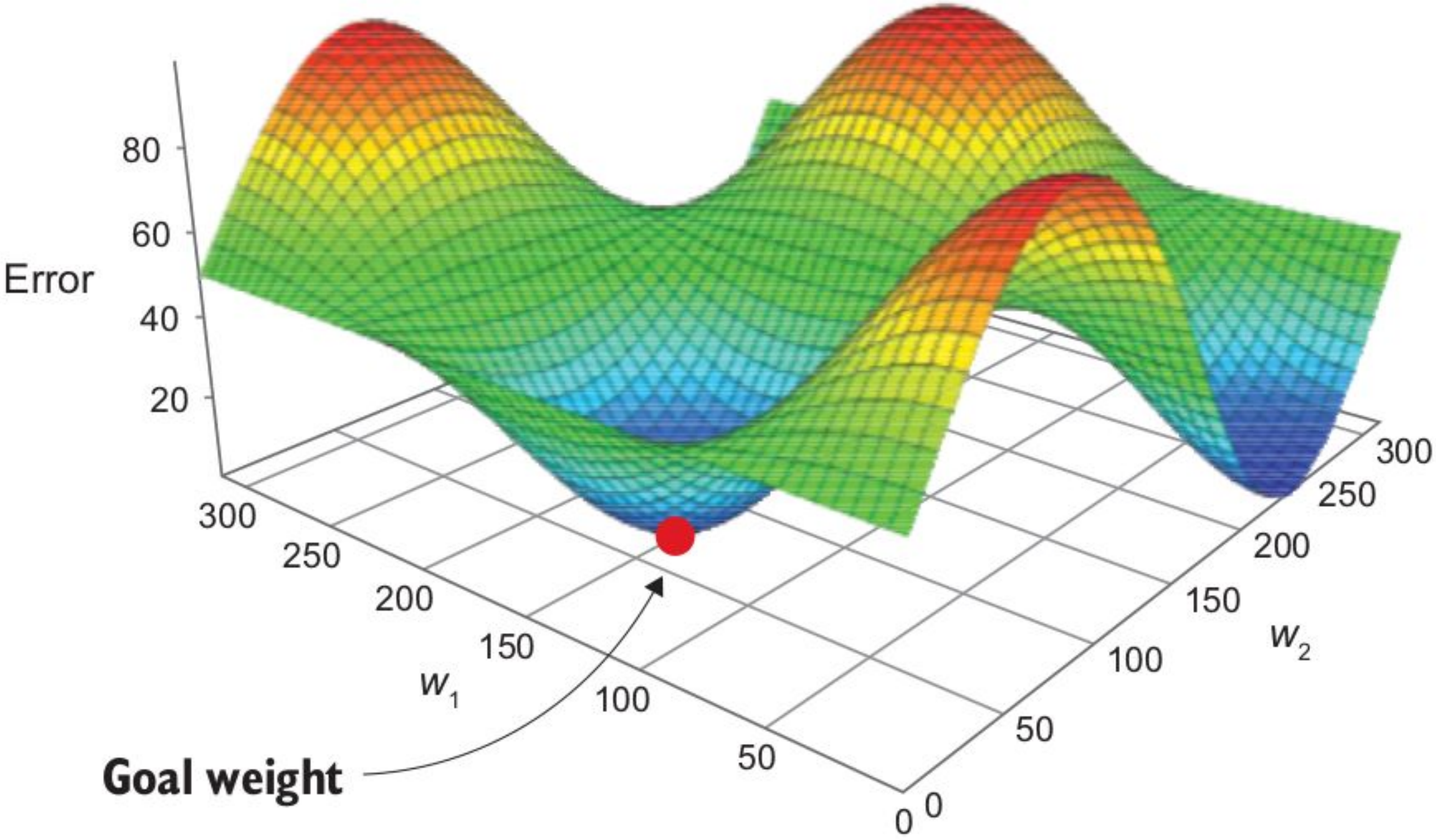
$$f'(x) =$$

$$
\begin{aligned}
\frac{d}{dx}\,\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \\
&= \frac{d}{dx}\,(1+e^{-x})^{-1} \qquad \textbf{power rule} \\
&= -(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^{2}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \sigma(x) \cdot (1-\sigma(x))
