INTRODUCTION TO MACHINE LEARNING METHODS

Toni Šćulac

Faculty of Science, University of Split, Croatia

Corresponding Associate, CERN

tCSC Machine Learning 2024, Split, Croatia

LECTURES OUTLINE

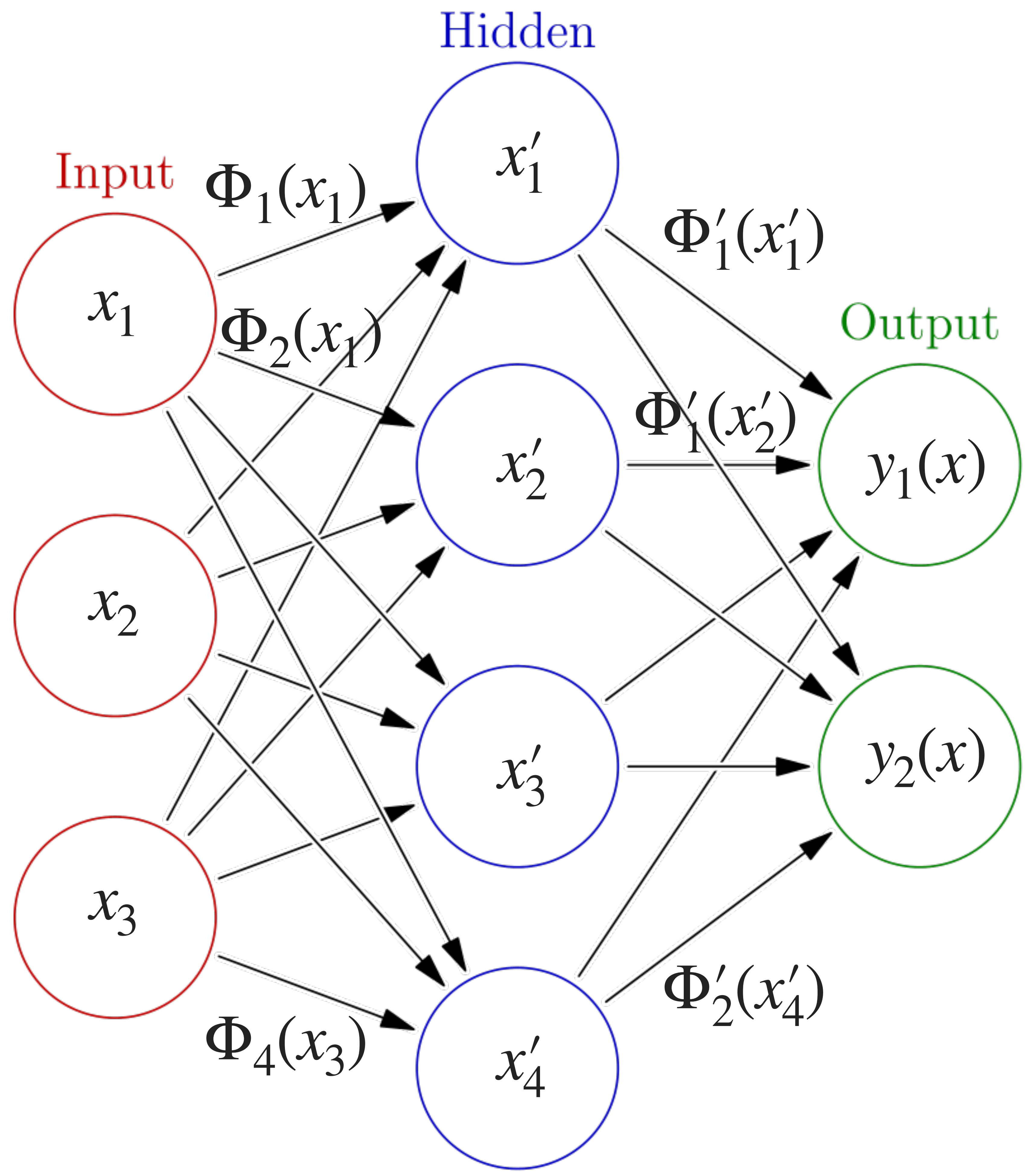
1) Introduction to Statistics
2) Statistics and Machine Learning
3) Classical Machine Learning
4) Introduction to Deep Learning
5) Advanced Deep Learning

INTRODUCTION TO DEEP LEARNING

*inspiration and examples from [3B1B] and [S. Zhang]

NEURAL NETWORKS

- Originate from attempts to model neural processes
- The input is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs.
- Neurons typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection.
- Neurons are aggregated into layers. Different layers may perform different transformations on their inputs.
- Can be viewed as a specific way of parametrising transformation functions $\Phi(x)$
  - transformation functions are usually called **activation functions**

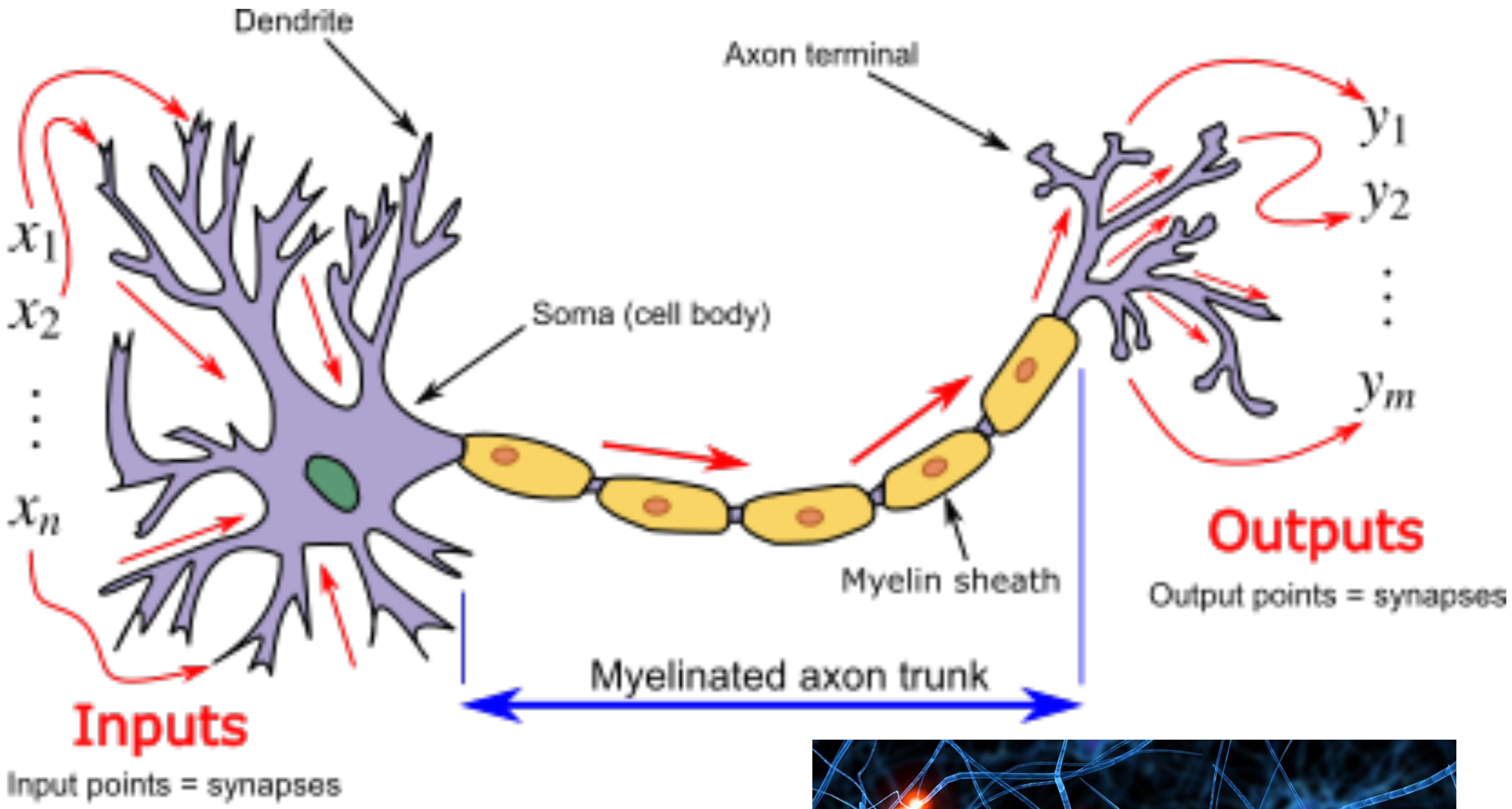


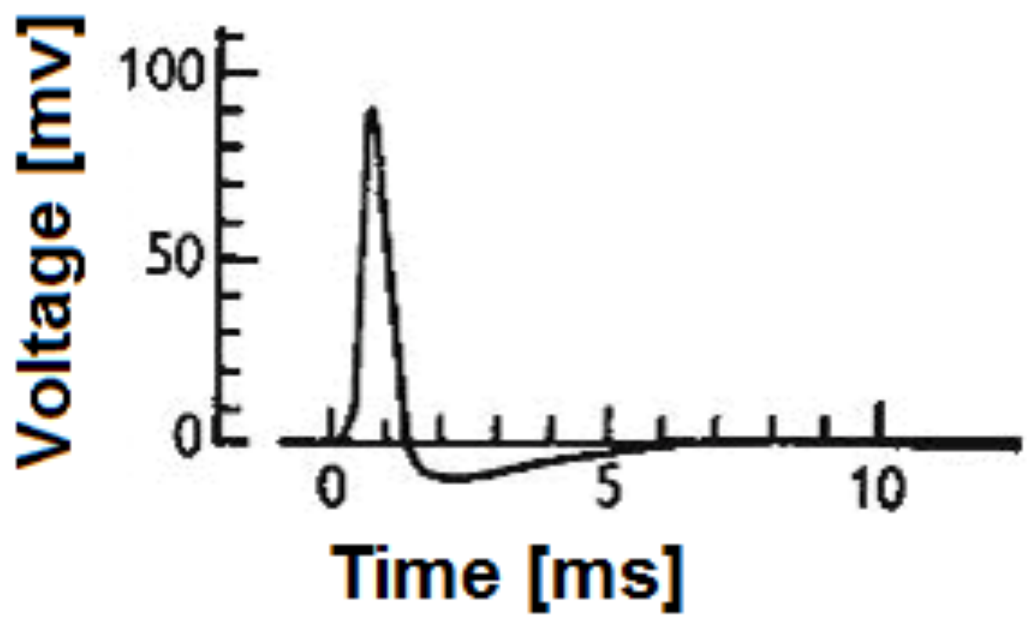

DATASHEET
Machine learning demystified
BLA
Machine Le
AI
BDT
Neural Networks

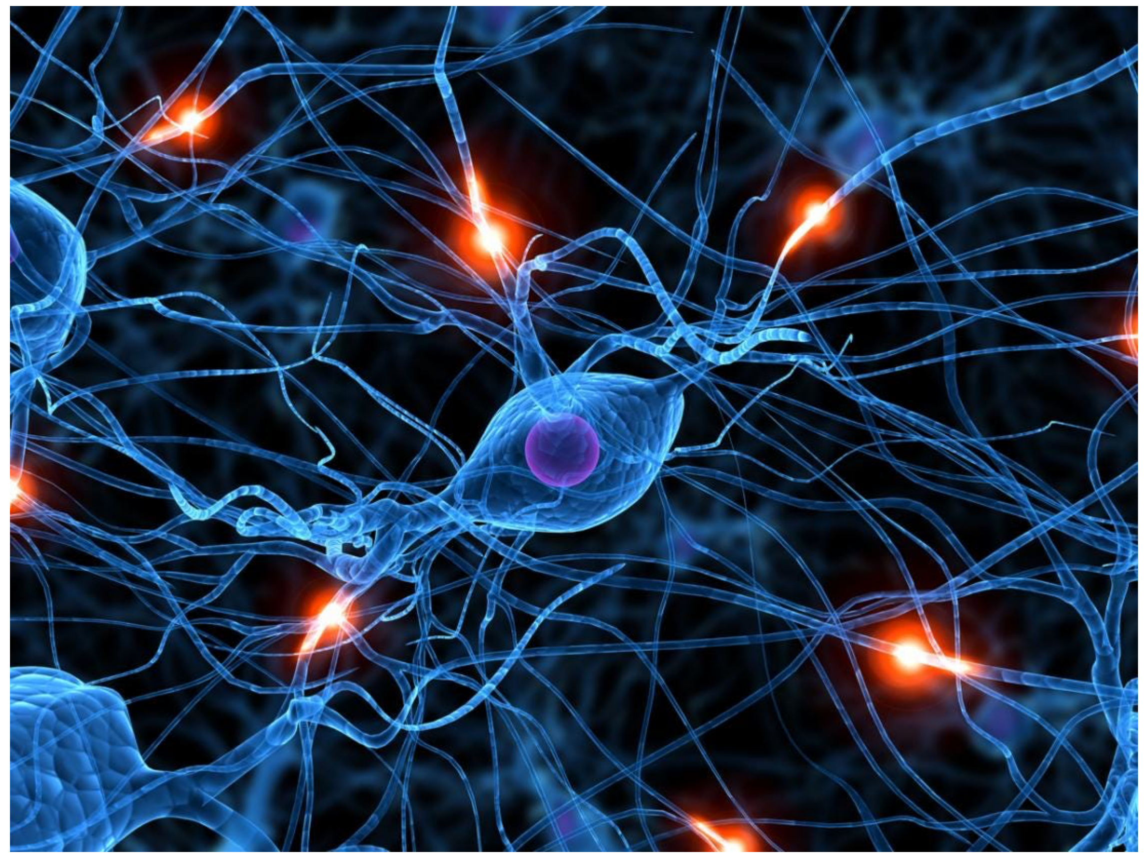
NEURAL NETWORKS

- Neural networks (NNs) are a ML model inspired by the structure and function of biological neural networks in animal brains.

- **Neurons** (or nerve cells) are electrically excitable cells able to fire electric signals across a neural network
- Collects inputs $(x_1, \cdots, x_N)$ from other neurons using **dendrites**
- Gathers all the inputs, and fires an electric signal if some conditions are met
- The fired signal is then sent to other neurons through the **axon**
- Our brain is an extremely large interconnected network of neurons that processes huge amounts of data and tries to model the World



