Introduzione alle Reti Neurali

Why Now?

1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

Neural Networks date back decades, so why the resurgence?

1. Big Data
- Larger Datasets
- Easier Collection & Storage

IMAGENET

WIKIPEDIA
The Free Encyclopedia

2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software
- Improved Techniques
- New Models
- Toolboxes

TensorFlow

ARTIFICIAL INTELLIGENCE

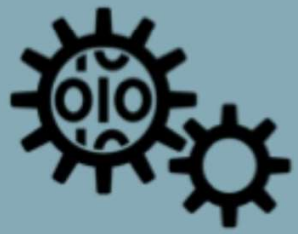
IS NOT NEW

ARTIFICIAL INTELLIGENCE

Any technique which enables computers to mimic human behavior

MACHINE LEARNING

AI techniques that give computers the ability to learn without being explicitly programmed to do so

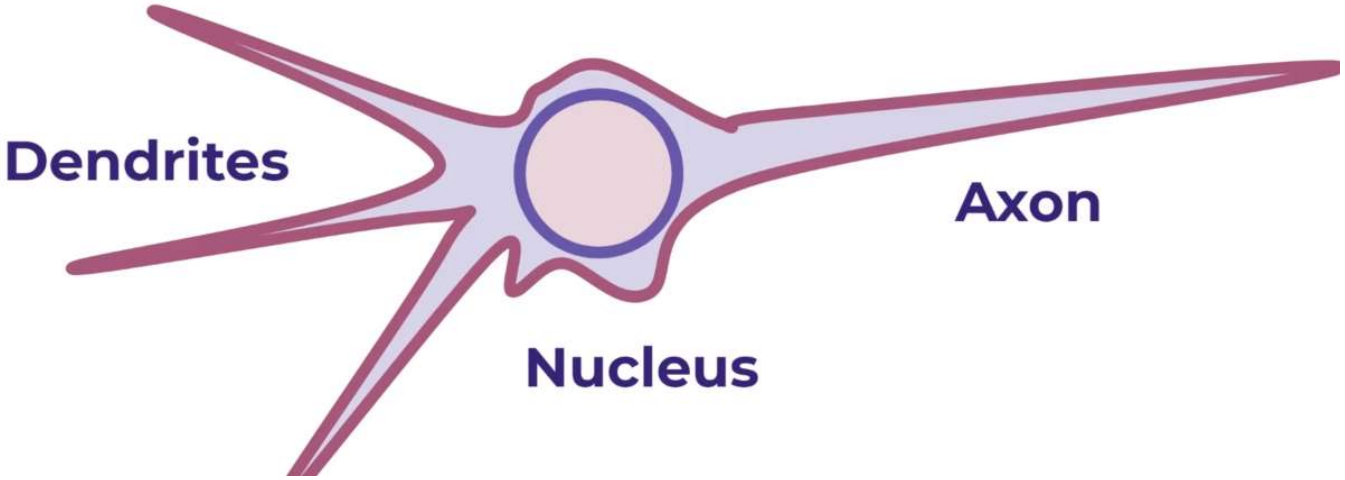


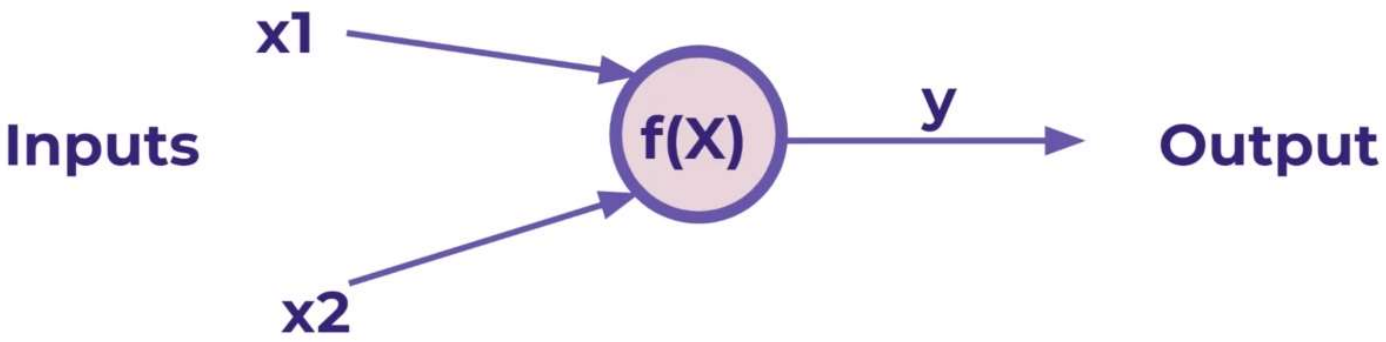
DEEP LEARNING

A subset of ML which make the computation of multi-layer neural networks feasible

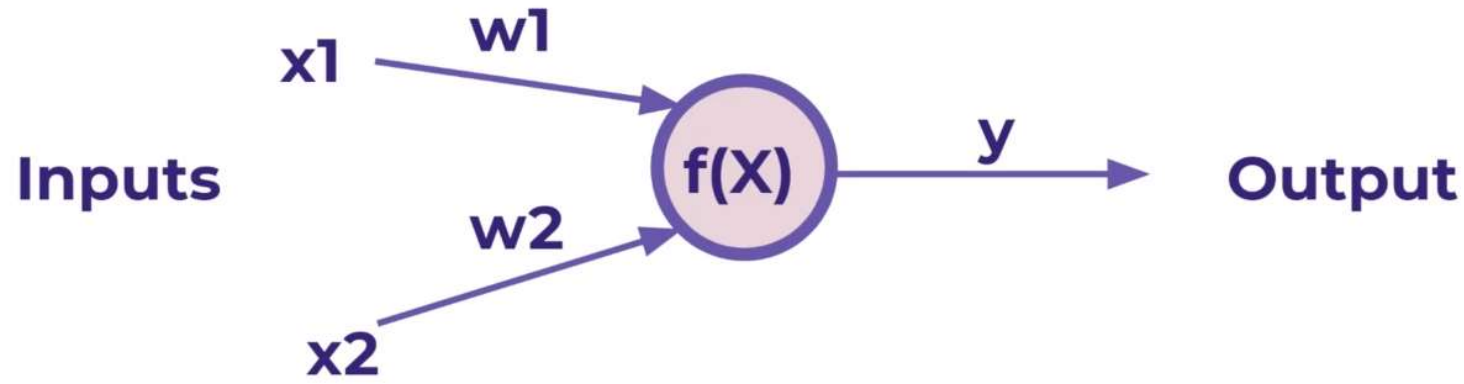
1950's 1960's 1970's 1980's 1990's 2000's 2010s

PERCEPTRON

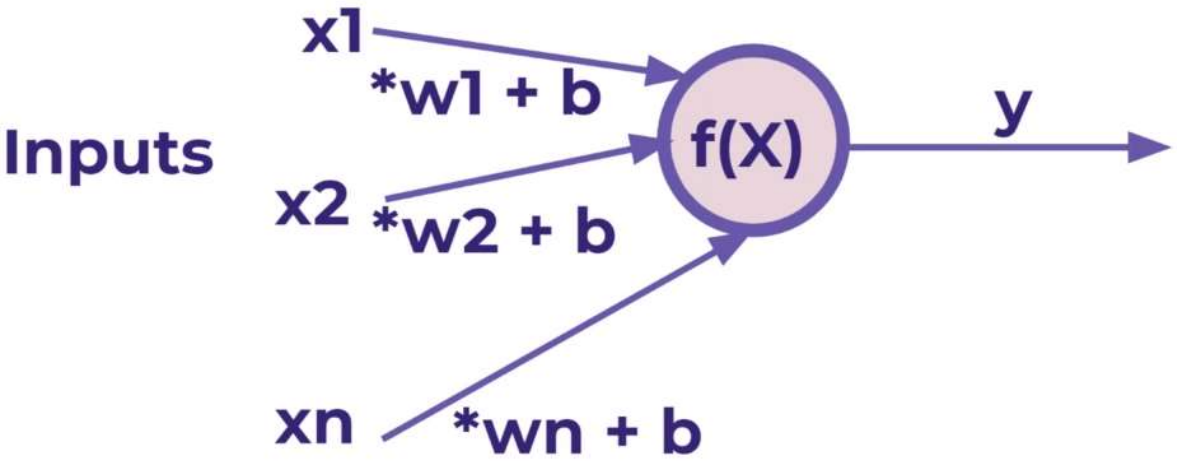




- But what if an **x** is zero? **w** won't change anything!