\end{aligned}
$$

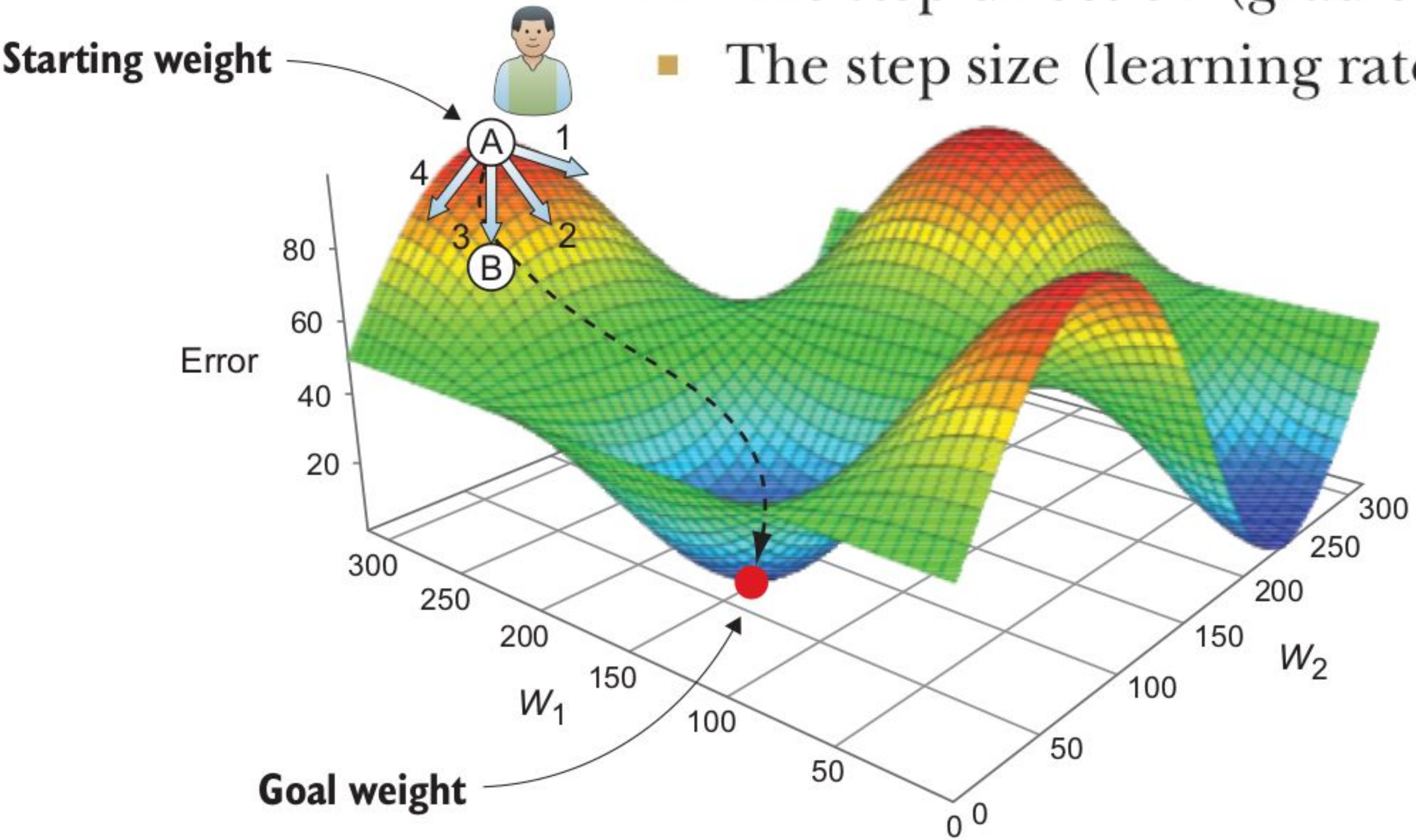
If you want to write out the derivative of the sigmoid activation function in code, it will look like this:

```
def sigmoid(x):
    return 1/(1+np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```


Error
80
60
40
20
300
250
200
150
100
50
0
w_1
0
50
100
150
200
250
300
w_2
Goal weight