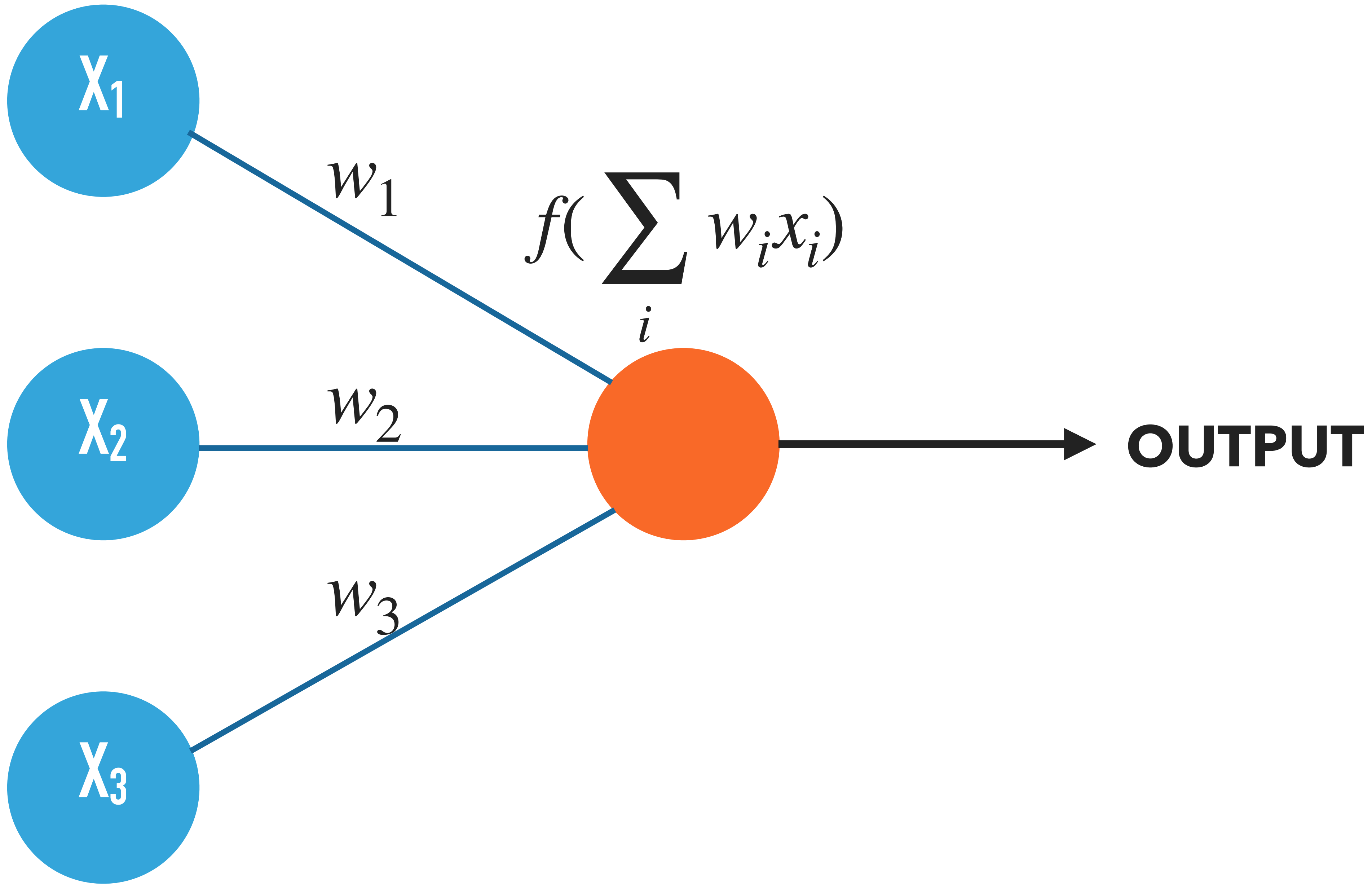




ARTIFICIAL NEURAL NETWORKS

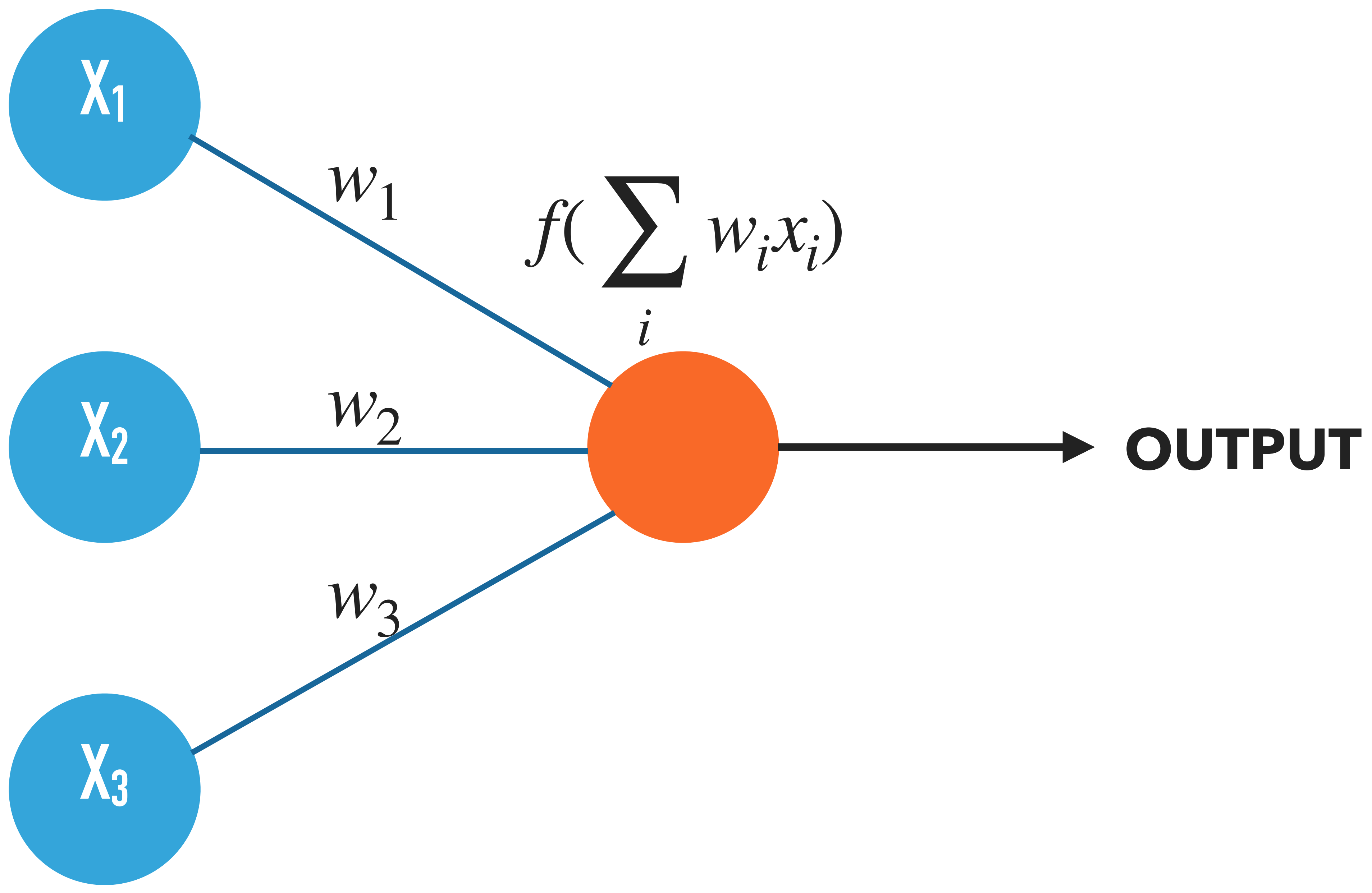
- Why not copy nature? Let's try to build the same thing, but in python (c++)…
- Keeping in mind our goal of approximating the LR
- Let's start simple, by designing an artificial Neural Network with:

- **3 input nodes:**
  - to keep it real simple in the beginning, let's limit ourselves to 0-1 input
  - strength of input signal is modelled by **weights**
- **1 neuron:**
  - we will mimic electric signal of a neuron with an **activation function**
- **1 output:**
  - just like in DT it can be 0(b) or 1(s)

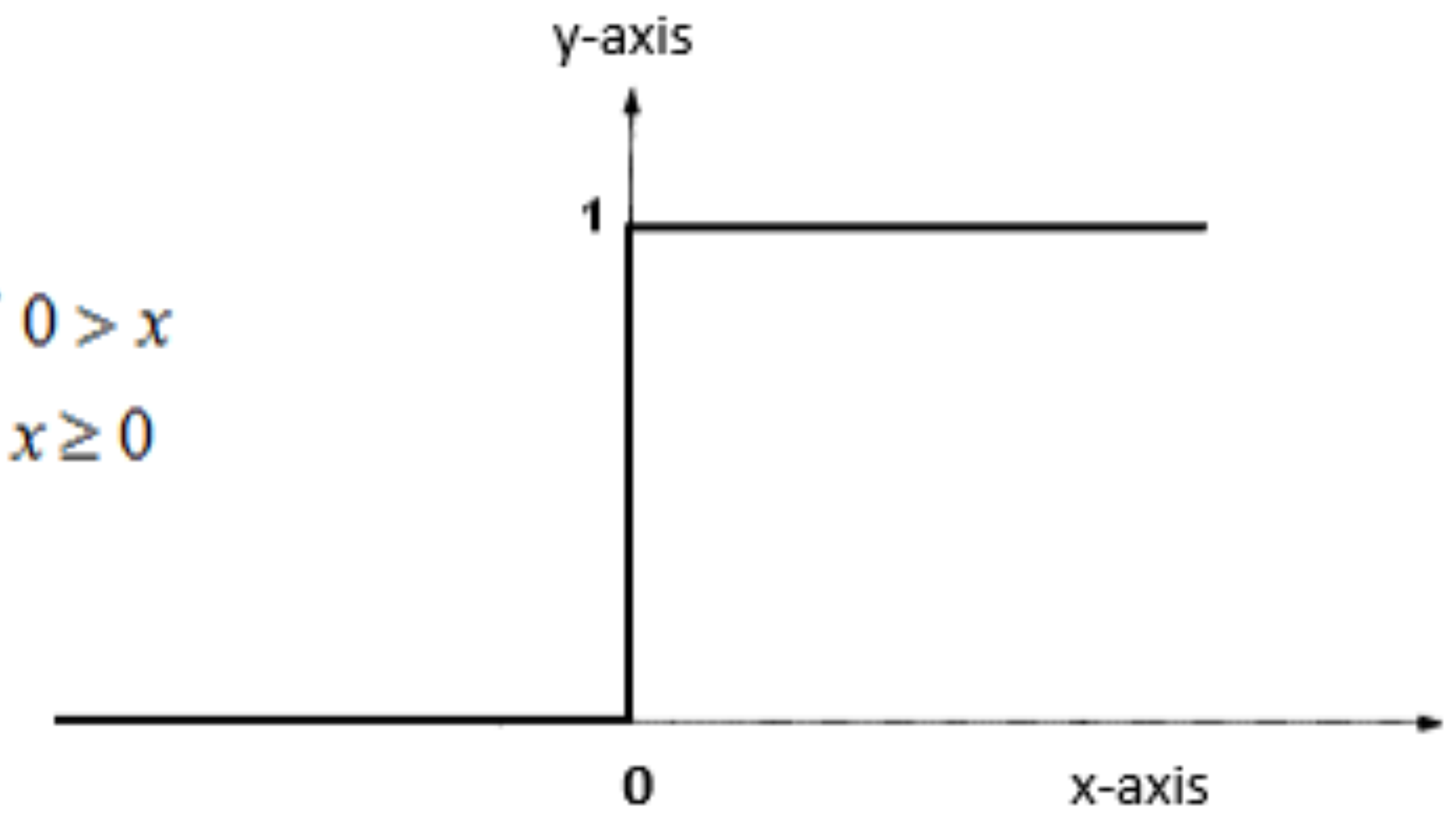


PERCEPTRON

- **Perceptron** is a simple neuron with a simple activation function that aggregates inputs and decides if neuron will fire an electrical signal (1) or not (0)
- Binary output:

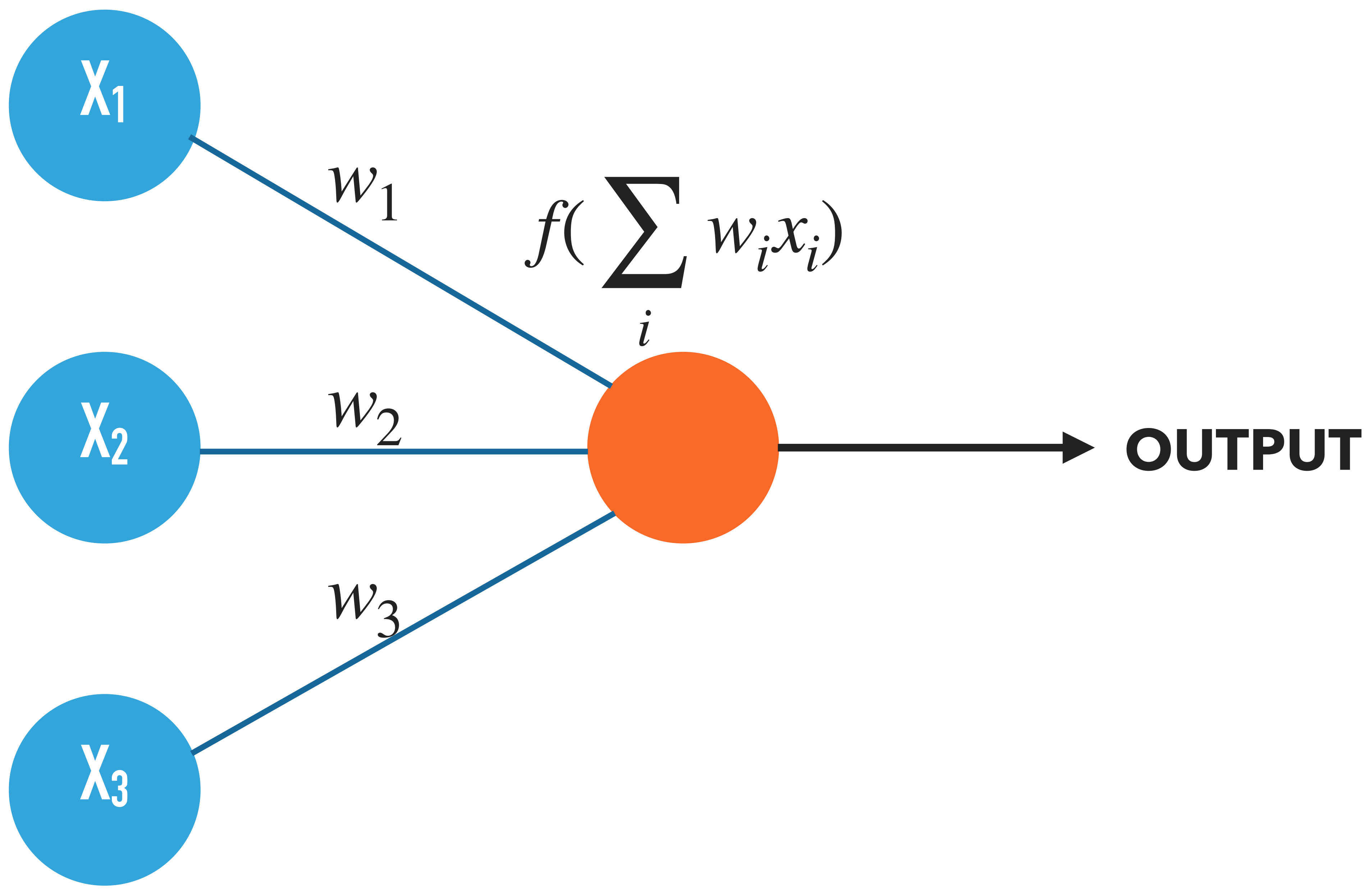


$$f(x) = \begin{cases} 0 \text{ if } 0 > x \\ 1 \text{ if } x \geq 0 \end{cases}$$

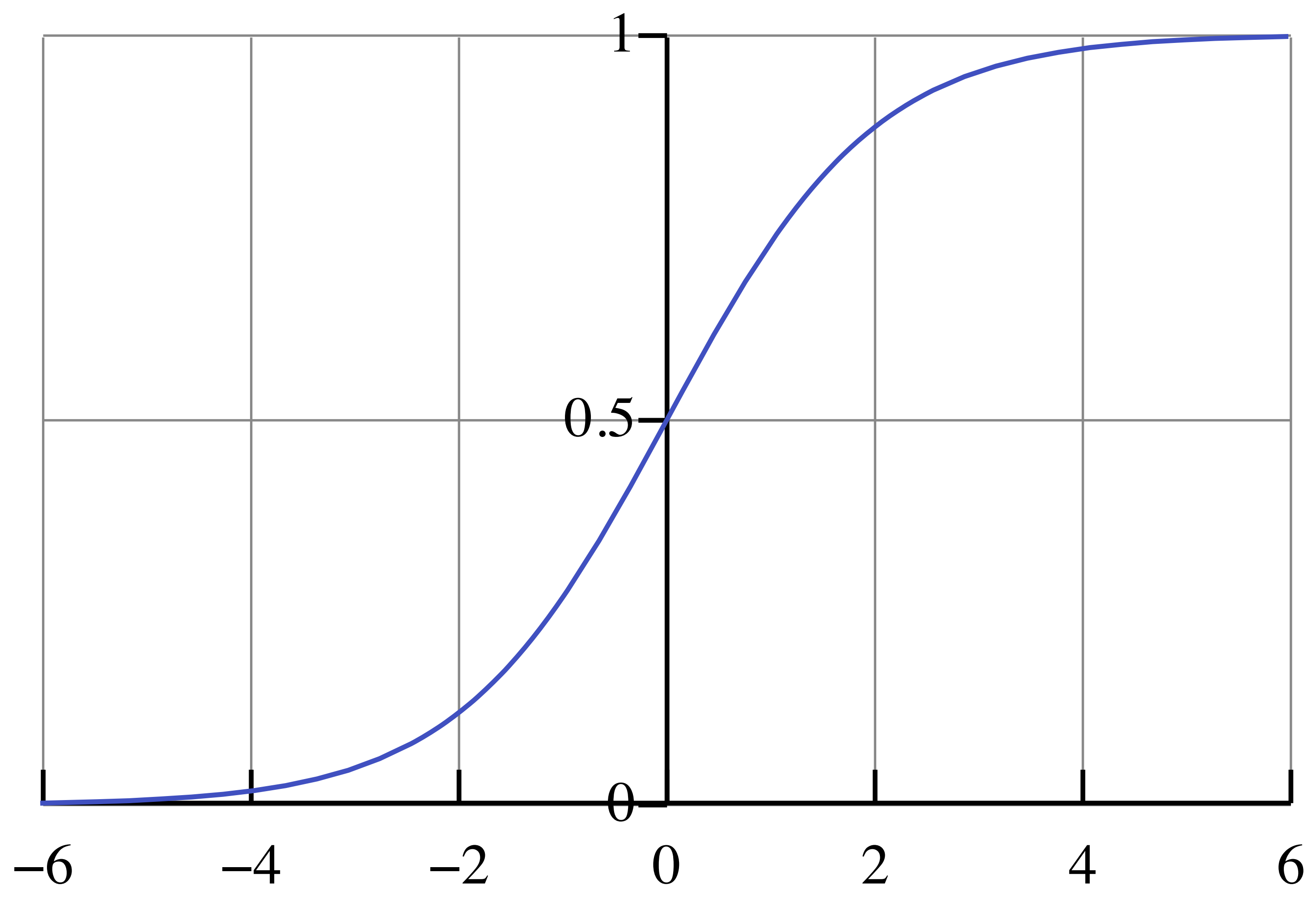


SIGMOID NEURON

- **Sigmoid neuron** uses a sigmoid function that provides a smoother output
  - Small change in weights will result in a small change in output
- Sigmoid functions are a class of functions that resemble shape “S”
- Let’s try to build our first NN with a logistic sigmoid function!



$$f(\sum_i w_i x_i) = \frac{1}{1 + e^{-\sum_i w_i x_i}}$$



BUILDING A NN FROM SCRATCH

- Let's define a very simple training dataset and see if we can build a single-layer single-neuron NN that can learn the model behind it:

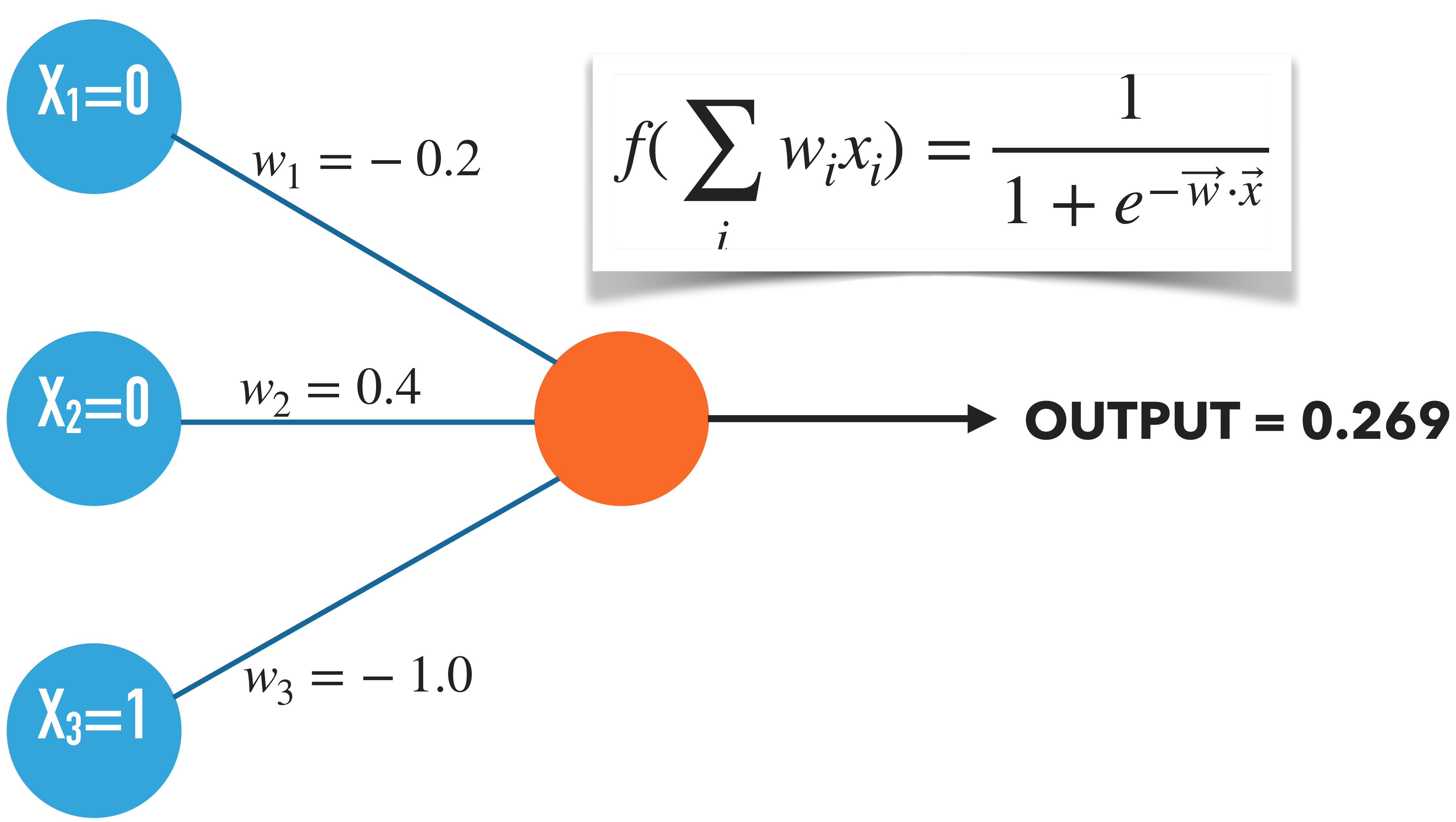
	x_1	x_2	x_3	Y(x)
data 1	0	0	1	0
data 2	1	1	1	1
data 3	1	0	1	0
data 4	0	1	1	1

- Can your NN 🧠 figure out the model Y(X)? 🤔
- Alright, we have the dataset that we will feed to the network and we have chosen our activation function. Let's plug in some numbers and see if we can figure out how to get our NN to learn from input data!

BUILDING A NN FROM SCRATCH

- We need to also choose some starting values for weights
  - why not $\vec{w} = (-0.2, 0.4, -1.0)$
- Plugging in numbers we get:

data 1



$$f\left(\sum_i w_i x_i\right) = \frac{1}{1 + e^{-\vec{w}\cdot\vec{x}}}$$

	x_1	x_2	x_3	Y(x)	Ŷ(x)
data 1	0	0	1	0	0.269
data 2	1	1	1	1	0.310
data 3	1	0	1	0	0.231
data 4	0	1	1	1	0.354

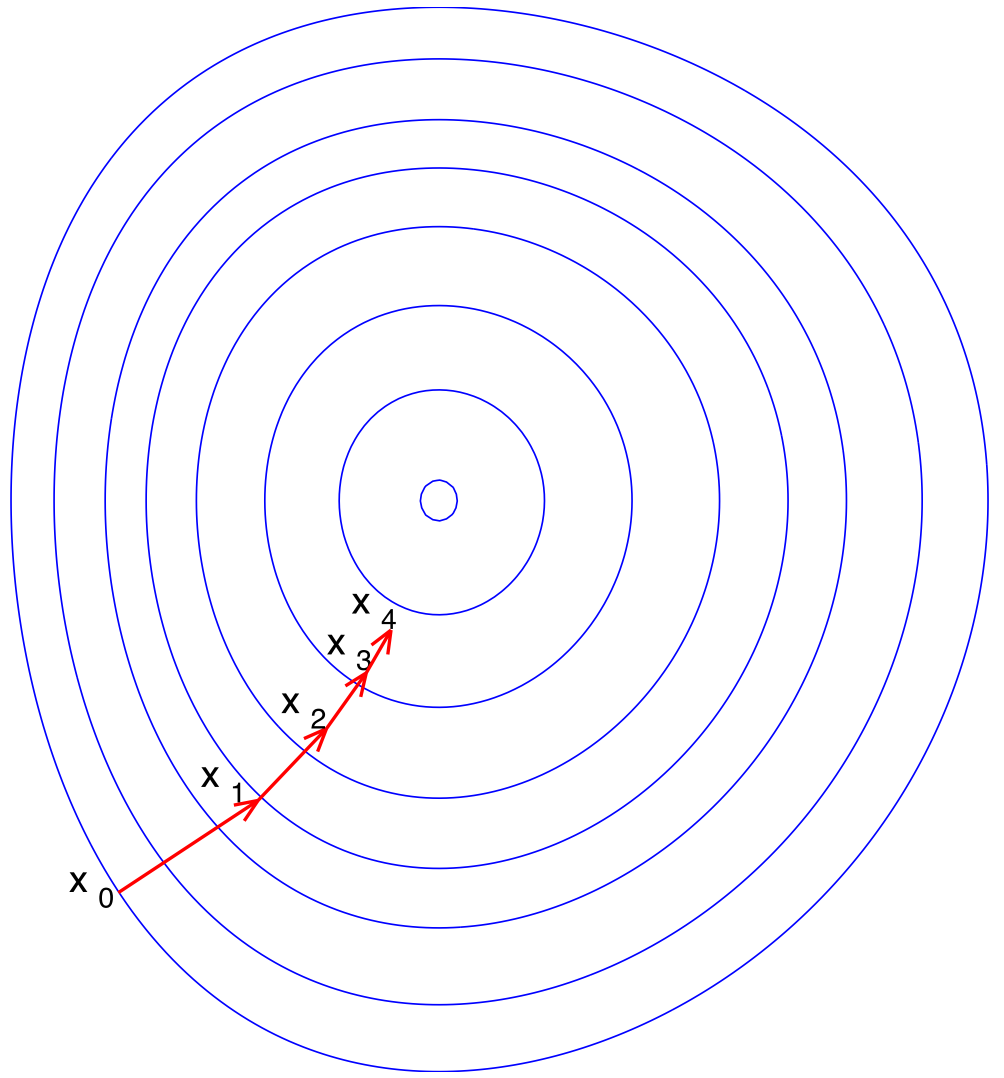
- How would you use this information to train your NN?

BACKPROPAGATION

- I think we all agree on the steps of the basic NN learning algorithm:
  1. Based on all input data and starting weights calculate the predicted output $\hat{y}(\vec{x}, \overrightarrow{w})$
  2. Define some kind of a **loss (cost) function** that is a function of real output values that are known from train data and predicted output $L(y, \hat{y})$
  3. Determine how you can change the weights so that the loss function would decrease in value
  4. Update weights
  5. Repeat steps 1-4 until loss (cost) function can't be reduced anymore
- This procedure is called **backpropagation** because we are propagating information in the opposite direction of the neural network
- [Fun fact] Biological neural backpropagation is proven to exist but its function remains mysterious. 😲

LOSS FUNCTION

- Next step is to define a **loss function** that measures the error of our estimate
  - Loss = measure of misclassification
- Many different possibilities, but let's start with a very simple and intuitive one:
  - $L(\vec{x}, \vec{w}) = \frac{1}{2}\sum_i \left(y_i - \hat{y}(\vec{x}_i, \vec{w}_i)\right)^2$
  - be careful, here index i goes over training data (in our case $i = 1,2,3,4$)
- It is obvious that the training should finish when the loss function is at its minimum
  - The idea is to take repeated steps in the opposite direction of the gradient of the function at the current point, because this is the direction of steepest descent.

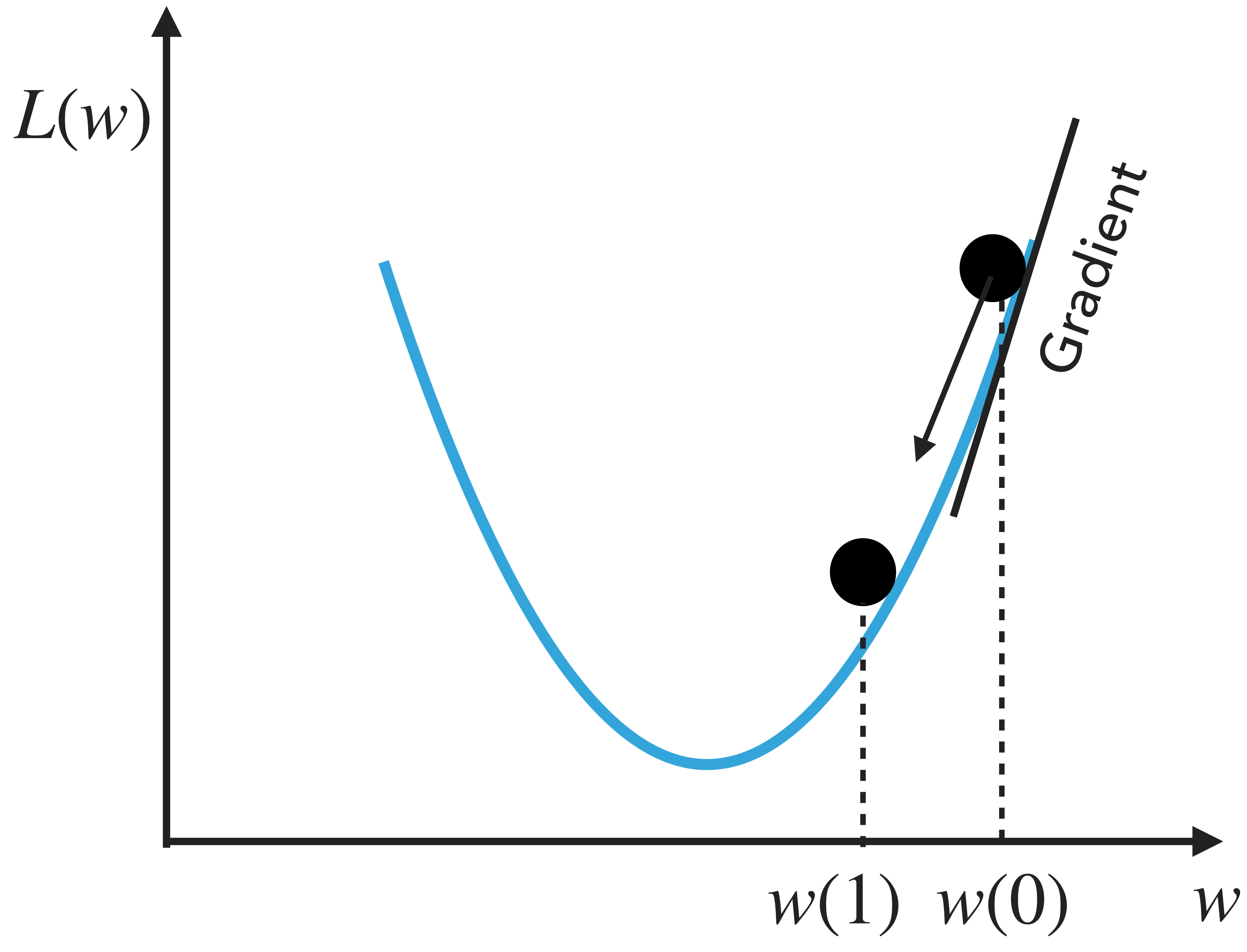


GRADIENT DESCENT

- From definition of a gradient we can conclude:
  - $w_j(t+1) = w_j(t) - \eta \frac{\partial L}{\partial w_j}(t)$
  - where t indicates the index of the **iteration of training** (**epoch**) and η is a free parameter called the **learning rate**
- This is where we realise that in order to apply gradient descent our **activation function** has to be **differentiable**!
- In our case:
  - $L(\overrightarrow{w}) = \frac{1}{2} \sum_i \left(y_i - \hat{y}(\vec{x}_i, \overrightarrow{w}_i)\right)^2; \hat{y}(\vec{x}, \overrightarrow{w}) = \sigma(\vec{x}, \overrightarrow{w}) = \frac{1}{1 + e^{-\overrightarrow{w} \cdot \vec{x}}}$
  - $\frac{\partial L}{\partial w_j} = - \sum_i [y - \sigma(\overrightarrow{w} \cdot \vec{x})] \cdot [\sigma(\overrightarrow{w} \cdot \vec{x})(1 - \sigma(\overrightarrow{w} \cdot \vec{x}))]x_j$

GRADIENT DESCENT

- $\dfrac{\partial L}{\partial w_j} = -\sum_i [y - \sigma(\vec{w} \cdot \vec{x})] \cdot [\sigma(\vec{w} \cdot \vec{x})(1 - \sigma(\vec{w} \cdot \vec{x}))]x_j$
- $w_j(t+1) = w_j(t) - \eta \dfrac{\partial L}{\partial w_j}(t)$
- In our simple example, taking learning rate to be $\eta = 1.0$ at the moment:
  - $\dfrac{\partial L}{\partial w_1}(0) = (0110)\begin{pmatrix} -0.053 \\ 0.148 \\ -0.041 \\ 0148 \end{pmatrix} = 0.106$
  - $w_1(1) = w_1(0) + \eta \cdot 0.106; w_1(1) = -0.094$
  - $\vec{w}(0) = (-0.2, 0.4, -1.0) \;\rightarrow\; \vec{w}(1) = (-0.094, 0.695, -0.799)$
  - $L(\vec{w}(0)) = 0.51 \;\rightarrow\; L(\vec{w}(1)) = 0.38$



BUILDING A NN FROM SCRATCH

- We just trained one epoch of a single-layer single-neuron by hand!
  - It is a great achievement, but now it is time to hire python and speed things up
- We can decide if neuron output is >0.5 the prediction is 1, otherwise it is 0.
- Let’s see what happens after 100 epochs!

```
-bash                                                        ⌥⌘1
(base) Tonis-MacBook-Air:Lectures tsculac$ python SingleNeuron.py
For test data [[1 1 0]] NN prediction is [0.98801299]
(base) Tonis-MacBook-Air:Lectures tsculac$
```

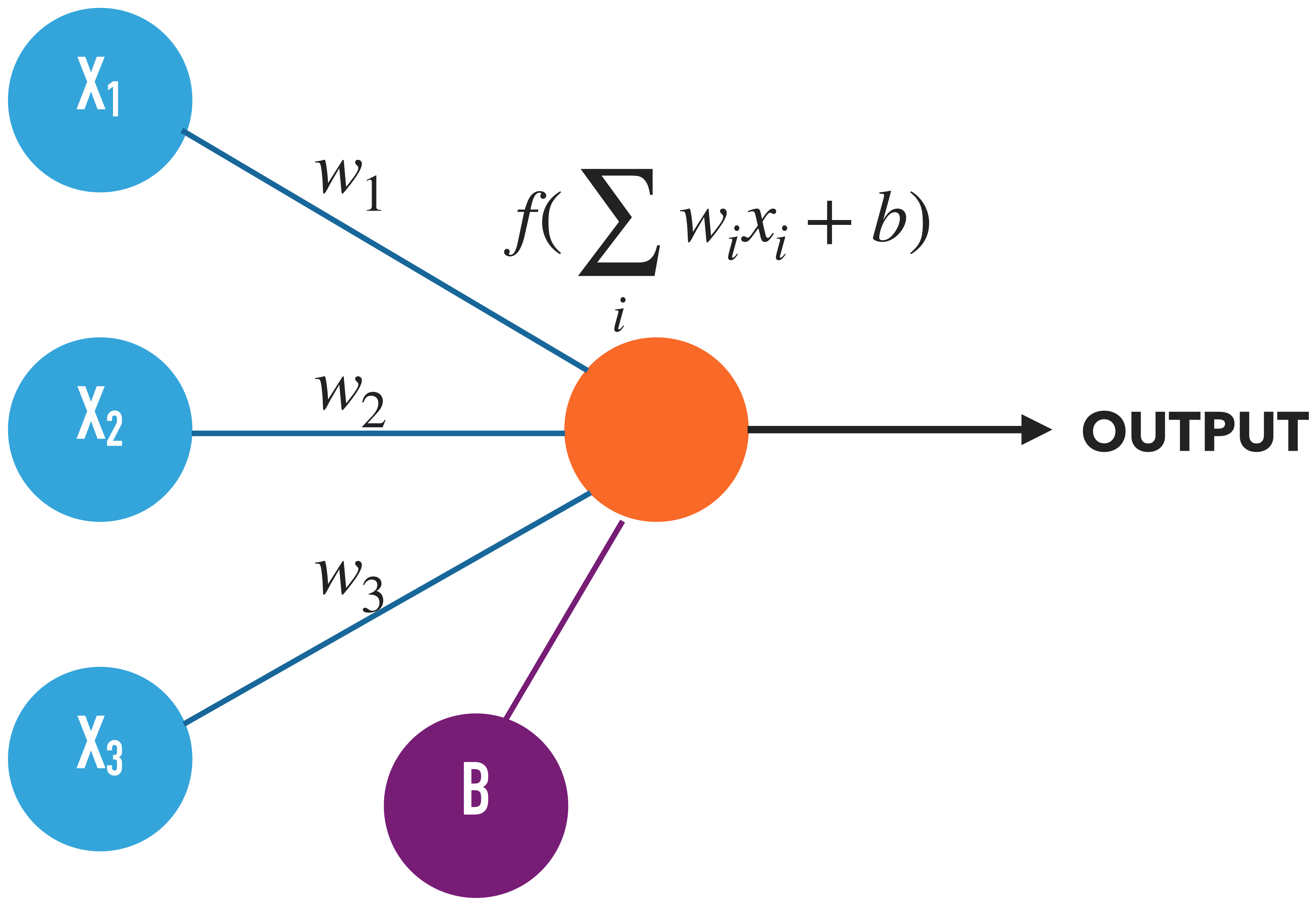
- Success!
- Or is it? How can we make sure the NN learned the correct model?

```
-bash                                                        ⌥⌘1
(base) Tonis-MacBook-Air:Lectures tsculac$ python SingleNeuron.py
For test data [[1 1 0]] NN prediction is [0.98801299]
(base) Tonis-MacBook-Air:Lectures tsculac$ python SingleNeuron.py
For test data [[0 0 0]] NN prediction is [0.5]
(base) Tonis-MacBook-Air:Lectures tsculac$
```