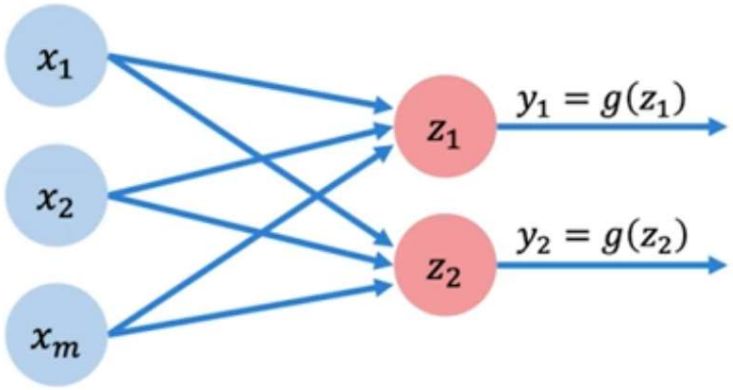
- We can expand this to a generalization:



$$\hat{y} = \sum_{i=1}^{n} x_i w_i + b_i$$

Multi Output Perceptron

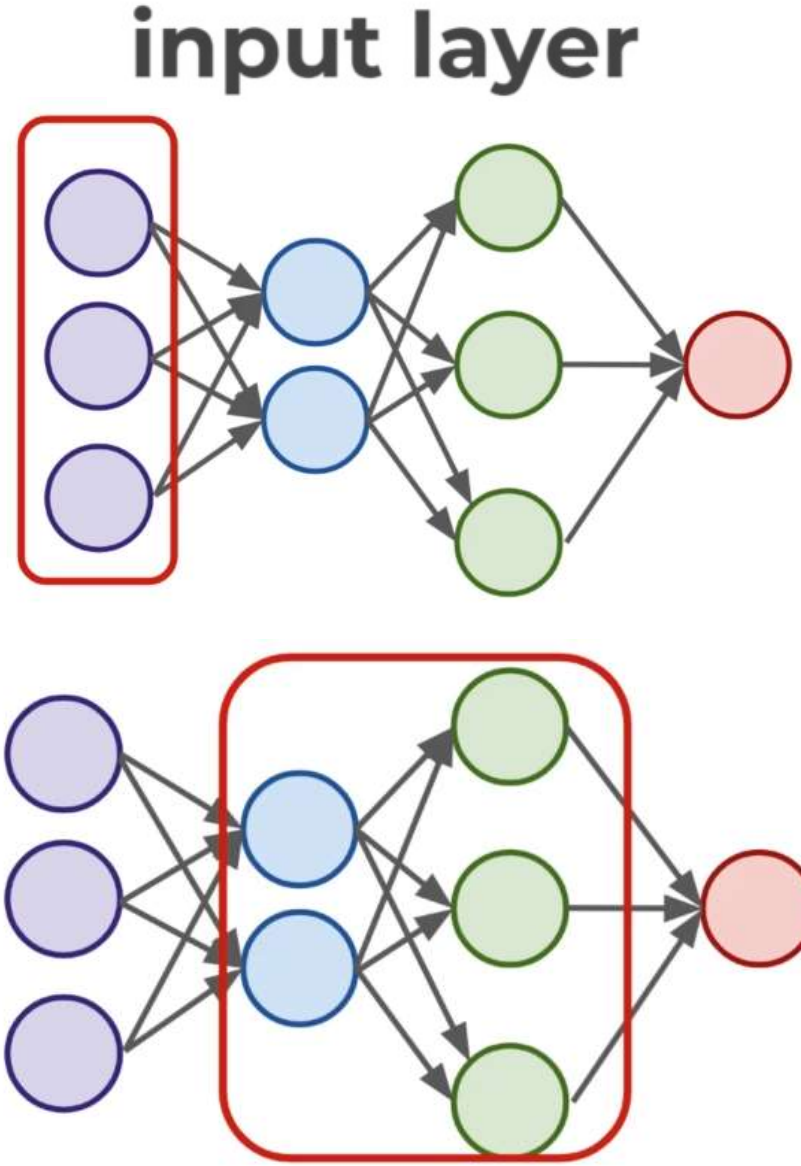
Because all inputs are densely connected to all outputs, these layers are called **Dense** layers

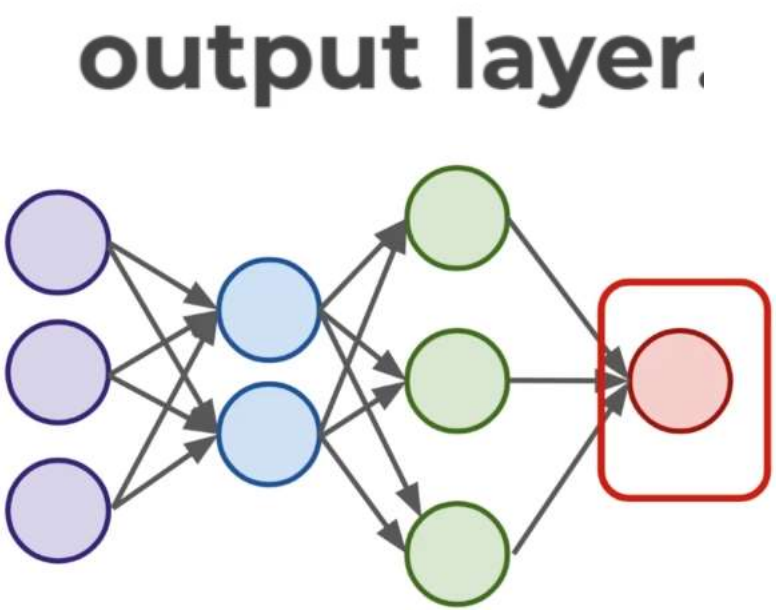


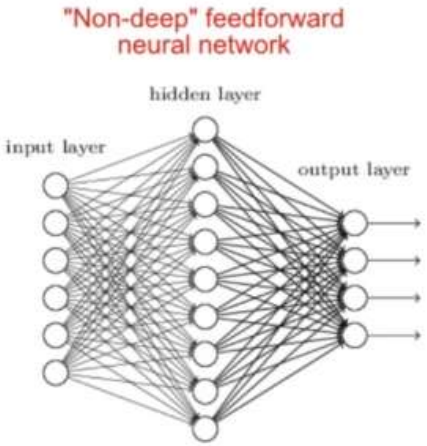
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

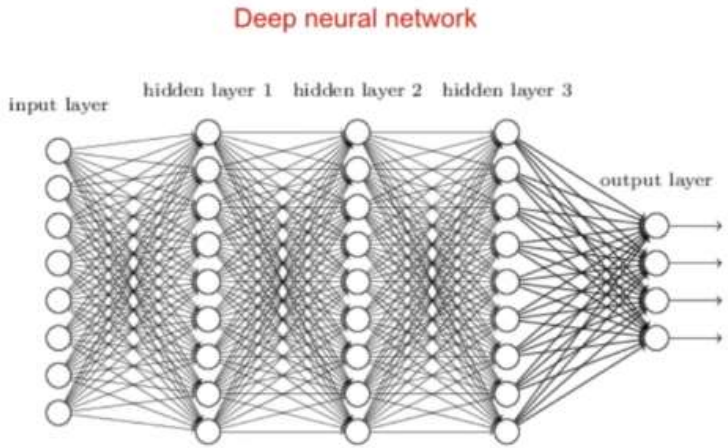
Dense layer from scratch

```
class MyDenseLayer(tf.keras.layers.Layer):
  def __init__(self, input_dim, output_dim):
    super(MyDenseLayer, self).__init__()

    # Initialize weights and bias
    self.W = self.add_weight([input_dim, output_dim])
    self.b = self.add_weight([1, output_dim])

  def call(self, inputs):
    # Forward propagate the inputs
    z = tf.matmul(inputs, self.W) + self.b

    # Feed through a non-linear activation
    output = tf.math.sigmoid(z)

    return output
```