- The step direction (gradient)
- The step size (learning rate)



$$\Delta w_i = -\alpha \frac{dE}{dw_i}$$

$$w_{\text{next–step}} = w_{\text{current}} + \Delta w$$

Derivative of error
with respect to weight

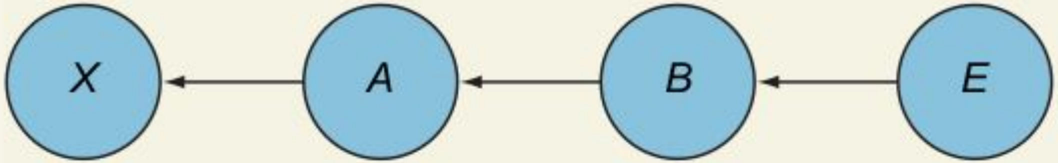
Old weight

$$W_{new} = W_{old} - \alpha \left(\frac{\partial Error}{\partial W_x}\right)$$

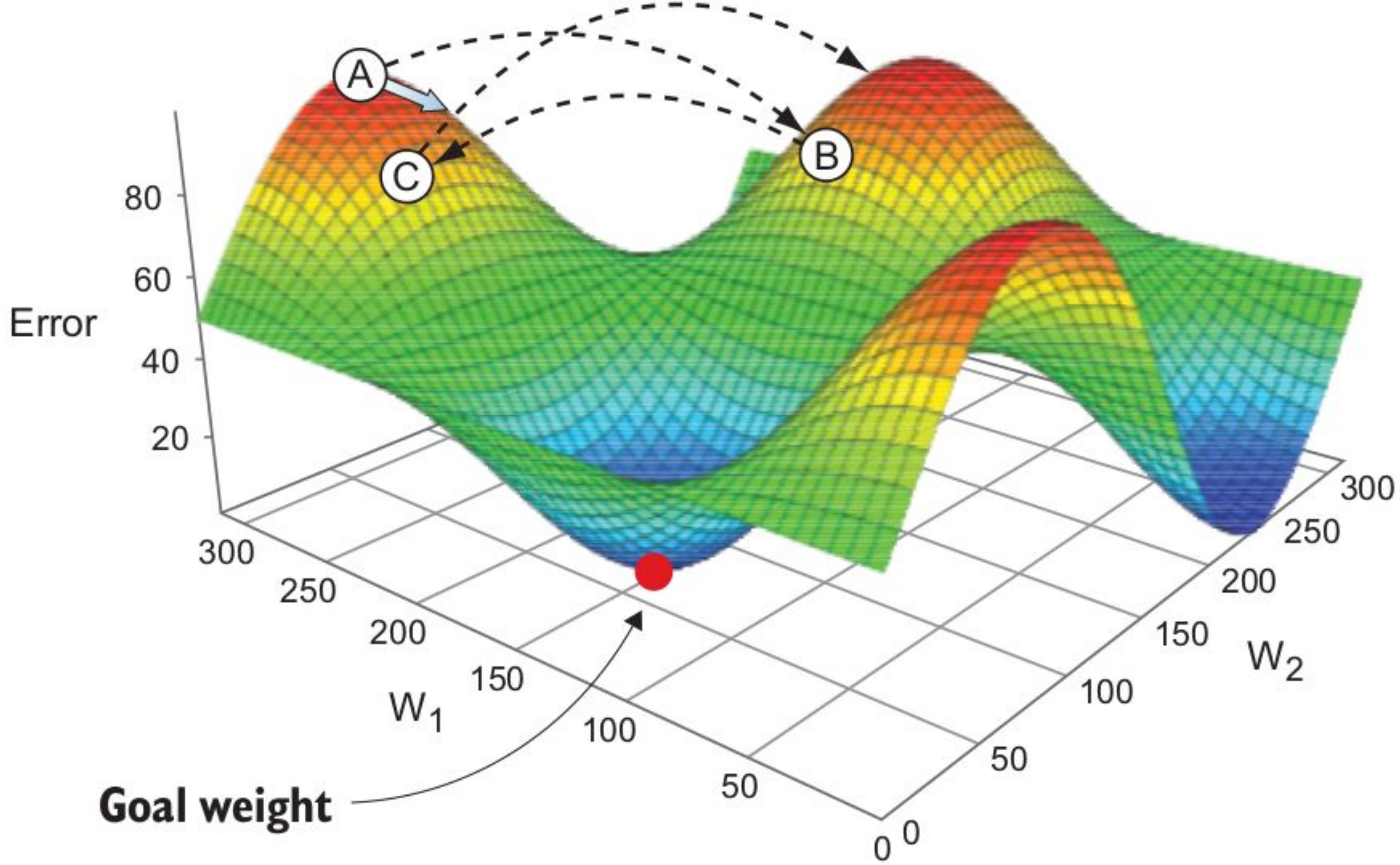