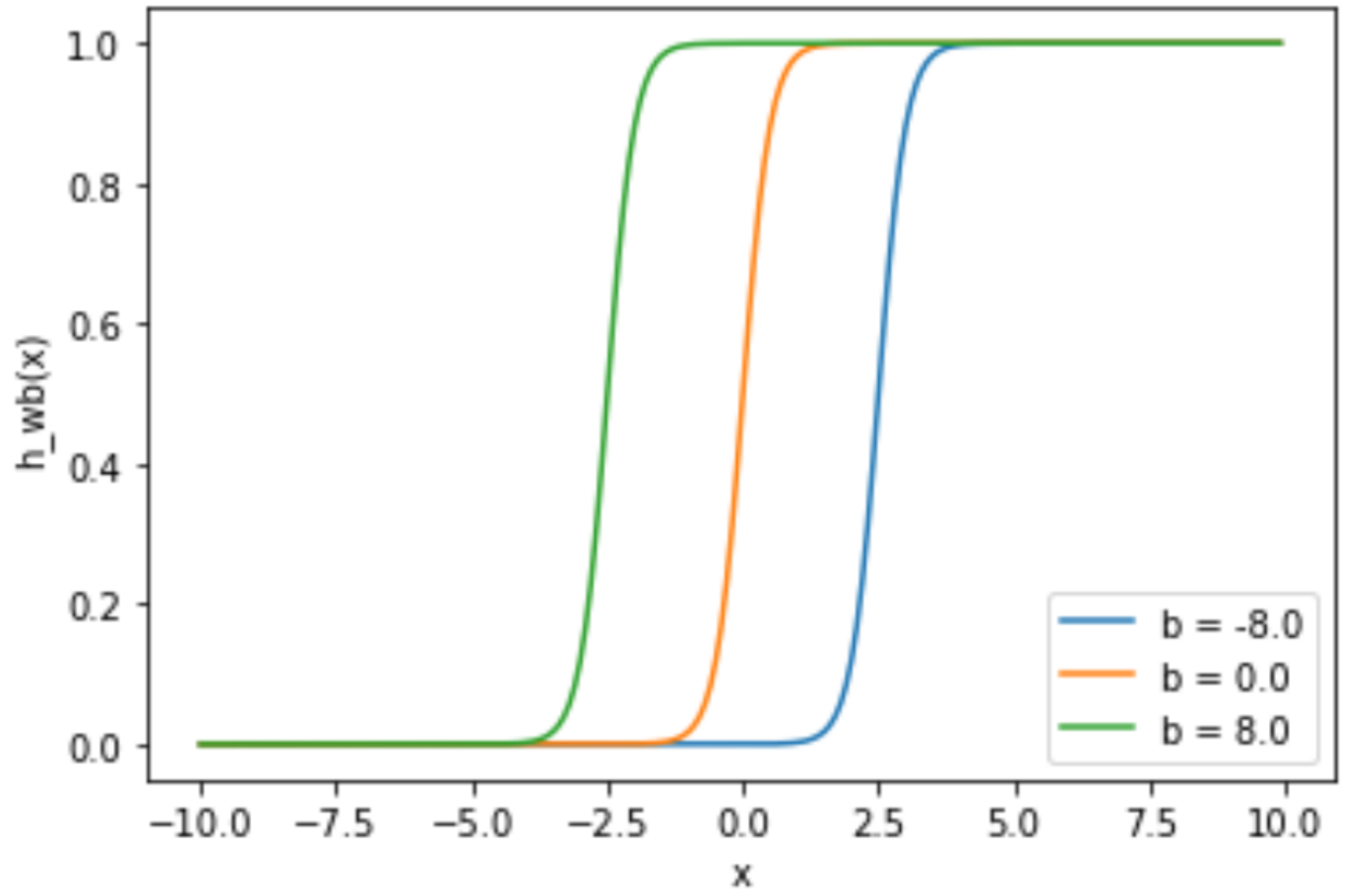
- What happened? How can we fix the problem?

INTRODUCING BIAS

- Having only 0 as input is tricky, because our NN gets stuck and it can never move
- This motivates the introduction of a bias term as one input node to our neuron



$$f(\sum_i w_i x_i + b) = \frac{1}{1 + e^{-\sum_i w_i x_i + b}}$$



- We want to guess initial bias and later update it in training

GRADIENT DESCENT WITH BIAS

- First we need to update the loss function:
  - $L(\overrightarrow{w}, b) = \frac{1}{2}\sum_i \left(y_i - \hat{y}(\vec{x}_i, \overrightarrow{w}_i, b)\right)^2; \hat{y}(\vec{x}, \overrightarrow{w}, b) = \sigma(\vec{x}, \overrightarrow{w}, b) = \frac{1}{1 + e^{-\overrightarrow{w}\cdot\vec{x}+b}}$
  - $\frac{\partial L}{\partial w_j} = -\sum_i [y - \sigma(\overrightarrow{w} \cdot \vec{x} + b)] \cdot [\sigma(\overrightarrow{w} \cdot \vec{x} + b)(1 - \sigma(\overrightarrow{w} \cdot \vec{x} + b))]x_j$
  - $\frac{\partial L}{\partial b} = -\sum_i [y - \sigma(\overrightarrow{w} \cdot \vec{x} + b)] \cdot [\sigma(\overrightarrow{w} \cdot \vec{x} + b)(1 - \sigma(\overrightarrow{w} \cdot \vec{x} + b))]$
  - $w_j(t+1) = w_j(t) - \eta\frac{\partial L}{\partial w_j}(t)$
  - $b(t+1) = b(t) - \eta\frac{\partial L}{\partial b}(t)$
- Time to go back and retrain our NN with bias!

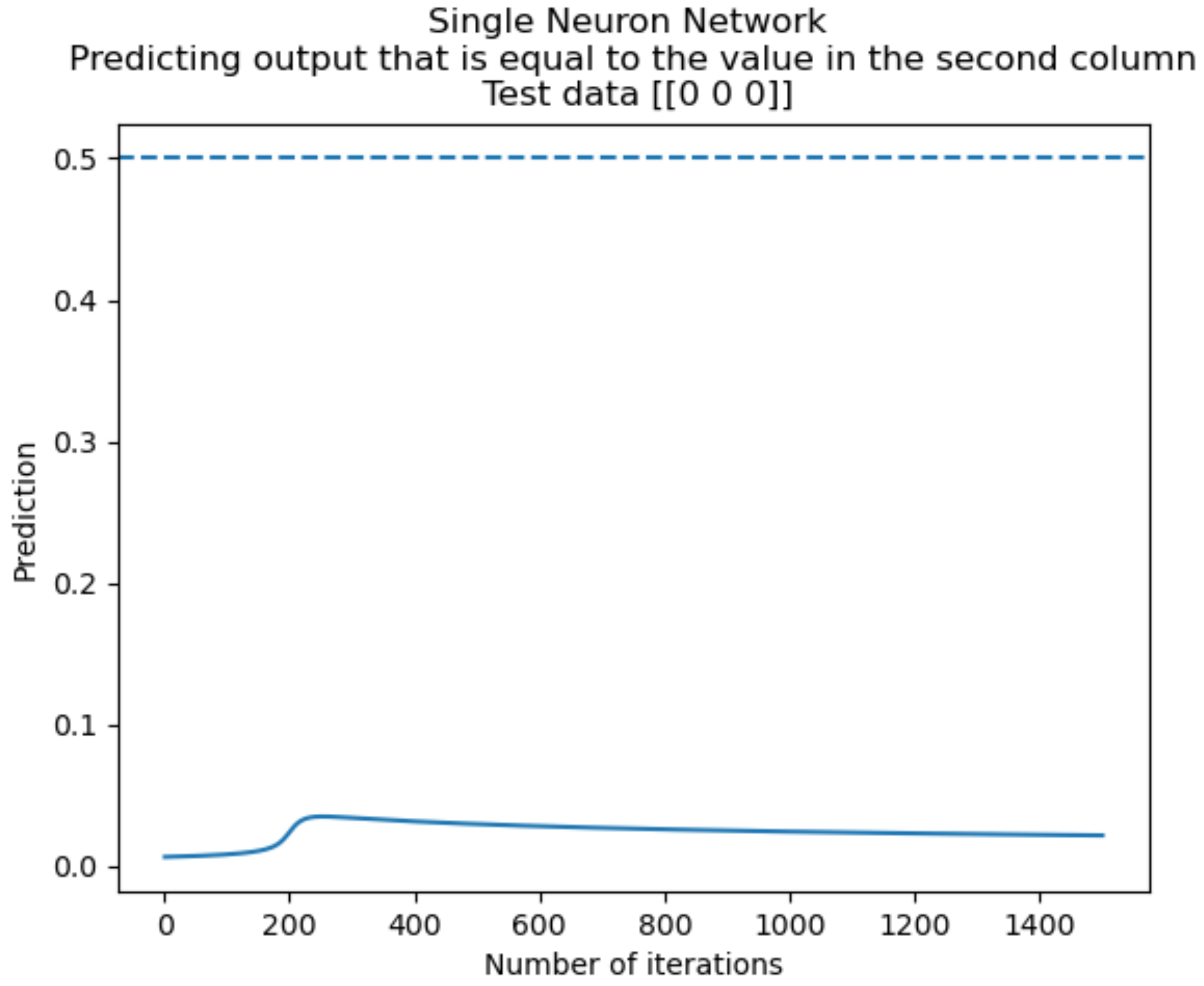
BUILDING A NN FROM SCRATCH

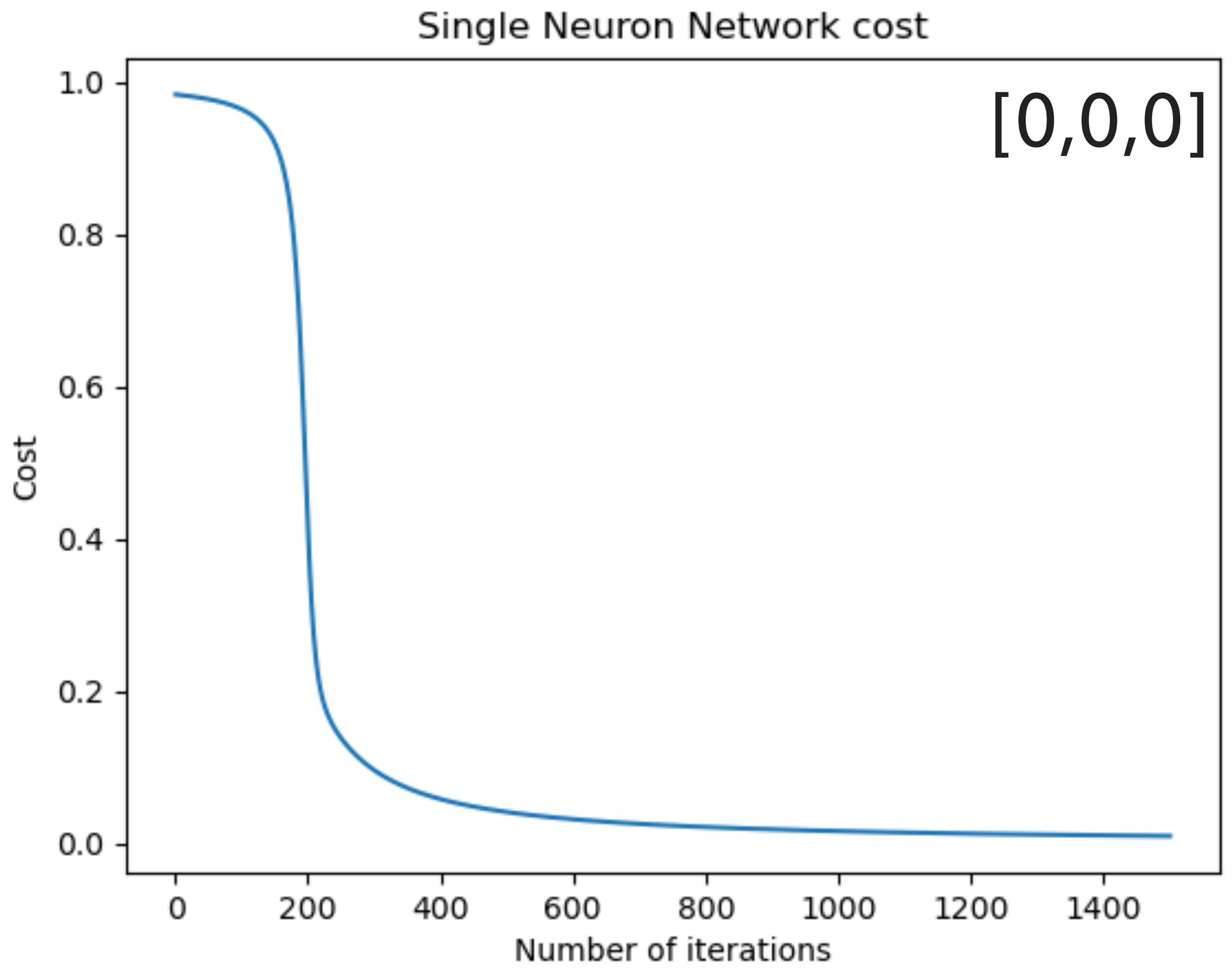
- Let’s define NN parameters:
  - N_epoch = 100, $\vec{w}(0) = (-0.2, 0.4, -1.0)$, b(0)=-5, $\eta = 0.1$

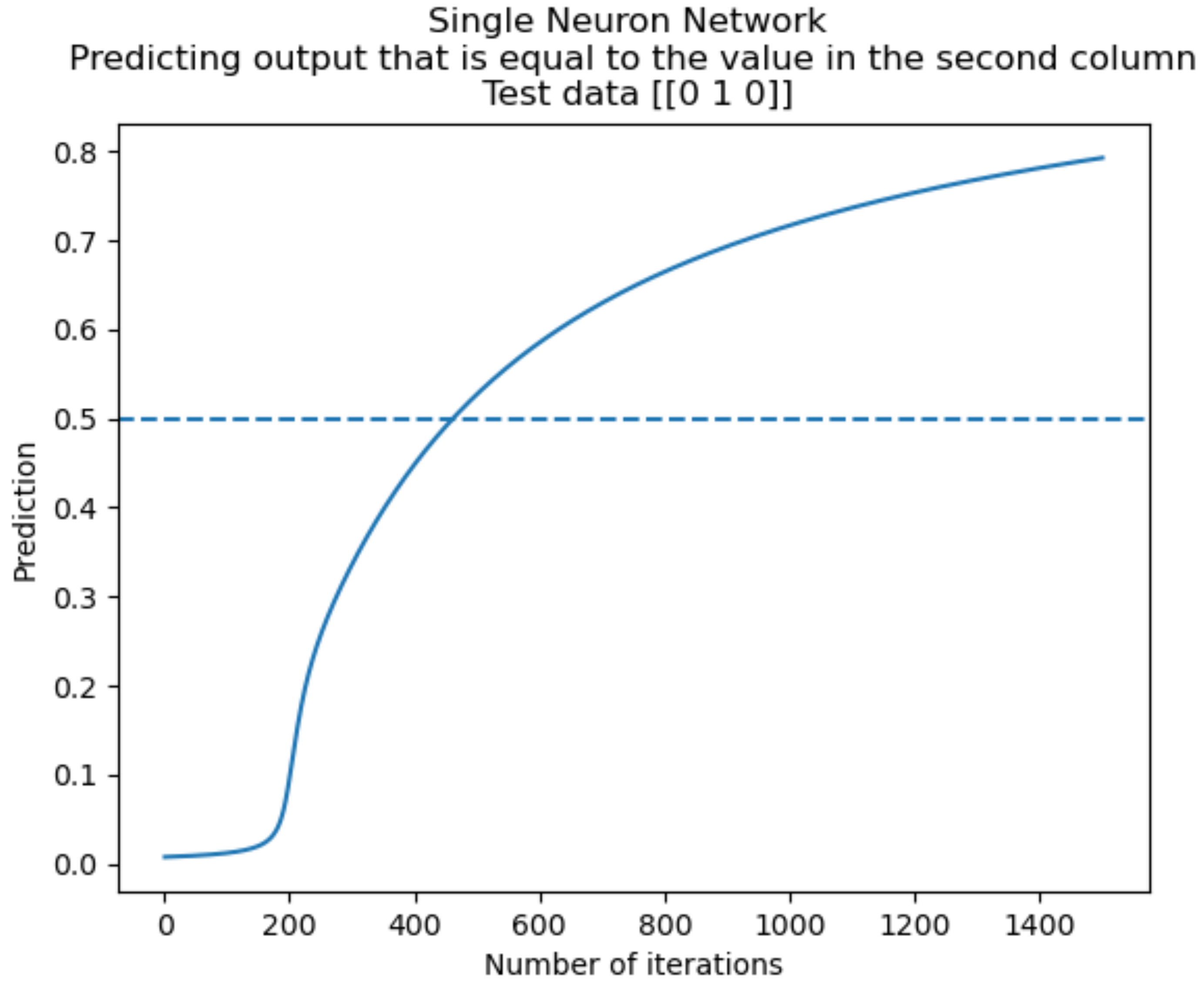
```
-bash
(base) Tonis-MacBook-Air:Lectures tsculac$ python SingleNeuron.py
For test data [[0 0 0]] NN prediction is [0.00720712]
(base) Tonis-MacBook-Air:Lectures tsculac$
```