input layer

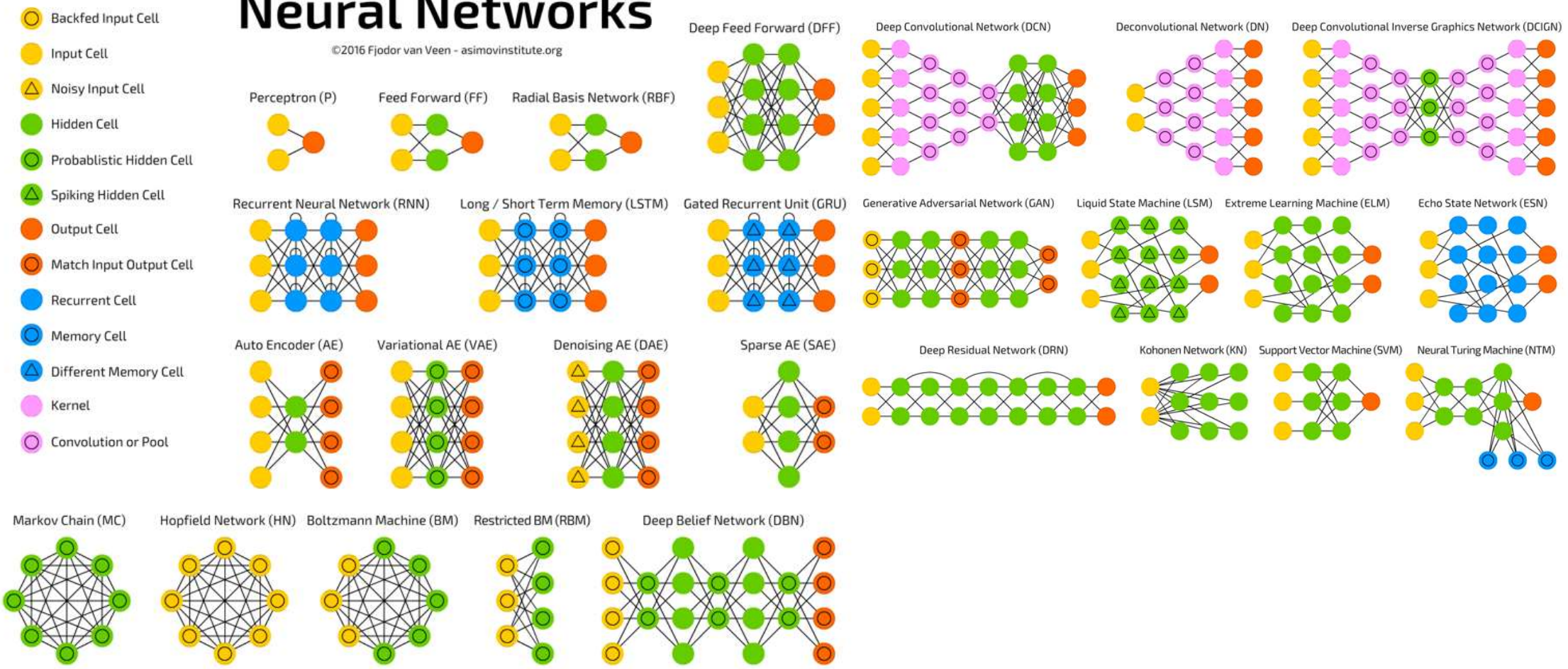

output layer


"Non-deep" feedforward
neural network
hidden layer
input layer
output layer


Deep neural network
input layer
hidden layer 1
hidden layer 2
hidden layer 3
output layer

A mostly complete chart of

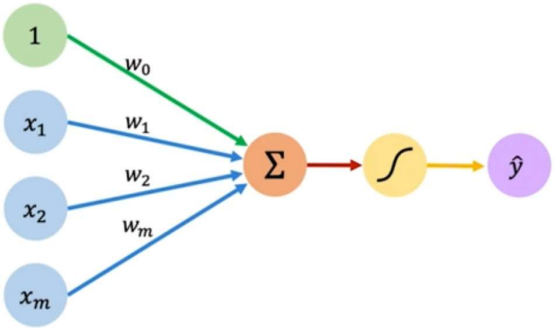
Neural Networks



- Terminology:
  - Input Layer: First layer that directly accepts real data values
  - Hidden Layer: Any layer between input and output layers
  - Output Layer: The final estimate of the output.
- What is incredible about the neural network framework is that it can be used to approximate any function.
- Zhou Lu and later on Boris Hanin proved mathematically that Neural Networks can approximate any convex continuous function.

- Previously in our simple model we saw that the perceptron itself contained a very simple summation function f(x).
- For most use cases however that won't be useful, we'll want to be able to set constraints to our output values, especially in classification tasks.
- In a classification tasks, it would be useful to have all outputs fall between 0 and 1.
- These values can then present probability assignments for each class.
- In the next lecture, we'll explore how to use **activation functions** to set boundaries to output values from the neuron.

The Perceptron: Forward Propagation



Inputs Weights Sum Non-Linearity Output

Output

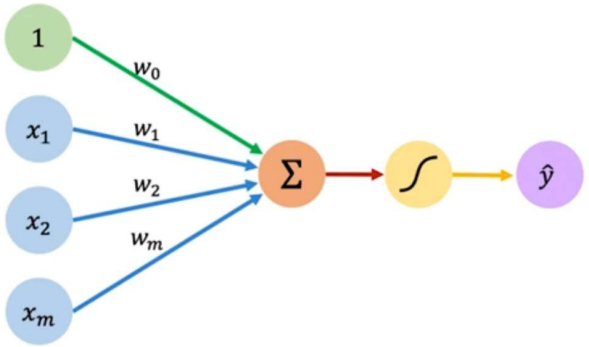
Linear combination of inputs

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

Non-linear activation function

Bias

The Perceptron: Forward Propagation



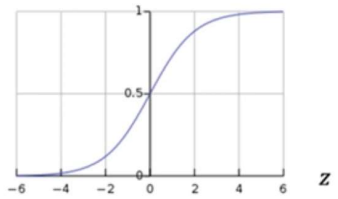
Inputs Weights Sum Non-Linearity Output

Activation Functions

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

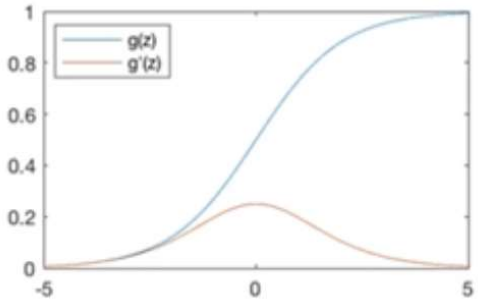
- Example: sigmoid function

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Common Activation Functions

Sigmoid Function

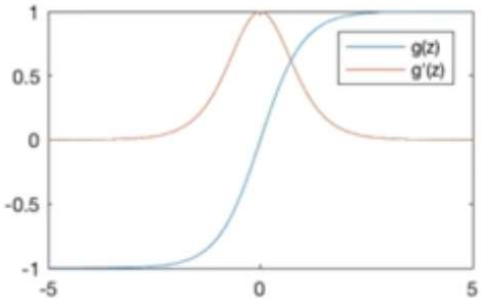


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

`tf.math.sigmoid(z)`

Hyperbolic Tangent

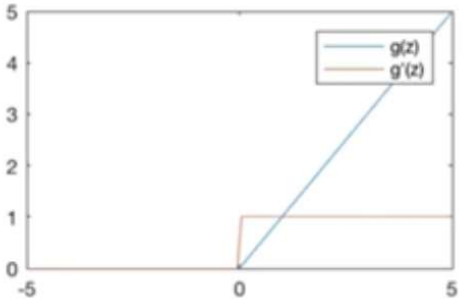


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

`tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



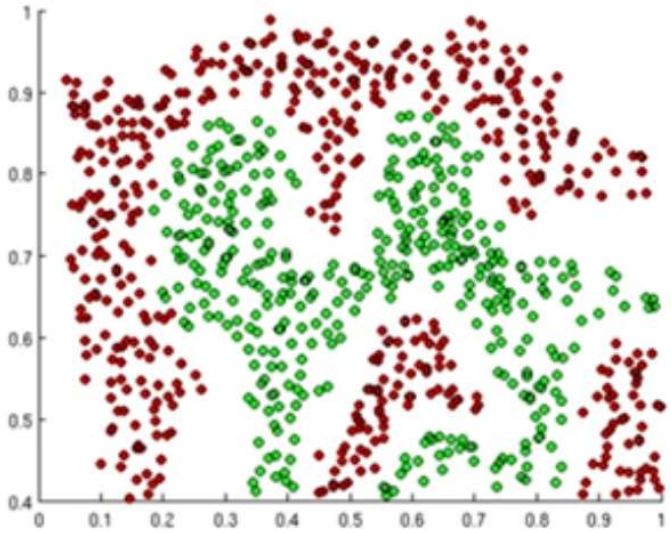
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

`tf.nn.relu(z)`

Importance of Activation Functions

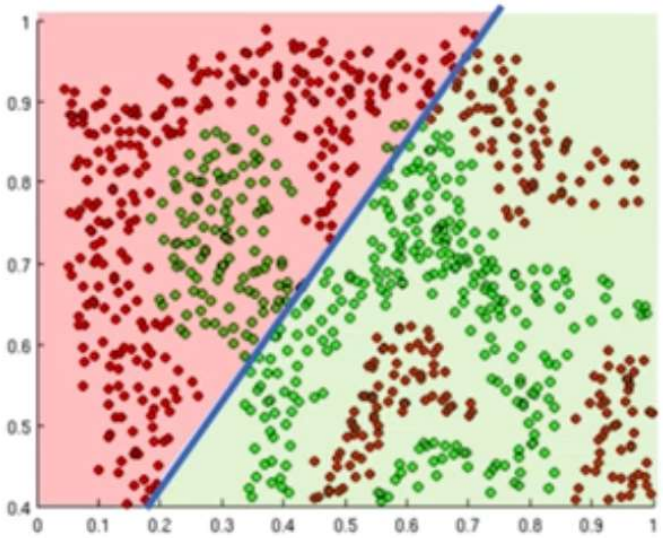
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