New weight

Learning rate

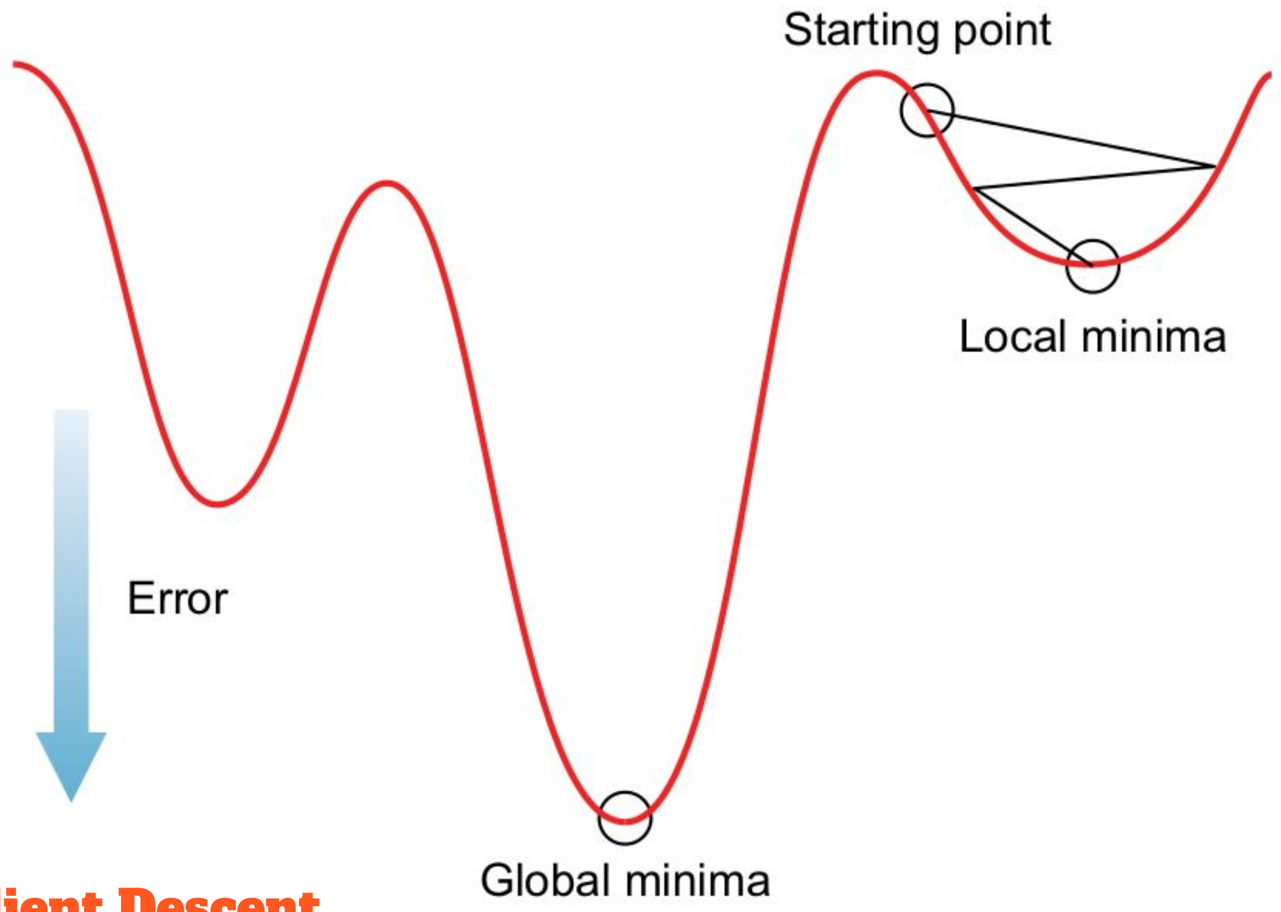
Chain Rule: $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$



$$\frac{dE}{dx} = \frac{dE}{dB} \cdot \frac{dB}{dA} \cdot \frac{dA}{dx}$$



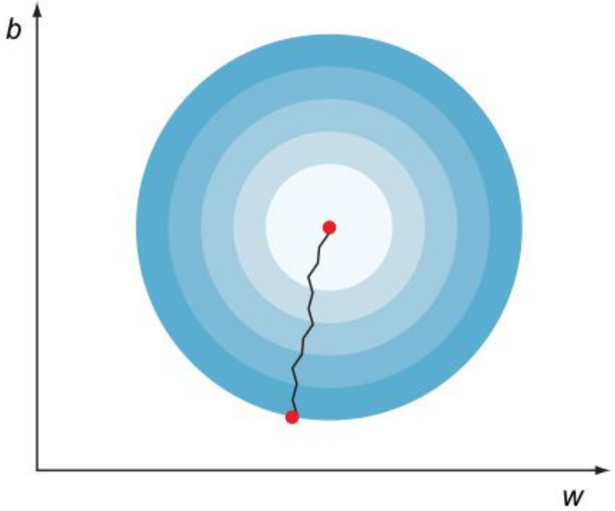
Setting a very large learning rate causes the error to oscillate and never descend.