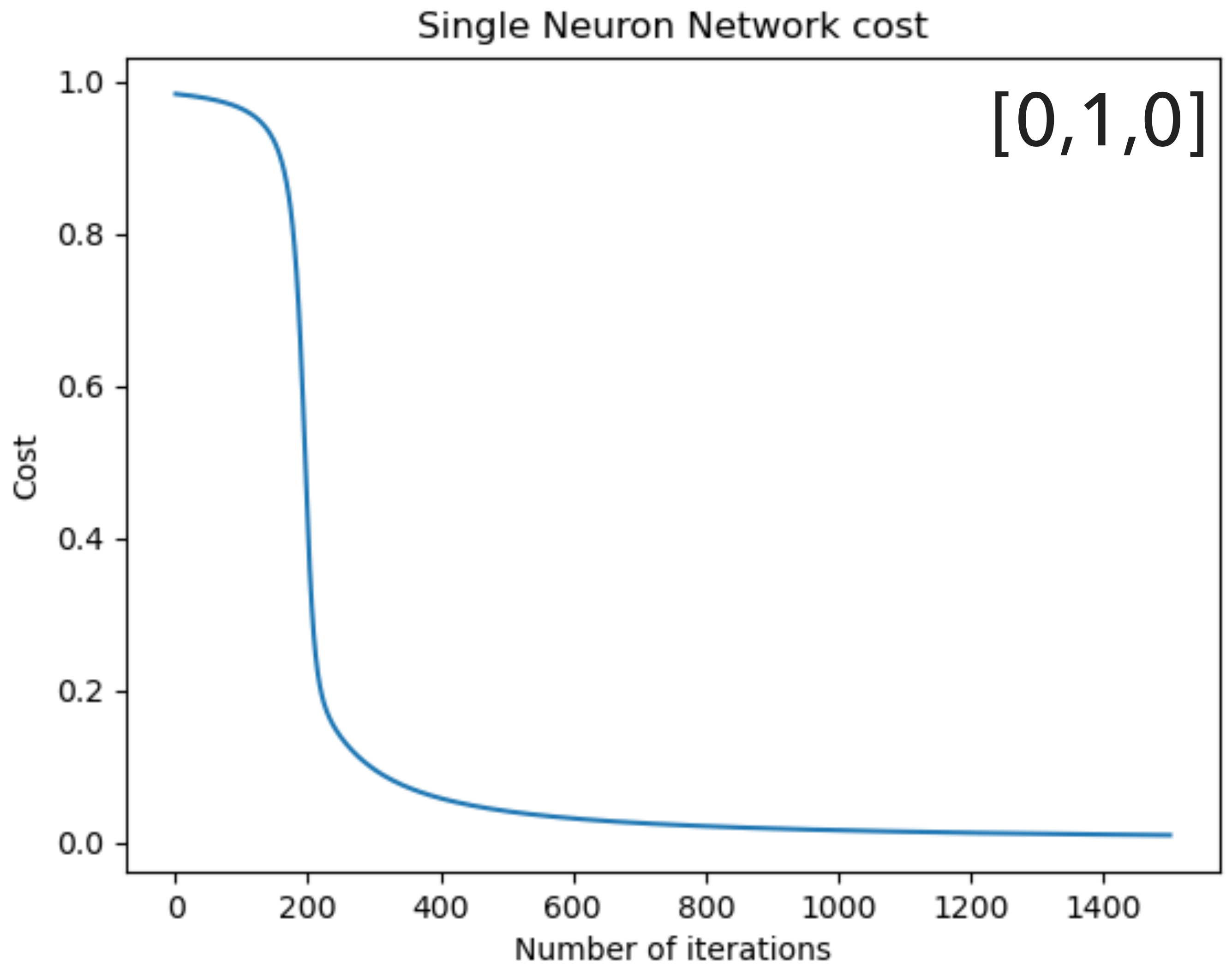
- Success!
- Or is it? How can we make sure the NN learned the correct model?
  - No! We can’t just test it on 1 example and call it a day!
  - We need to make a thorough study on more test data
  - We need to see what is happening to the loss/cost function as well!
- In our case, we have 4 test data points that are not part of training:
  - [0,0,0], [0,1,0], [1,0,0], [1,1,0]

TESTING A NN

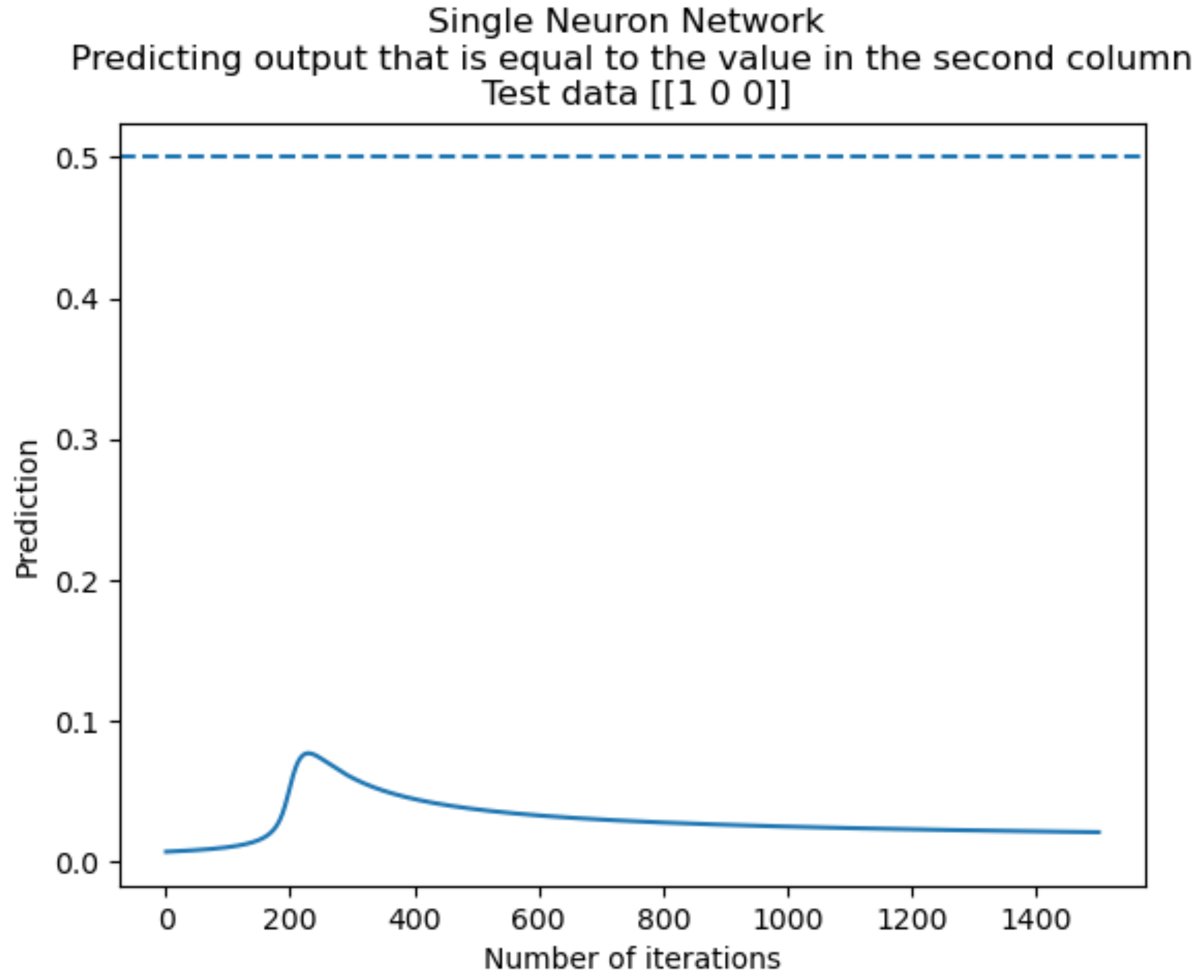

Single Neuron Network
Predicting output that is equal to the value in the second column
Test data [[0 0 0]]
Prediction
Number of iterations

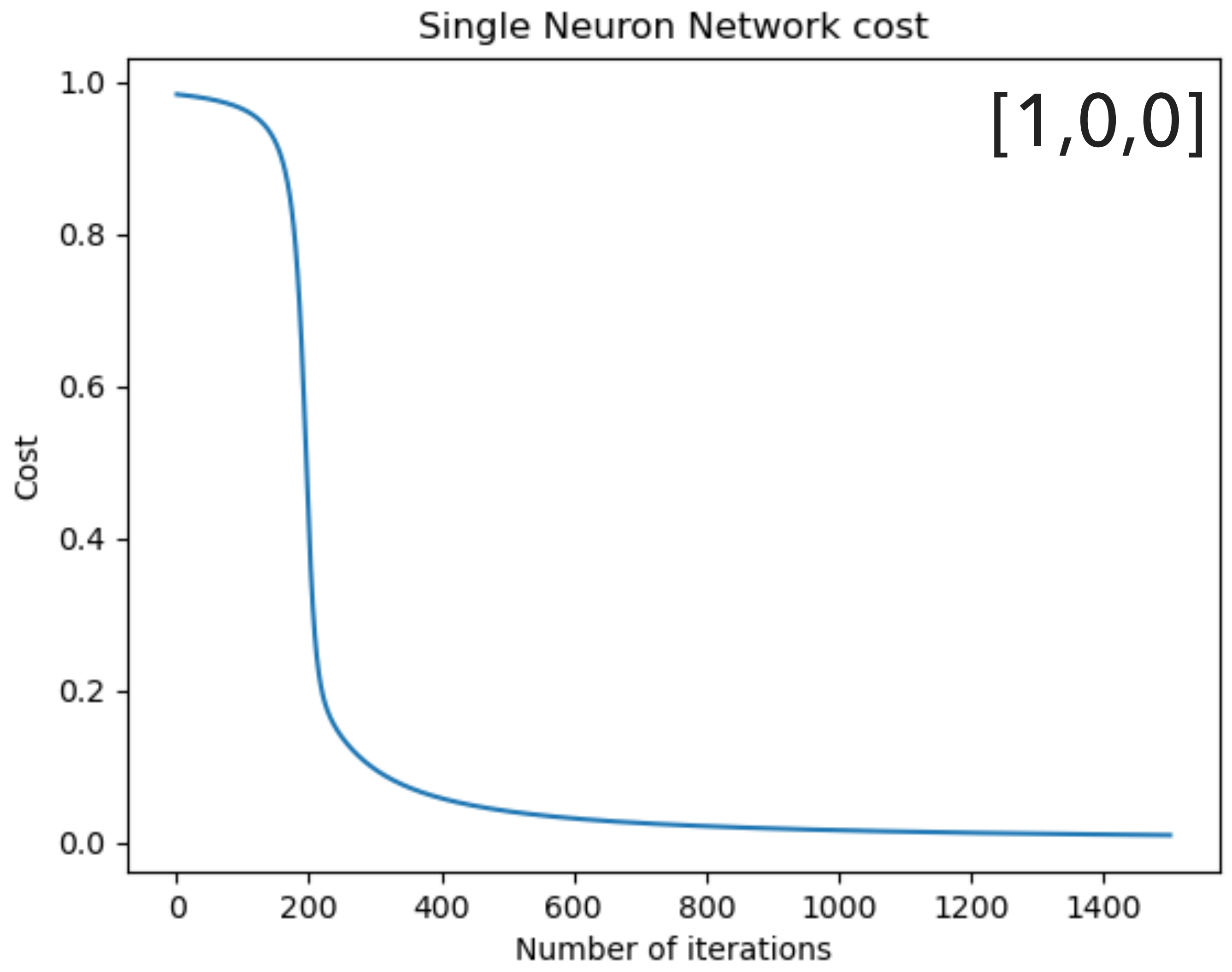

Single Neuron Network cost
[0,0,0]
Cost
Number of iterations

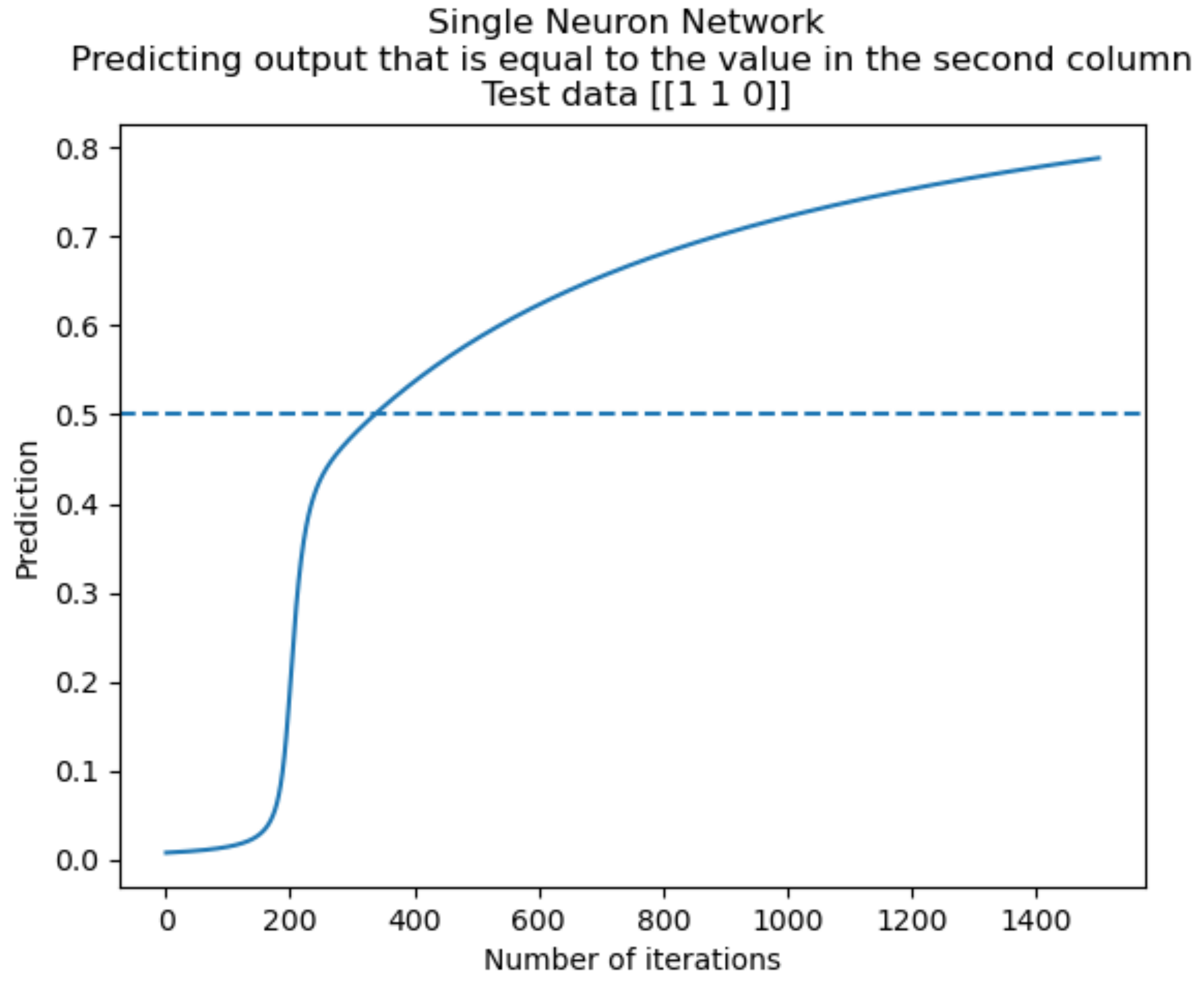

Single Neuron Network
Predicting output that is equal to the value in the second column
Test data [[0 1 0]]
Prediction
Number of iterations