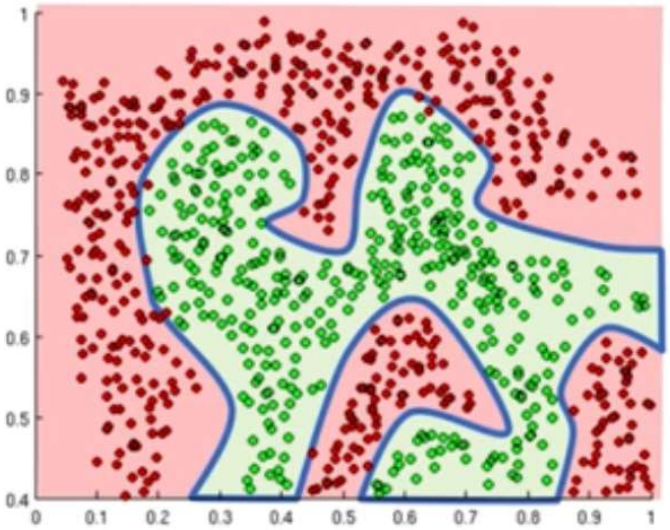
What if we wanted to build a neural network to distinguish green vs red points?

Importance of Activation Functions

*The purpose of activation functions is to **introduce non-linearities** into the network*

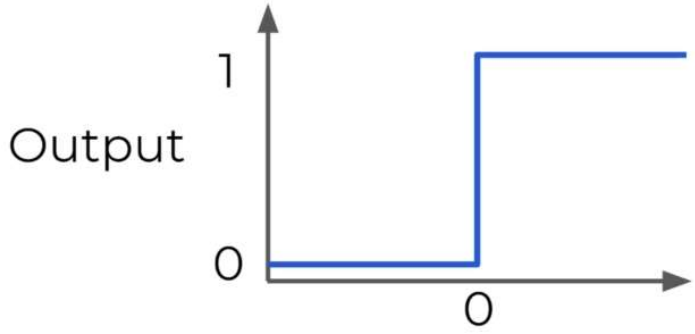


Linear activation functions produce linear decisions no matter the network size



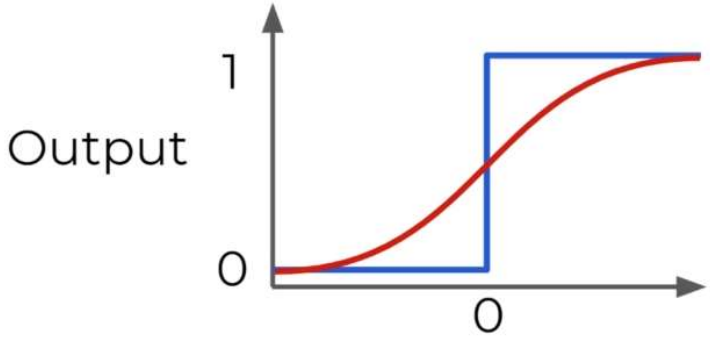
Non-linearities allow us to approximate arbitrarily complex functions

- The most simple networks rely on a basic **step function** that outputs 0 or 1.



z = wx + b

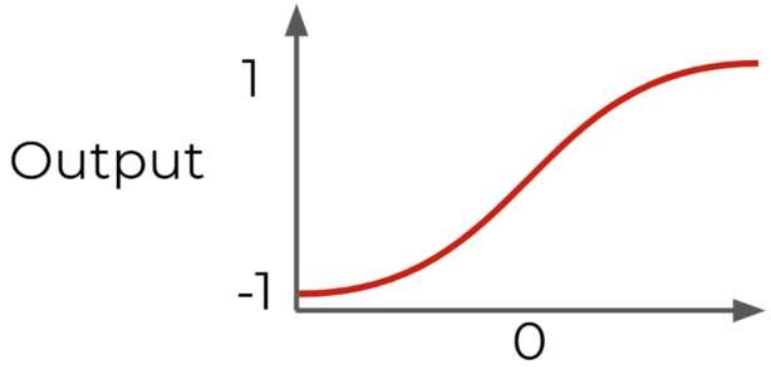
This still works for classification, and will be more sensitive to small changes.



$$f(z) = \frac{1}{1 + e^{(-z)}}$$

z = wx + b

- Hyperbolic Tangent: tanh(z)
- Outputs between -1 and 1 instead of 0 to 1

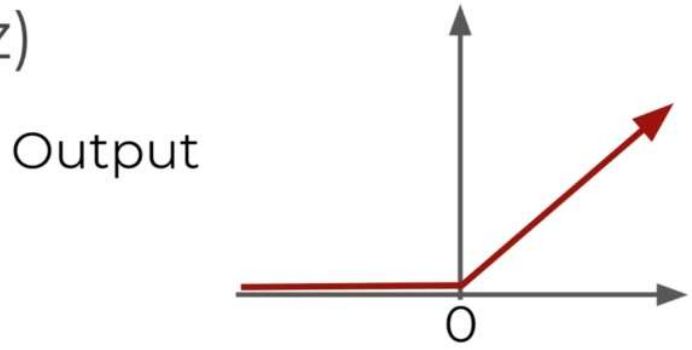


$$\cosh x = \frac{\mathbf{e}^{x} + \mathbf{e}^{-x}}{2}$$

$$\sinh x = \frac{\mathbf{e}^{x} - \mathbf{e}^{-x}}{2}$$

$$\tanh x = \frac{\sinh x}{\cosh x}$$

- Rectified Linear Unit (ReLU): This is actually a relatively simple function: max(0,z)