Pitfalls Of Batch Gradient Descent

GD

1. Take all the data.
2. Compute the gradient.
3. Update the weights and take a step down.

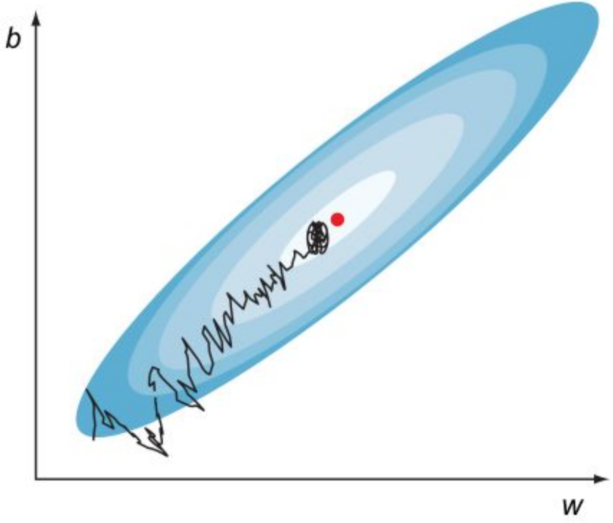


4. Repeat for *n* number of epochs (iterations).

A smooth path for the GD down the error curve

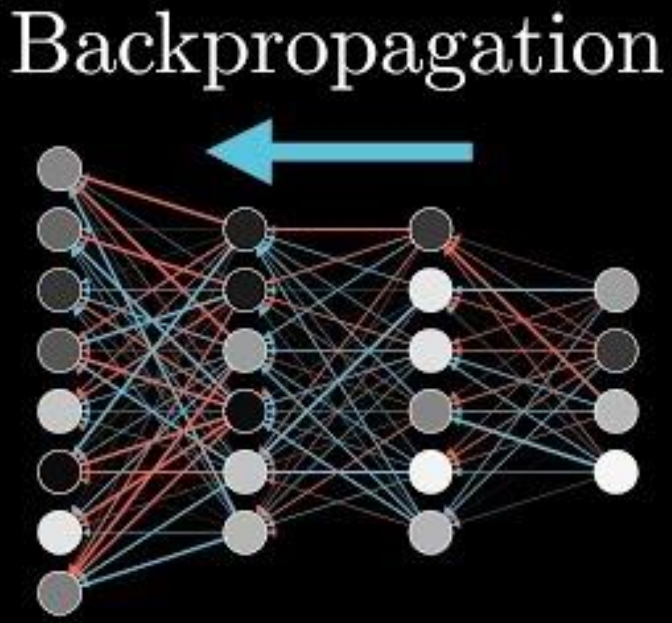
Stochastic GD

1. Randomly shuffle samples in the training set.
2. Pick one data instance.
3. Compute the gradient.
4. Update the weights and take a step down.
5. Pick another one data instance.
6. Repeat for *n* number of epochs (training iterations).



An oscillated path for SGD down the error curve

Mini-batch Gradient Descent
An Alternative



https://www.youtube.com/watch?v=Ilg3gGewQ5U&t=4s&ab_channel=3Blue1Brown