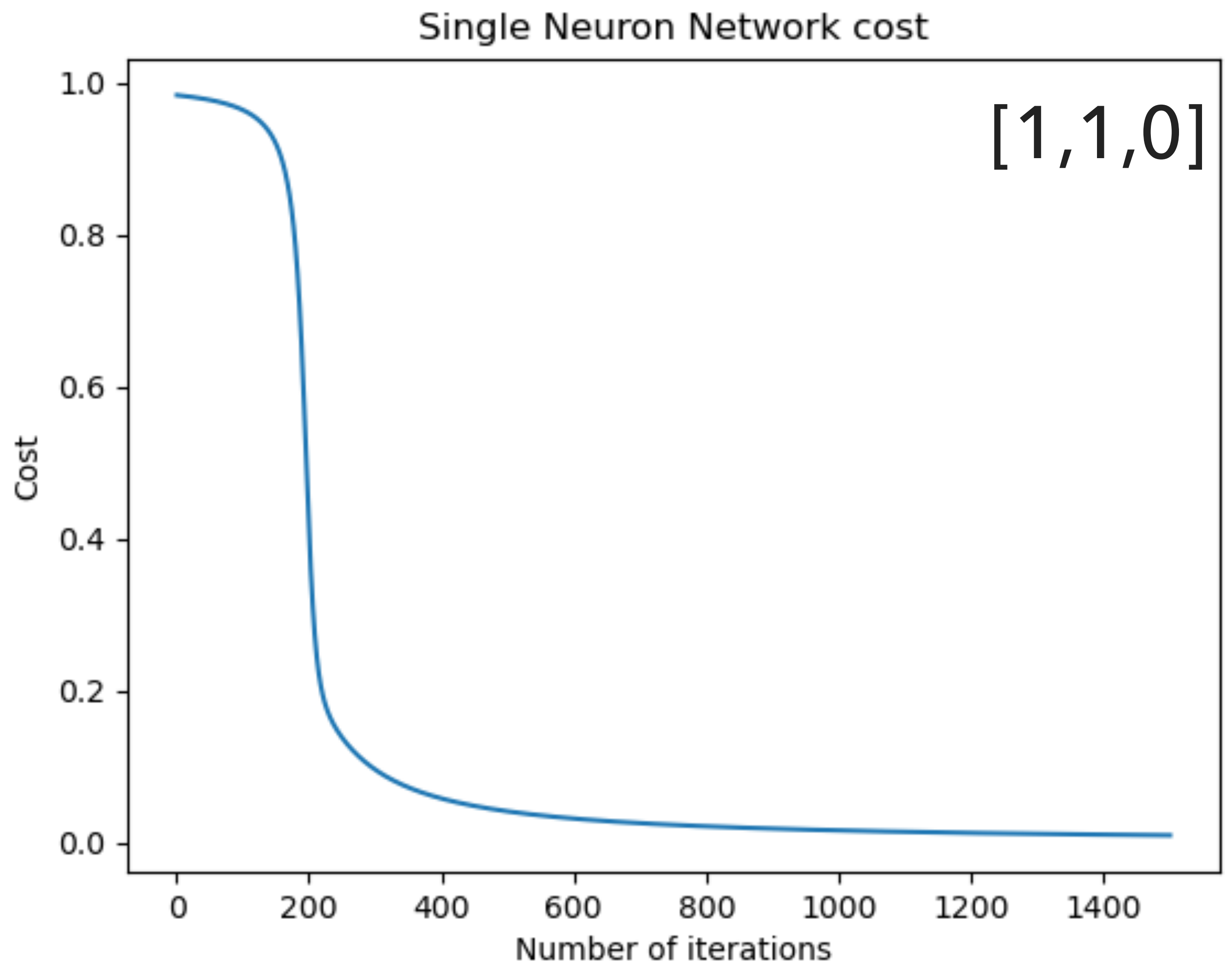

Single Neuron Network cost
[0,1,0]
Cost
Number of iterations

TESTING A NN


Single Neuron Network
Predicting output that is equal to the value in the second column
Test data [[1 0 0]]
Prediction
0.0
0.1
0.2
0.3
0.4
0.5
0
200
400
600
800
1000
1200
1400
Number of iterations

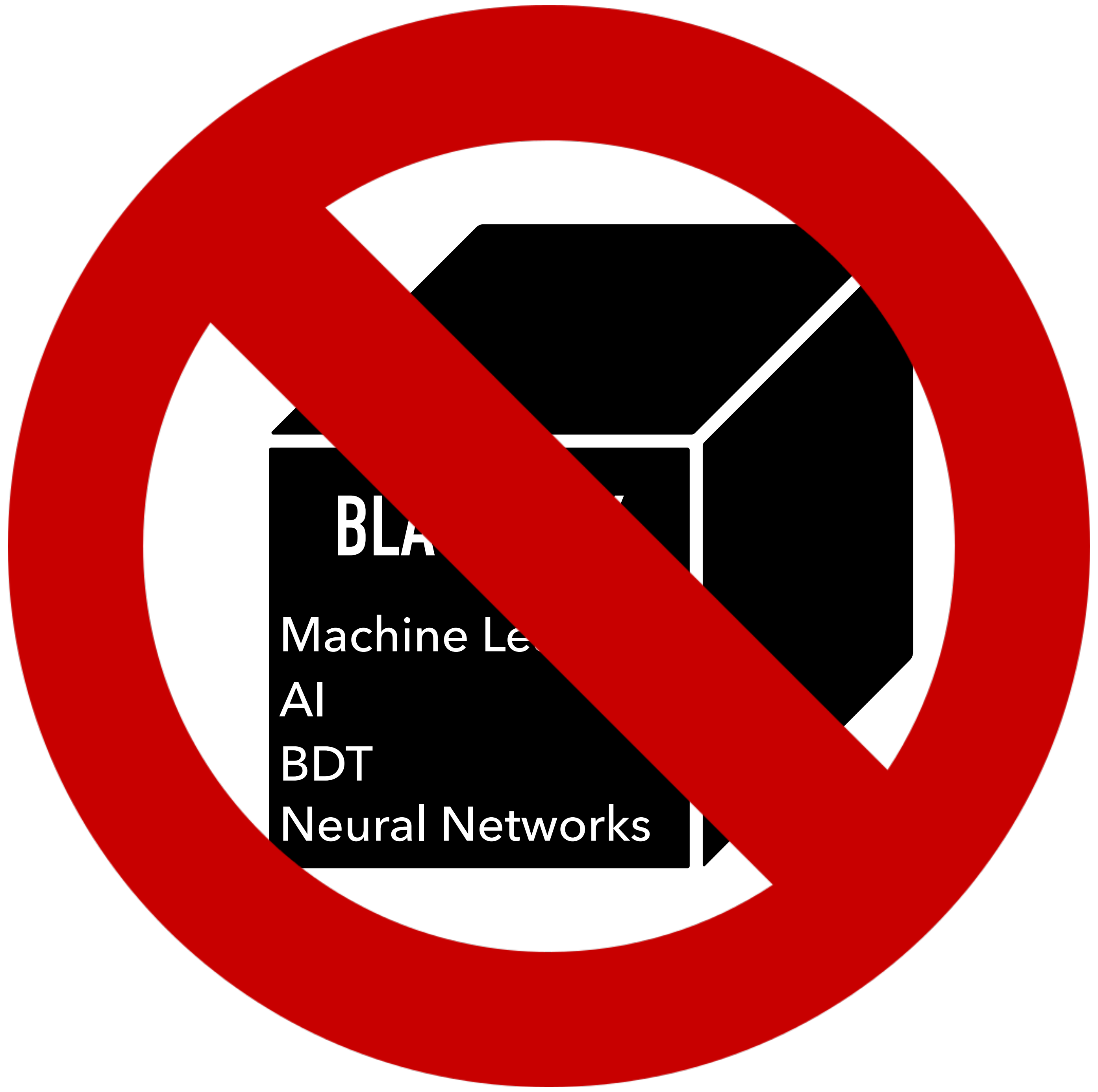

Single Neuron Network cost
[1,0,0]
Cost
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000
1200
1400
Number of iterations


Single Neuron Network
Predicting output that is equal to the value in the second column
Test data [[1 1 0]]
Prediction
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
200
400
600
800
1000
1200
1400
Number of iterations


Single Neuron Network cost
[1,1,0]
Cost
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000
1200
1400
Number of iterations

SIMPLE NN OVERVIEW

- Building a simple NN from scratch helps us understand key concepts:
  - Input nodes, layers, neurons, outputs
  - Activation functions
  - Backpropagation
  - Loss/cost functions
  - Gradient descent
  - Bias input nodes, learning rate, epochs
  - Training
  - Testing

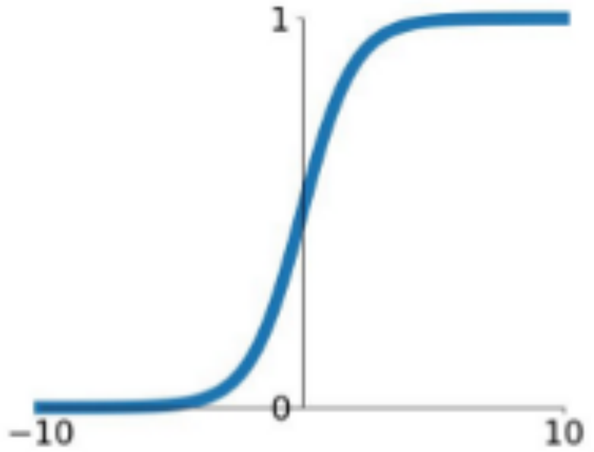


- We will continue by coming back to some of the above mentioned concepts and discussing how they affect NN and different approaches
- This will give us a deep understanding to be able to move to deep learning

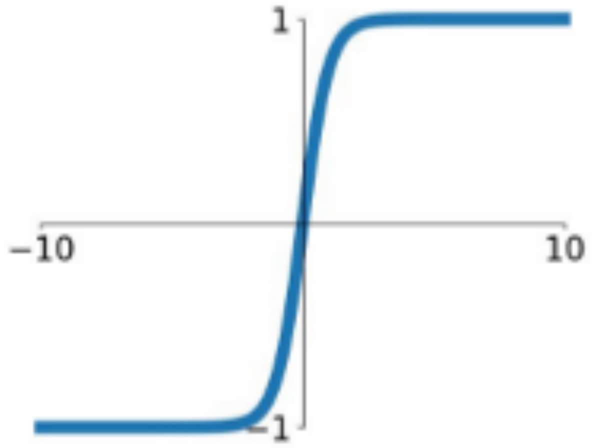
ACTIVATION FUNCTIONS

- It determines at what threshold the neuron will fire or the frequency at which a neuron fires
- If we want to apply Gradient descent it needs to be differentiable
- Some common choices that we didn't mention:

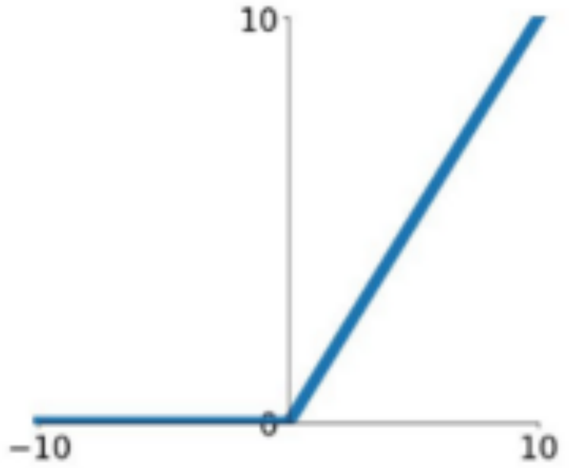
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



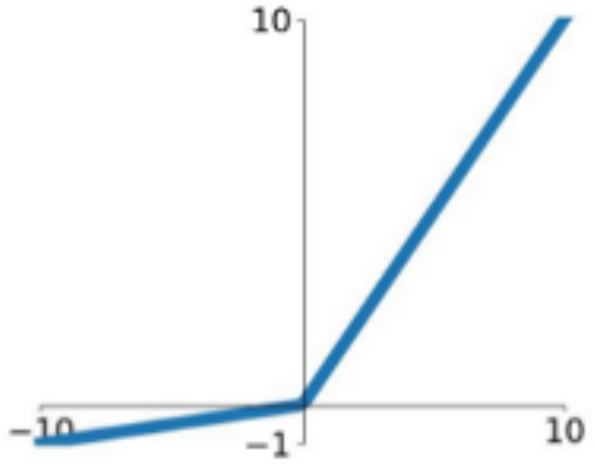
tanh
$\tanh(x)$



ReLU
$\max(0, x)$

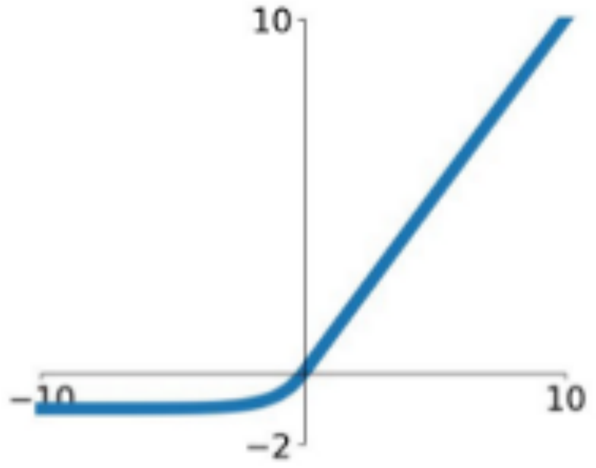


Leaky ReLU
$\max(0.1x, x)$



Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



*LU = Linear Unit (commonly used for for regression)

UNIVERSAL APPROXIMATION THEOREM

- Neural Networks are extremely powerful because of it
- Let σ be any continuous function. The finite sums of the form

$$G(x) = \sum_{i=1}^{N} \alpha_i \sigma(w_i^T x + b_i)\,,\ \ w_i \in \mathbb{R}^N,\ \alpha_i, b_i \in \mathbb{R} \text{ are dense in } C(\mathbb{R}, \mathbb{R}).$$

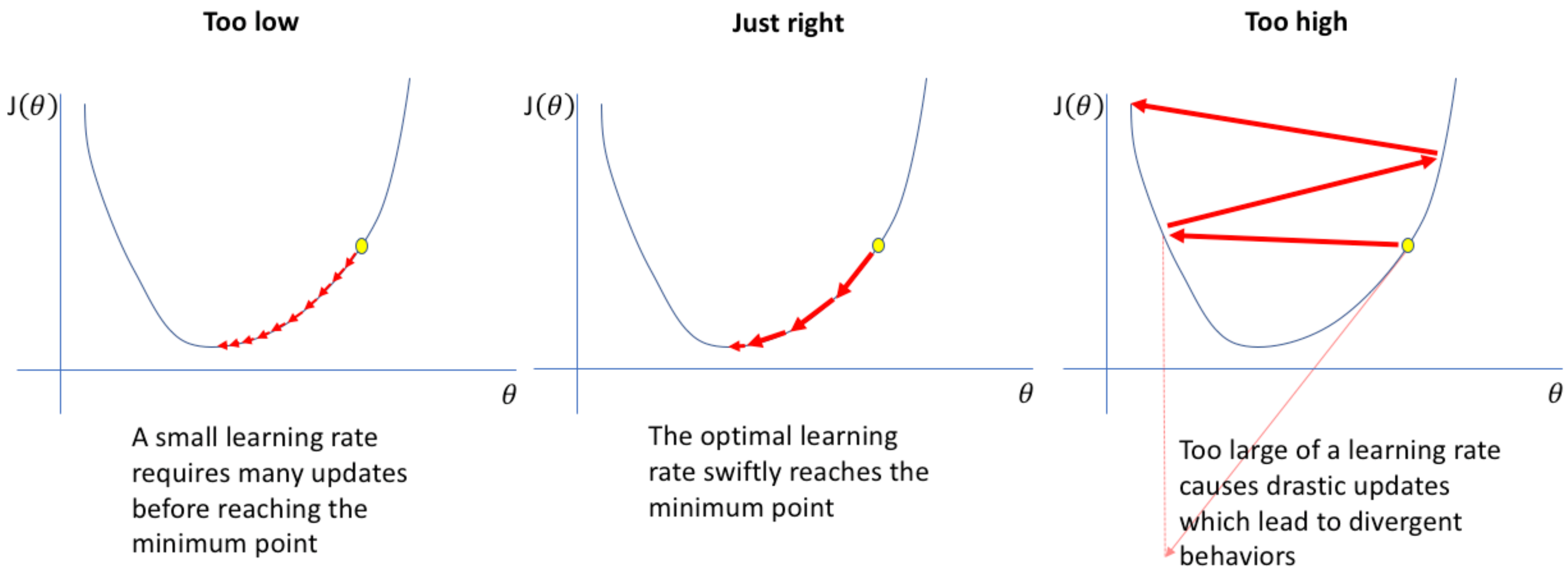
  - In other words, given any $\epsilon > 0$ and $f \in C(\mathbb{R}, \mathbb{R})$, there is a sum $G(x)$ of the above form such that $|G(x) - f(x)| < \epsilon, \forall x \in \mathbb{R}^m$
  - The theorem states that for a given function, if there are enough neurons in a NN, then there exists a neural network with that many neurons that does approximate function f to within ϵ
- We solved the problem!
- No :(Theorem does not provide any way to actually find such a sequence.
  - It also doesn't guarantee any method, such as backpropagation, might actually find such a sequence

GRADIENT DESCENT

- Another commonly used **loss function** is cross-entropy

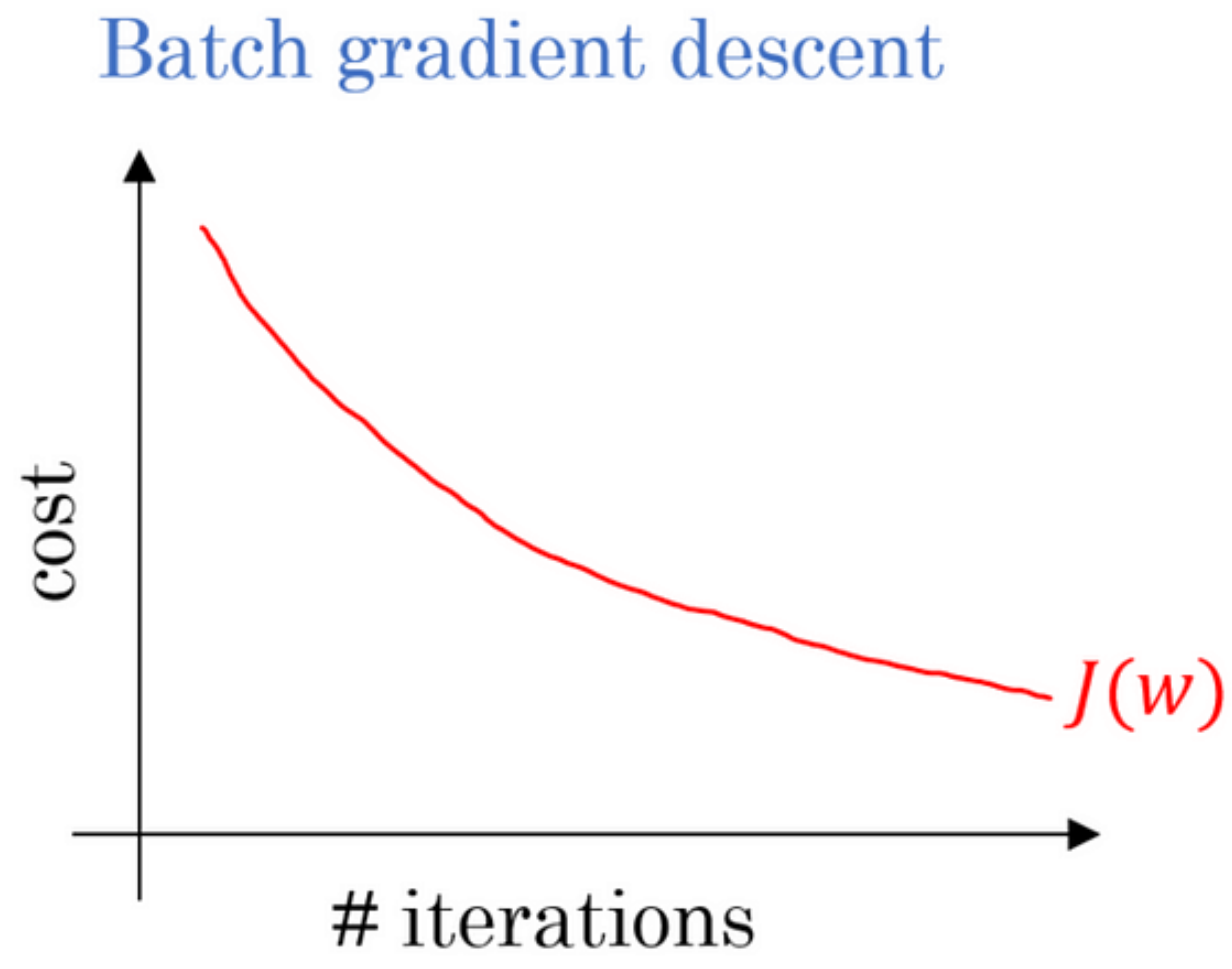
$$L(w,b) = -\sum_k \sum_i y_{ik} \log \hat{y}_{ik}$$