z = wx + b

Deep Learning

Activation Functions: Dance Moves

Sigmoid

$$y = \frac{1}{1+e^{-x}}$$

Tanh

$$y = \tanh(x)$$

Step Function

$$y = \begin{cases} 0, & x<n \\ 1, & x \geq n \end{cases}$$

Softplus

$$y = Ln(1+e^{x})$$

ReLU

$$y = \begin{cases} 0, & x<0 \\ x, & x \geq 0 \end{cases}$$

Softsign

$$y = \frac{x}{(1+|x|)}$$

ELU

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

Log of Sigmoid

$$y = Ln\left(\frac{1}{1+e^{-x}}\right)$$

Swish

$$y = \frac{x}{1+e^{-x}}$$

Sinc

$$y = \frac{\sin(x)}{x}$$

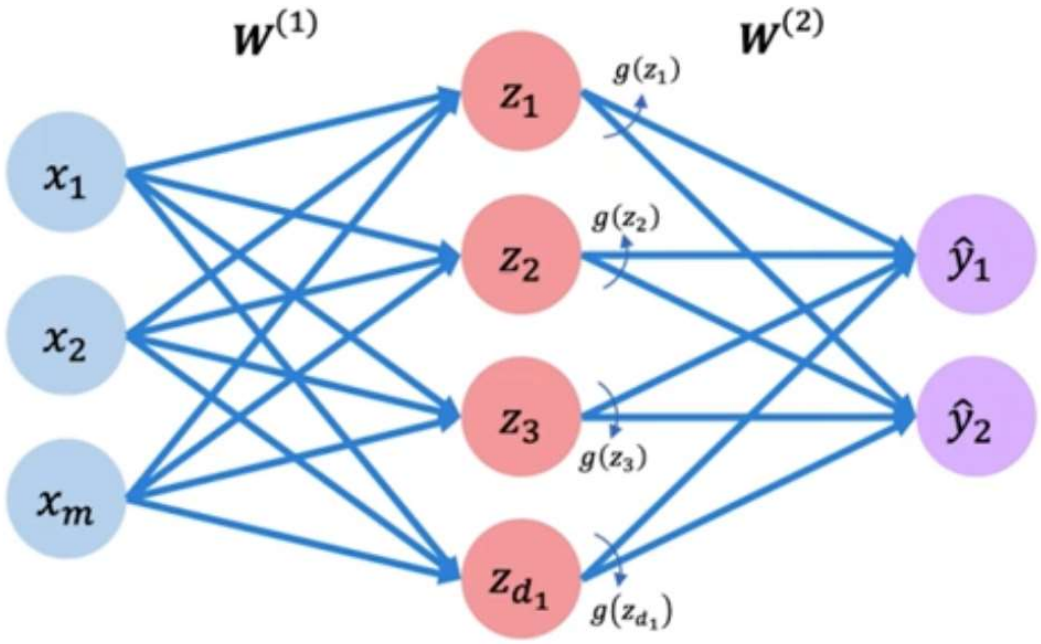
Leaky ReLU

$$y = \max(0.1x, x)$$

Mish

$$y = x\left(\tanh\left(\text{softplus}(x)\right)\right)$$

Single Layer Neural Network

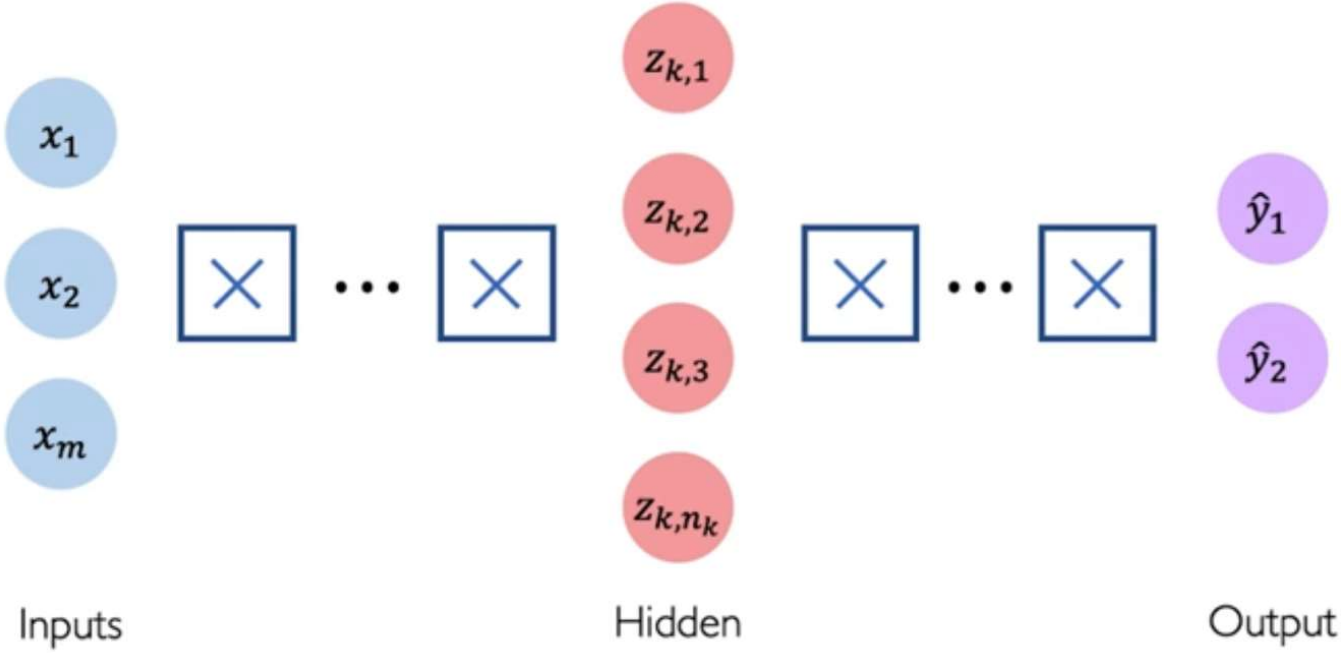


Inputs Hidden Final Output

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)}$$

$$\hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \, w_{j,i}^{(2)}\right)$$

Deep Neural Network

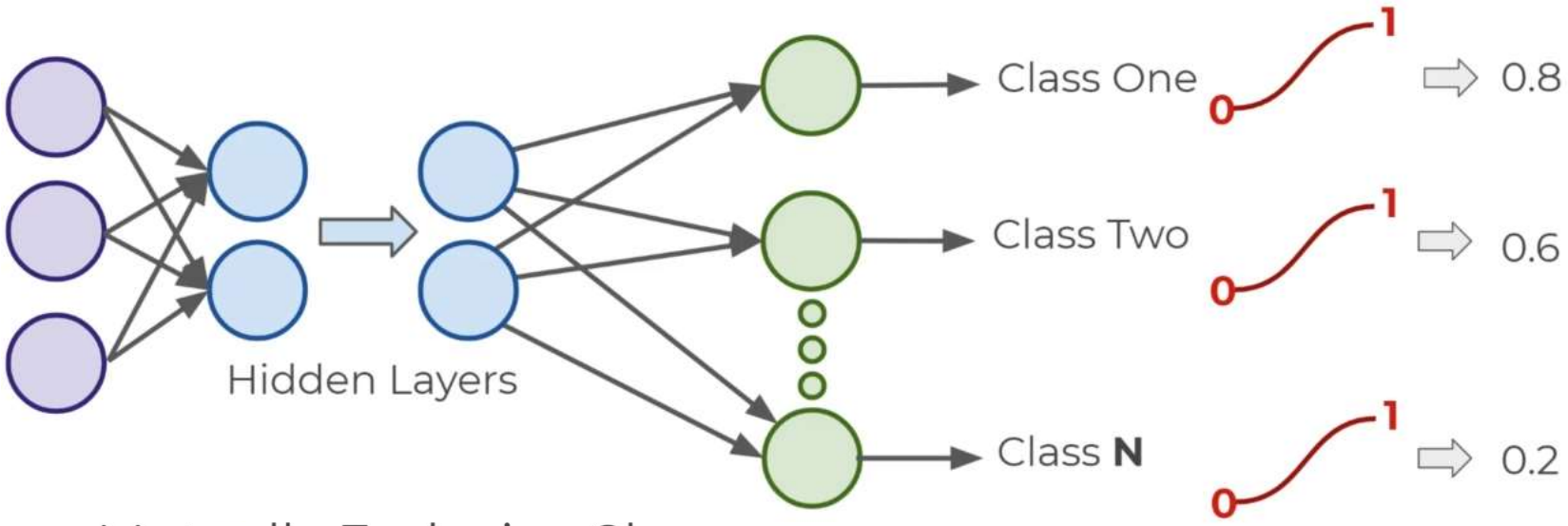



```tensorflow
import tensorflow as tf

model = tf.keras.Sequential([
  tf.keras.layers.Dense(n1),
  tf.keras.layers.Dense(n2),
  ⋮
  tf.keras.layers.Dense(2)
])
```

$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

Multiclass Classification

- Sigmoid Function for Non-Exclusive Classes



- Mutually Exclusive Classes
  - But what do we do when each data point can only have a single class assigned to it?
  - We can use the **softmax function** for this!

Softmax Function

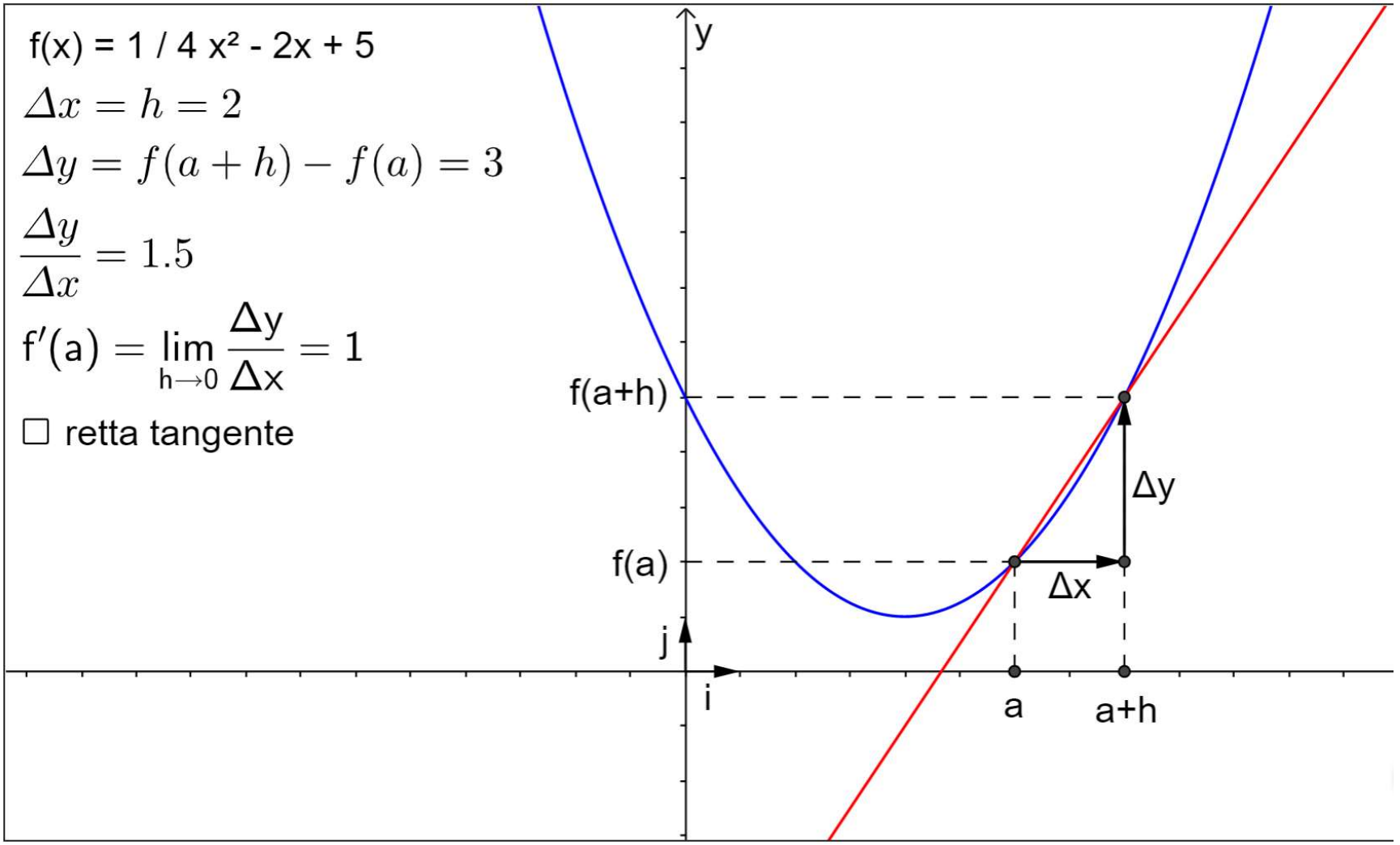
$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K$$