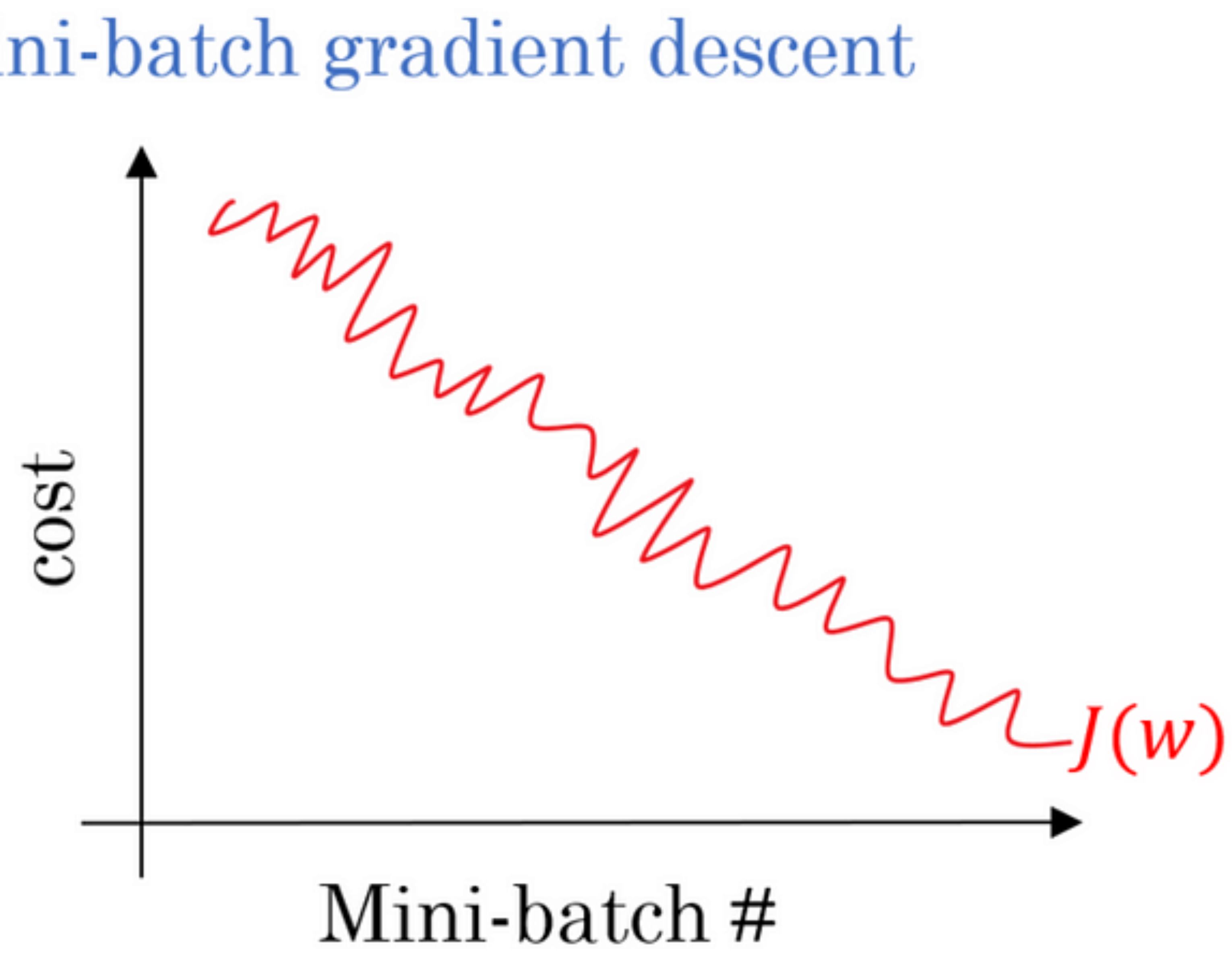
- Choice of learning rate can determine if we get stuck in a local minima or overshoot global minima
  - Possible to update (reduce) learning rate as we approach minima



STOCHASTIC GRADIENT DESCENT

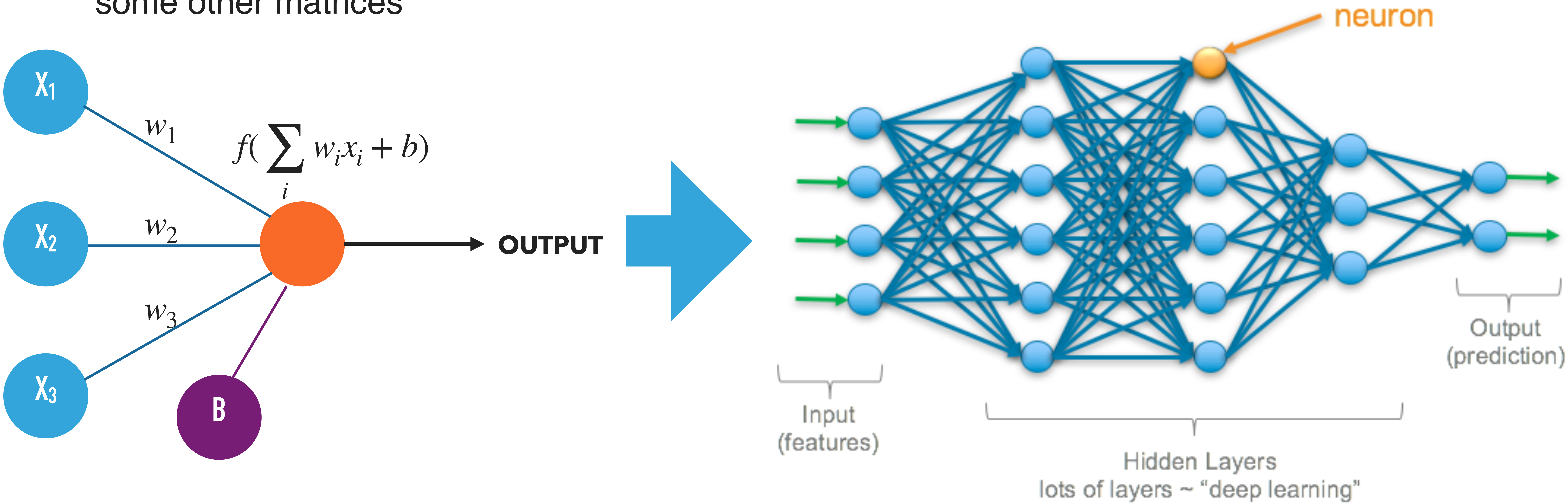
- So far we have updated weights averaged over all training cases and repeated this for N epochs
  - this is called **batch gradient descent** and results in smooth cost vs epoch graph
  - Deep learning uses huge amounts of data so this approach can become extremely slow
- Idea of **stochastic gradient descent** is to update weights after each event
- **Mini-batch gradient descent** is a combination of 2 that is achieved by splitting the training dataset into several smaller ones





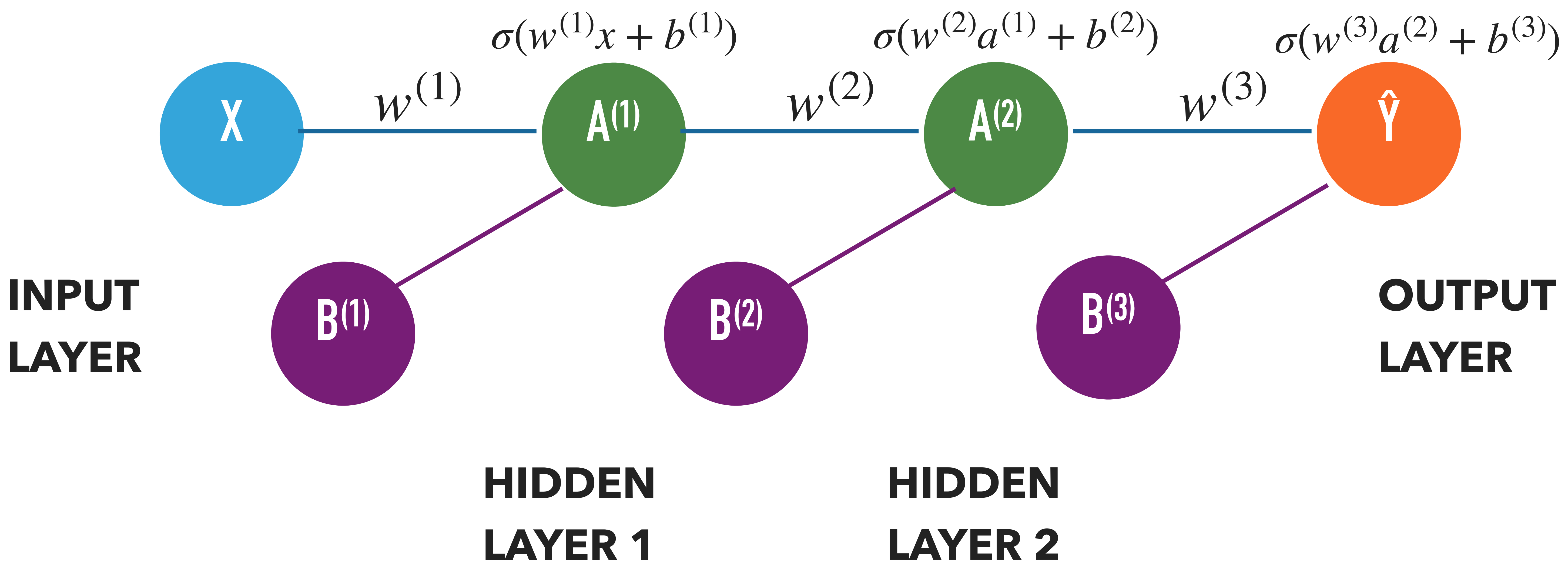
DEEP NEURAL NETWORK

- Now that we have mastered a single-neuron single-layer NN it is time to move to deep NNs
- “Deep” simply means a lot of hidden layers (and neurons) :)
    - In math language we will move from simple functions that map a couple of real numbers to a single number to extremely complicated functions that map very high-dimensional matrices to some other matrices



ADDING HIDDEN LAYERS

- Let's again start very simple by investigating addition of hidden layers:



- Key element for backpropagation is to calculate the cost/loss function
- We again start simple with cost function of a single training event 0:
  - $C_0(w^{(1)}, b^{(1)}, w^{(2)}, b^{(2)}, w^{(3)}, b^{(3)}) = (a^{(3)} - y)^2$
  - $a^{(3)} = \sigma(z^{(L)})\,,\ z^{(3)} = w^{(3)}a^{(2)} + b^{(3)}\,,\ a^{(0)} = x$

HIDDEN LAYERS MATH

- We want to understand how a small change in $w^{(3)}$ affects C_0?
- Calculus tells us we can figure it out by applying the well known "chain rule"
  - $$\frac{\partial C_0}{\partial w^{(3)}} = \frac{\partial C_0}{\partial a^{(3)}}\frac{\partial a^{(3)}}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial w^{(3)}}$$
  - $$\frac{\partial C_0}{\partial a^{(3)}} = 2(a^{(L)} - y)\ ,\ \frac{\partial a^{(3)}}{\partial z^{(3)}} = \sigma'(z(3))\ ,\ \frac{\partial z^{(3)}}{\partial w^{(3)}} = a^{(2)}$$
- C_0 is the cost function of a single training event. To move to all N events we simply average over all events:
  - $$\frac{\partial C}{\partial w^{(3)}} = \frac{1}{N}\sum_{k=0}^{N-1}\frac{\partial C_k}{\partial w^{(3)}}$$

HIDDEN LAYERS MATH

- To apply gradient descent we need the gradient of the cost function:
- $$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix} , \quad \frac{\partial C_0}{\partial w^{(L)}} = 2a^{(L-1)}\sigma'(z^{(L)})(a^{(L)} - y) \, , \quad \frac{\partial C_0}{\partial b^{(L)}} = 2\sigma'(z^{(L)})(a^{(L)} - y)$$
- Knowing the gradient and defining the learning rates we can update our weights and biases
- Final step is to allow multiple (many) neurons in each layer!

DEEP NN = MATRIX MULTIPLICATION

- Turns out that equations for deep NN look very nice and simple if you use matrix notation
- Let's look at the example with 2 hidden layers:
- **Forward propagation**:
  - $Z_1 = W_1 \cdot X + B_1$
  - $Z_2 = W_2 \cdot A_1 + B_2\,,\ A_1 = \sigma(Z_1)$
  - $Z_3 = W_3 \cdot A_2 + B_3\,,\ A_2 = \sigma(Z_2)$

DEEP NN = MATRIX MULTIPLICATION

- Turns out that equations for deep NN look very nice and simple if you use matrix notation
- Let's look at the example with 2 hidden layers:
- **Backpropagation** with $C = |A_3 - Y|^2$:
  - $dZ_3 = 2(A_3 - Y) \circ f'(Z3)\ ,\ \dfrac{\partial C}{\partial W_3} = dZ_3 \cdot A_2^T,\ \dfrac{\partial C}{\partial B_3} = dZ_3$
  - $dZ_2 = (W_3^T \cdot dZ_3) \circ f'(Z2)\ ,\ \dfrac{\partial C}{\partial W_2} = dZ_2 \cdot A_1^T,\ \dfrac{\partial C}{\partial B_2} = dZ_2$
  - $dZ_1 = (W_2^T \cdot dZ_2) \circ f'(Z1)\ ,\ \dfrac{\partial C}{\partial W_1} = dZ_1 \cdot X^T,\ \dfrac{\partial C}{\partial B_1} = dZ_1$

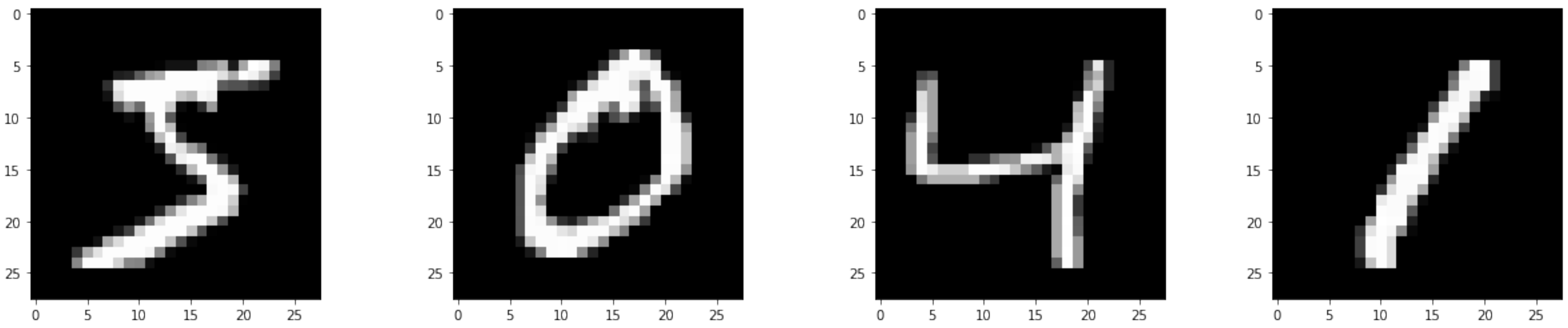
* careful not to mix the dot product ($\cdot$) and Hadamard product ($\circ$)

DEEP NN = MATRIX MULTIPLICATION

- Turns out that equations for deep NN look very nice and simple if you use matrix notation
- Let's look at the example with 2 hidden layers:
- **Update weights and biases**:
  - $W_1(t+1) = W_1(t) - \eta \frac{\partial C}{\partial W_1}, B_1(t+1) = B_1(t) - \eta \frac{\partial C}{\partial B_1}$
  - $W_2(t+1) = W_2(t) - \eta \frac{\partial C}{\partial W_2}, B_2(t+1) = B_2(t) - \eta \frac{\partial C}{\partial B_2}$
  - $W_3(t+1) = W_3(t) - \eta \frac{\partial C}{\partial W_3}, B_3(t+1) = B_3(t) - \eta \frac{\partial C}{\partial B_3}$