- Mutually Exclusive Classes
  - The range will be 0 to 1, and **the sum of all the probabilities will be equal to one.**
  - The model returns the probabilities of each class and the target class chosen will have the highest probability.

Mutually Exclusive Classes

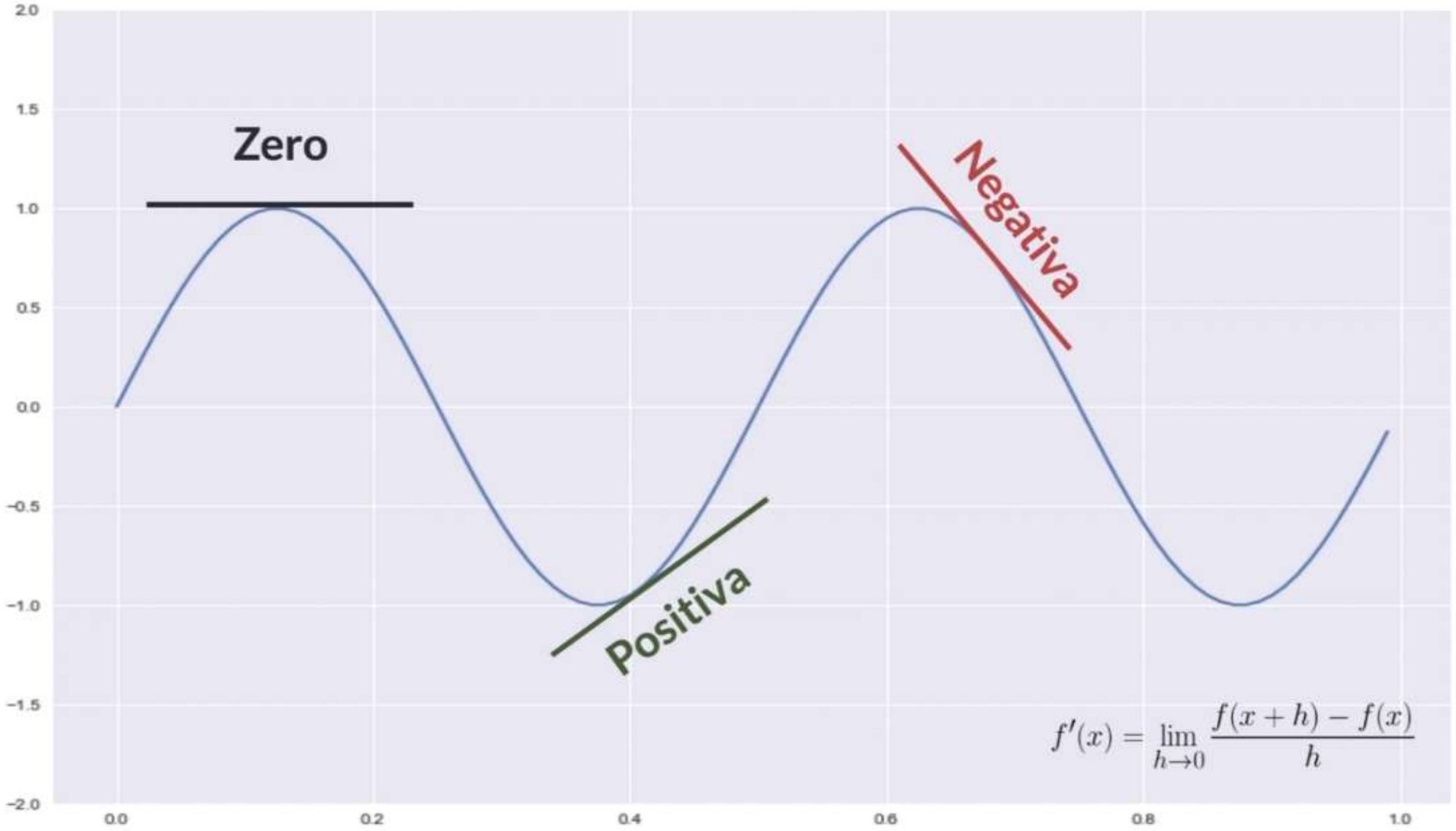
- The probabilities for each class all sum up to 1. We choose the highest probability as our assignment.
  - [Red , Green , Blue]
  - [0.1 , 0.6 , 0.3]

LA DERIVATA

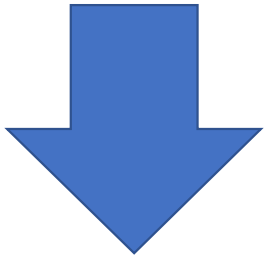

f(x) = 1 / 4 x² - 2x + 5
$\Delta x = h = 2$
$\Delta y = f(a+h) - f(a) = 3$
$\frac{\Delta y}{\Delta x} = 1.5$
$f'(a) = \lim_{h \to 0} \frac{\Delta y}{\Delta x} = 1$
retta tangente
y
f(a+h)
f(a)
Δy
Δx
j
i
a
a+h

Derivate fondamentali

	Funzione	Derivata
costante	$f(x) = \mathrm{n}$	$f'(x) = 0$
lineare	$f(x) = x$	$f'(x) = 1$
potenza	$f(x) = x^{\mathrm{n}}$	$f'(x) = \mathrm{n}x^{n-1}$
esponenziale	$f(x) = e^x$	$f'(x) = e^x$
logaritmica	$f(x) = \ln x$	$f'(x) = \frac{1}{x}$


Zero
Negativa
Positiva
$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.0
0.2
0.4
0.6
0.8
1.0

How do we evaluate the performance of a Neural network?



COST FUNCTION

Cost Funcion/Loss Function qunatifies the error between predicted values and expected values and presents it in the form of a single real number

- The cost function (often referred to as a loss function) must be an average so it can output a single value.
- We can keep track of our loss/cost during training to monitor network performance.

punishes large errors!

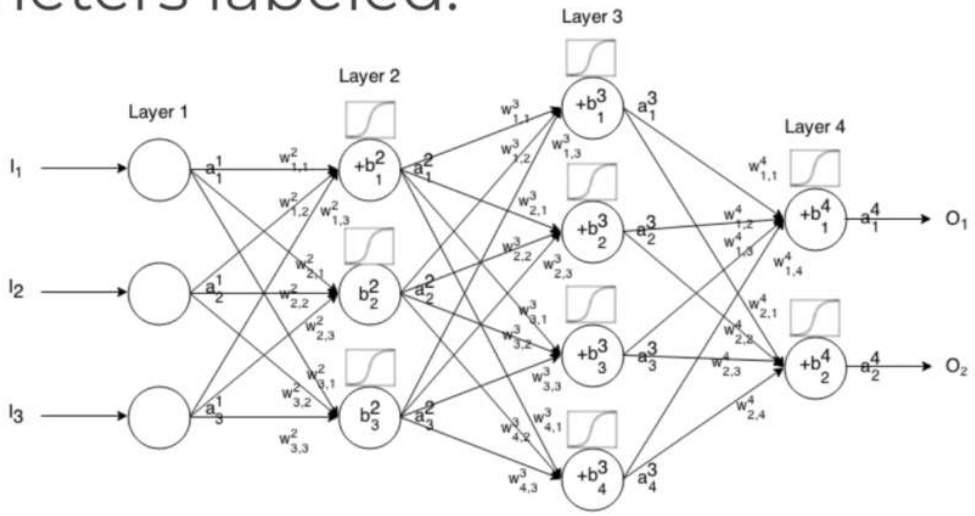
$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

- **W** is our neural network's weights, **B** is our neural network's biases, $\mathbf{S^r}$ is the input of a single training sample, and $\mathbf{E^r}$ is the desired output of that training sample.