BUILDING DEEP NN FROM SCRATCH

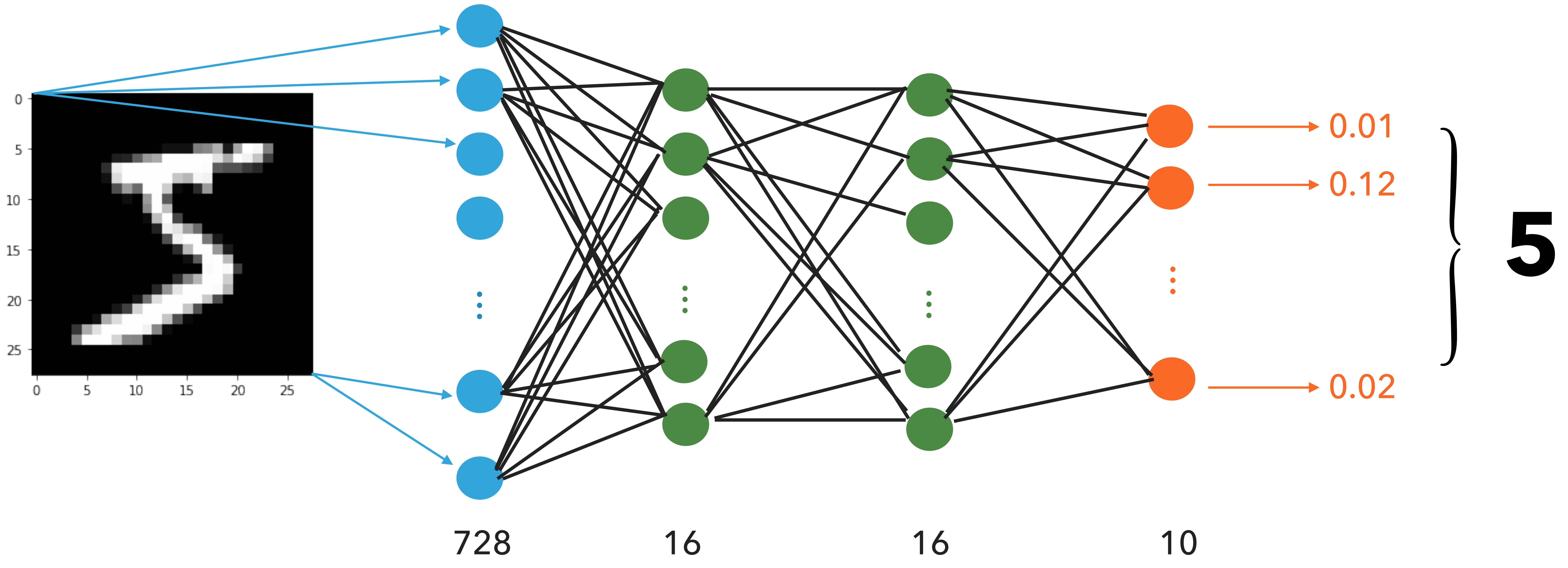
- Did we really understand deep NNs if we don't code these matrix equations ourselves?
- Let’s try building a NN that can recognise handwritten digits from the famous MNIST data



- We will take 70k 28x28 pixel images
- For each pixel there is a value 0-255 indicating its value on grayscale (0=black, 255= white)
- Each picture comes with a 0-9 label

BUILDING DEEP NN FROM SCRATCH

- We simply choose a NN with 28x28=784 input neurons
- We add 2 hidden layers with 16 neurons in each one
- Output layer has 10 neurons (idea is that each one carries information on how likely the number corresponds to that digit)

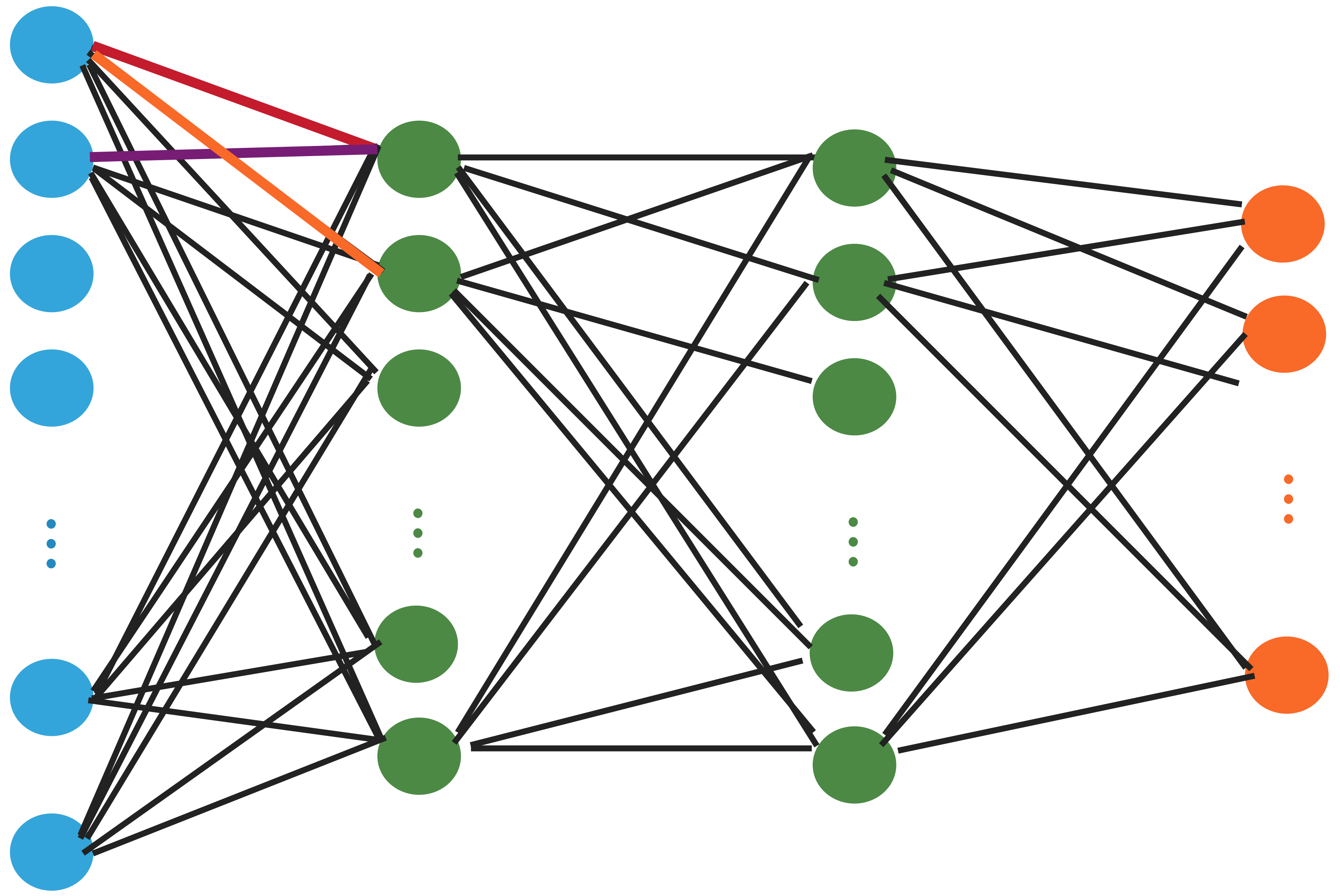


BUILDING DEEP NN FROM SCRATCH

- Let's define our matrices and their dimensions:

$$X = \begin{bmatrix} x_1^{(0,1)} & x_2^{(0,1)} & \cdots & x_{60000}^{(0,1)} \\ x_1^{(0,2)} & x_2^{(0,2)} & \cdots & x_{60000}^{(0,2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(0,727)} & x_2^{(0,727)} & \cdots & x_{60000}^{(0,727)} \end{bmatrix} \quad W_1 = \begin{bmatrix} w_{0,0}^{(1)} & w_{1,0}^{(1)} & \cdots & w_{727,0}^{(1)} \\ w_{0,1}^{(1)} & w_{1,1}^{(1)} & \cdots & w_{727,1}^{(1)} \\ \vdots & \vdots & \vdots & \vdots \\ w_{0,15}^{(1)} & w_{1,15}^{(1)} & \cdots & w_{727,15} \end{bmatrix}$$

-



BUILDING DEEP NN FROM SCRATCH

- We are ready to turn this math into python code!

```
def init_NN_parameters():
    w1 = np.random.rand(16,28*28)-0.5
    b1 = np.random.rand(16,1)-0.5
    w2 = np.random.rand(16,16)-0.5
    b2 = np.random.rand(16,1)-0.5
    w3 = np.random.rand(10,16)-0.5
    b3 = np.random.rand(10,1)-0.5
    return w1,b1,w2,b2,w3,b3
```

```
def forward_propagation(X, w1,b1,w2,b2,w3,b3):
    Z1 = w1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = w2.dot(A1) + b2
    A2 = ReLU(Z2)
    Z3 = w3.dot(A2) + b3
    A3 = ReLU(Z3)
    return Z1,A1,Z2,A2,Z3,A3
```

```
def back_propagation(X,Y, Z1,A1,Z2,A2,Z3,A3,w2,w3):
    m = Y.size
    dZ3 = 2*(A3 - Y)*der_ReLU(Z3)
    dW3 = (1/m)*dZ3.dot(A2.T)
    dB3 = (1/m)*np.sum(dZ3,1)

    dZ2 = w3.T.dot(dZ3)*der_ReLU(Z2)
    dW2 = (1/m)*dZ2.dot(A1.T)
    dB2 = (1/m)*np.sum(dZ2,1)

    dZ1 = w2.T.dot(dZ2)*der_ReLU(Z1)
    dW1 = (1/m)*dZ1.dot(X.T)
    dB1 = (1/m)*np.sum(dZ1,1)

    return dW3,dB3,dW2,dB2,dW1,dB1
```

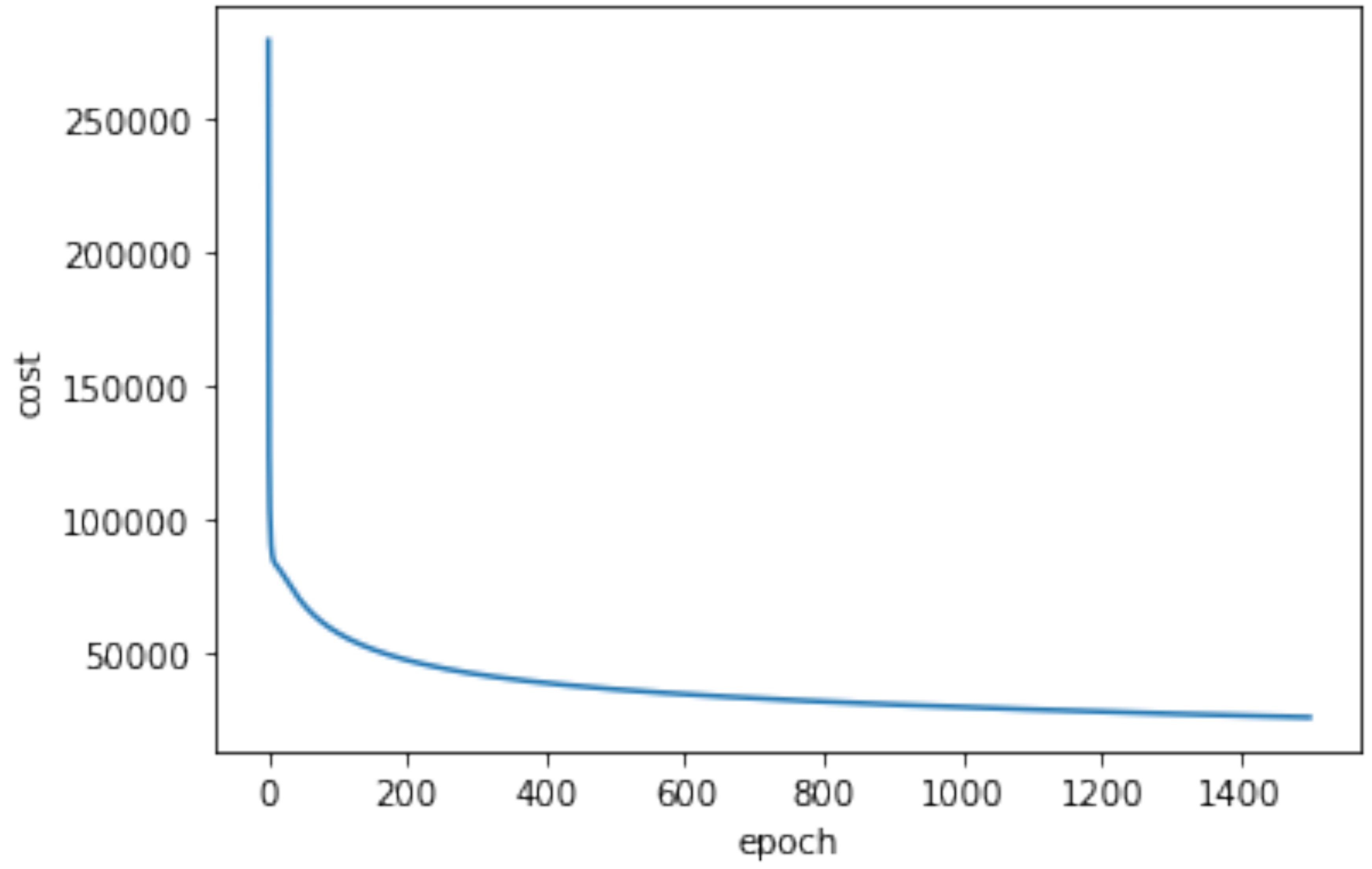
```
def update_params(alpha, w3,b3,w2,b2,w1,b1,dW3,dB3,dW2,dB2,dW1,dB1):
    w3 = w3 - alpha*dW3
    b3 = b3 - alpha*dB3.reshape(10,1)

    w2 = w2 - alpha*dW2
    b2 = b2 - alpha*dB2.reshape(16,1)

    w1 = w1 - alpha*dW1
    b1 = b1 - alpha*dB1.reshape(16,1)

    return w3,b3,w2,b2,w1,b1
```

BUILDING DEEP NN FROM SCRATCH

- We decide to split our dataset into 60k training points and 10k test points.
- Training it for 1500 epochs with 0.1 learning rate:

```
Epoch 0
[7 6 7 ... 5 6 7] [5 0 4 ... 5 6 8]
Accuracy 0.11831666666666667
Cost function = 279650.33817947295
Epoch 50
[9 0 4 ... 5 6 8] [5 0 4 ... 5 6 8]
Accuracy 0.263
Cost function = 68318.50986350718
Epoch 100
[5 0 4 ... 5 6 8] [5 0 4 ... 5 6 8]
Accuracy 0.35961666666666664
Cost function = 57361.1053264526
```

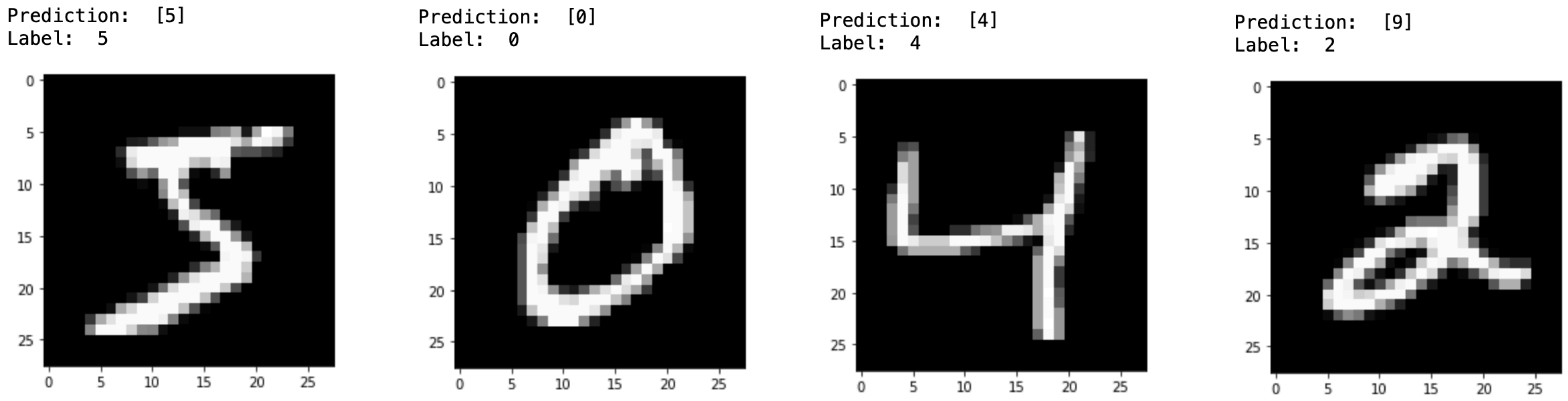
```
Epoch 1400
[5 0 4 ... 5 6 8] [5 0 4 ... 5 6 8]
Accuracy 0.7349333333333333
Cost function = 26348.145017085022
Epoch 1450
[5 0 4 ... 5 6 8] [5 0 4 ... 5 6 8]
Accuracy 0.7387
Cost function = 26018.279174079744
Epoch 1499
[5 0 4 ... 5 6 8] [5 0 4 ... 5 6 8]
Accuracy 0.7420333333333333
Cost function = 25708.492532038963
```



- Don't forget to check for overtraining!

```
[7 2 1 ... 4 8 6] [7 2 1 ... 4 5 6]
Accuracy on test data 0.7494
```

BUILDING DEEP NN FROM SCRATCH



- Not a bad performance for numpy only deep NN!
- [CHALLENGE] Can you beat me? Can you write a better deep NN to solve the same problem using only NumPy?

CONCLUSIONS AND OUTLOOK

- Machine Learning is only a powerful tool in the hands of someone who understands how it works
- It is impossible to fully understand every single detail of what happens inside of a multilayered NN/BDT
  - But that doesn't mean you can't fully understand the concept and mathematics behind it and be able to apply it in a simplified version
- Everything we have discussed so far is considered Classical (old) Machine Learning and is not even close in performance with State of the Art
  - However, all the concepts that we have learned about are the key of modern ML
- In the reminder of the school you will learn about state of the art ML and its application to HEP
  - Whenever things seem to be too complicated to understand, just drop back to basics and try to understand it in an overly simplified version like we did together