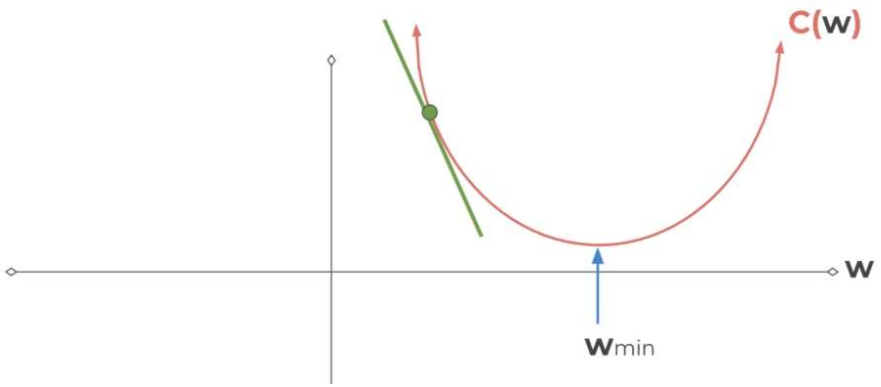
$$C(W, B, S^r, E^r)$$

- For simplicity, let's imagine we only had one weight in our cost function **w.**
- We want to **minimize** our loss/cost (overall error).
- Which means we need to figure out what value of **w** results in the minimum of **C(w)**

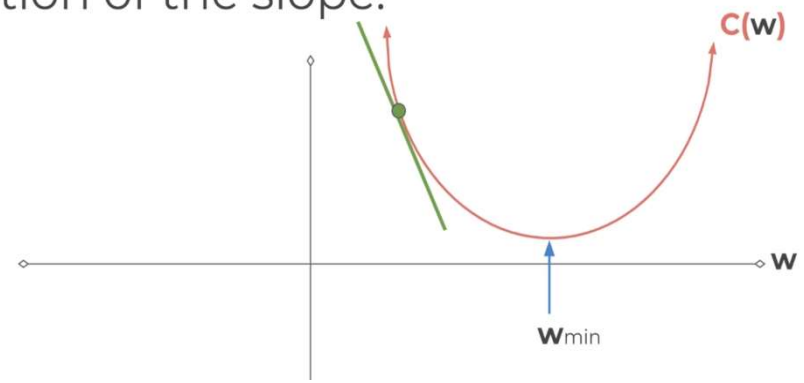
Here is a small network with all its parameters labeled:



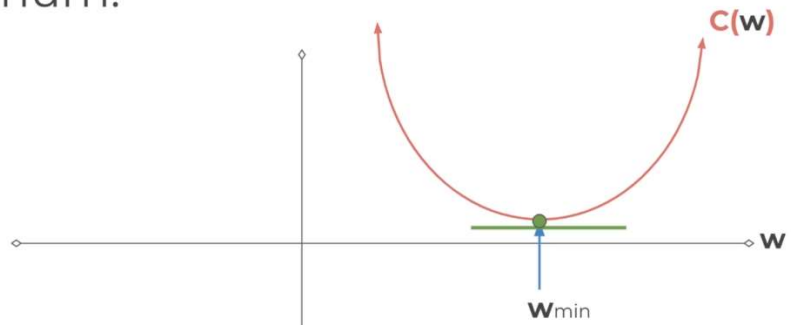
We can calculate the slope at a point



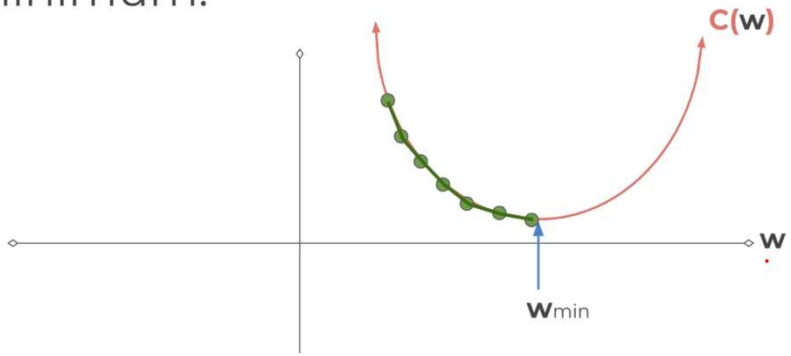
Then we move in the downward direction of the slope.



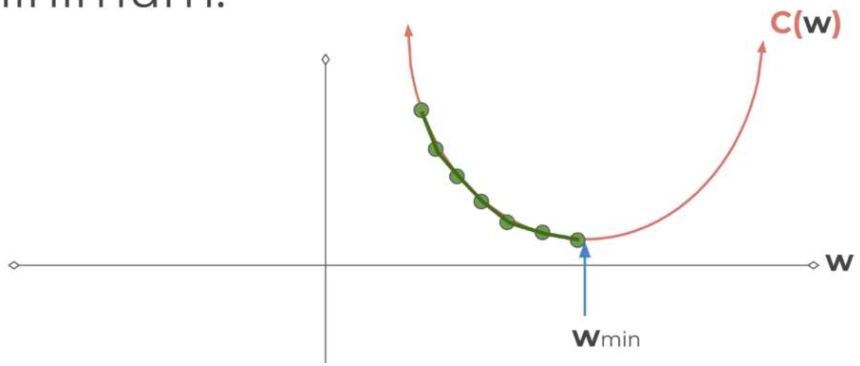
Until we converge to zero, indicating a minimum.



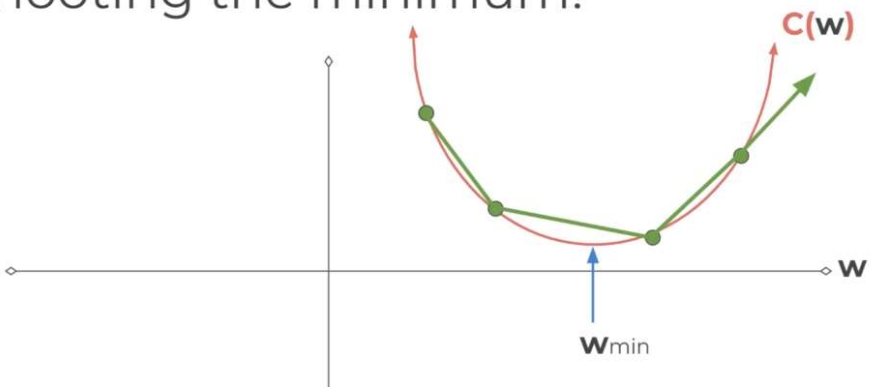
Smaller steps sizes take longer to find the minimum.



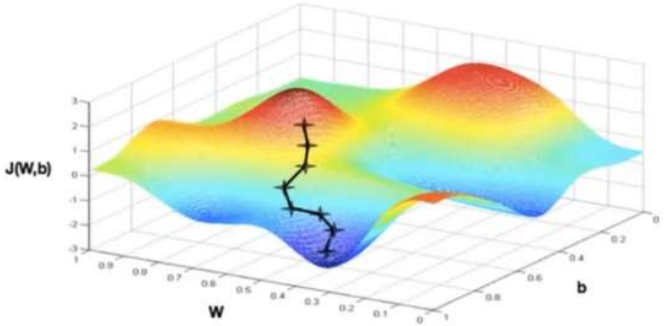
Smaller steps sizes take longer to find the minimum.



Larger steps are faster, but we risk overshooting the minimum!



Realistically we're calculating this descent in an n-dimensional space for all our weights.



Derivate parziali

$$\frac{df}{dx} \qquad \frac{df}{dy} \qquad \frac{df}{dz}$$

Si ottengono calcolando la derivata rispetto ad una variabile per volta e trattando le altre come costanti

Gradiente di una funzione

$$\nabla f(x,y,z) = [\ \frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}\ ]$$

E' un vettore composto dalle derivate parziali della funzione

Come si calcola il gradiente della funzione di costo ?

Regressione Somma dei quadrati residui

$$\mathrm{J}(w,b) = \frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - (w^{(i)}x^{(i)} + b))$$

Classificazione Log Loss

$$\mathrm{J}(w,b) = -\sum_{i=1}^{N}(y^{(i)} \log(\phi(w^{(i)}x^{(i)} + b) - (1-y)\log(1 - \phi(w^{(i)}x^{(i)} + b)$$

Differenziando le equazioni otteniamo le stesse derivate parziali

Derivata della funzione di costo rispetto al peso

$$\frac{d\mathrm{J}}{dw} = \sum_{i=1}^{N}(\phi(w^{(i)}x^{(i)} + b) - y^{(i)})x^{(i)}$$

Derivata della funzione di costo rispetto al bias

$$\frac{d\mathrm{J}}{db} = \sum_{i=1}^{N}(\phi(w^{(i)}x^{(i)} + b) - y^{(i)})$$

... insieme costituiscono il gradiente della funzione di costo $\nabla\mathrm{J}(w,b) = [\frac{d\mathrm{J}}{dw}, \frac{d\mathrm{J}}{db}]$

```
def gradient_descent(X,y,alpha=0.001,epochs=100):

    w = rand(X.shape[1],1)
    b = 0

    for _ in range(epochs):

        z = dot(X,w)+b
        a = activation(z)
        dw = X.T.dot(a-y)
        db = sum(a-y)
        w = w - alpha*dw
        b = b - alpha*db

   return w,b
```

```
def stochastic_GD(X,y,epochs=100):

    w = rand(X.shape[1],1)
    b = rand(1,1)

    for _ in range(epochs):
        for x_i,y_i in zip(X,y):
            z = dot(x_i,w)+b
            a = activation(z)
            dw = (a-y_i)*x_i
            db = a-y_i
            w = w - alpha*dw
            b = b - alpha*db

   return w,b
```

Gradient Descent

Algorithm

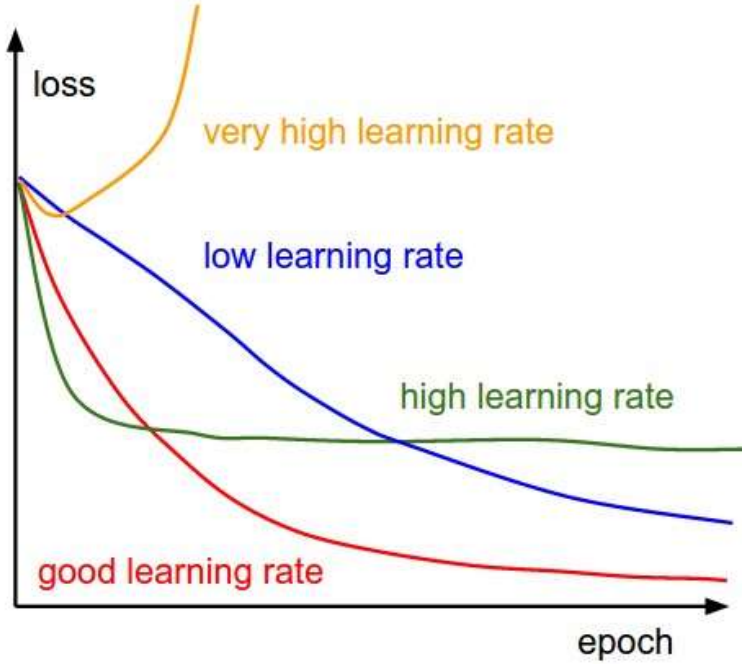
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$
4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```

EPOCHS/ LEARNING RATE

One Epochs is when an ENTIRE dataset is passed forward and backward throught the neaural network only once

Why Multiple Epochs?
Passing the entire dataset through a neural network is not enough.
To optimise the learning rate, Stochastic Gradient Descent is used, which is an interative process. So, updating the weights with single pass or one epoch is not enough!

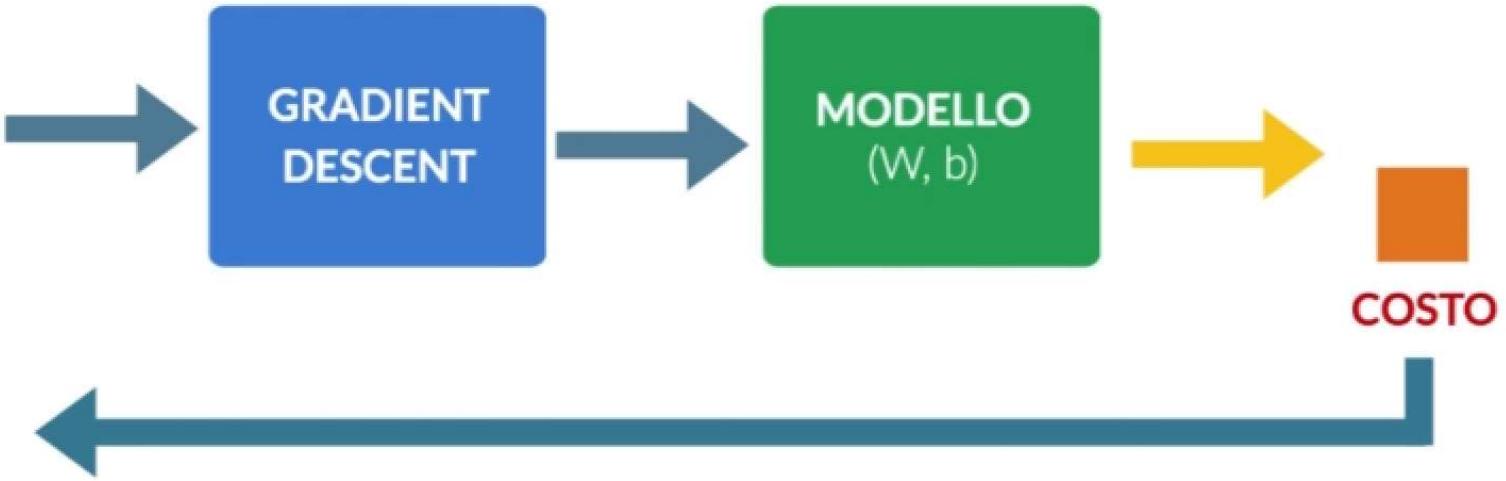


The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs.

Stochastic Gradient Descent

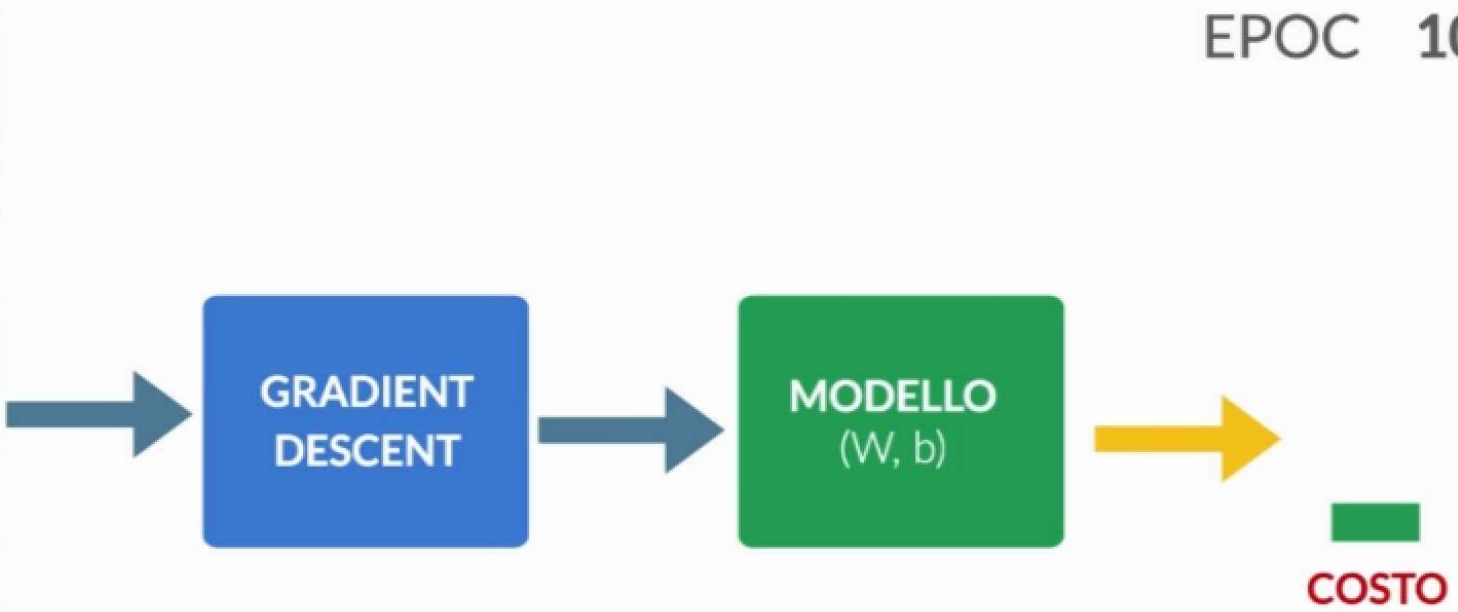
METRI QUADRI	VALORE (€)
120	180.000
50	100.000
80	130.000
60	120.000
100	150.000
110	200.000
30	120.000
....	....

EPOC 3



Stochastic Gradient Descent

METRI QUADRI	VALORE (€)
50	100.000
30	120.000
100	150.000
120	180.000
60	120.000
110	200.000
80	130.000
....	....



Stochastic Gradient Descent

VANTAGGI

Pesa poco in memoria

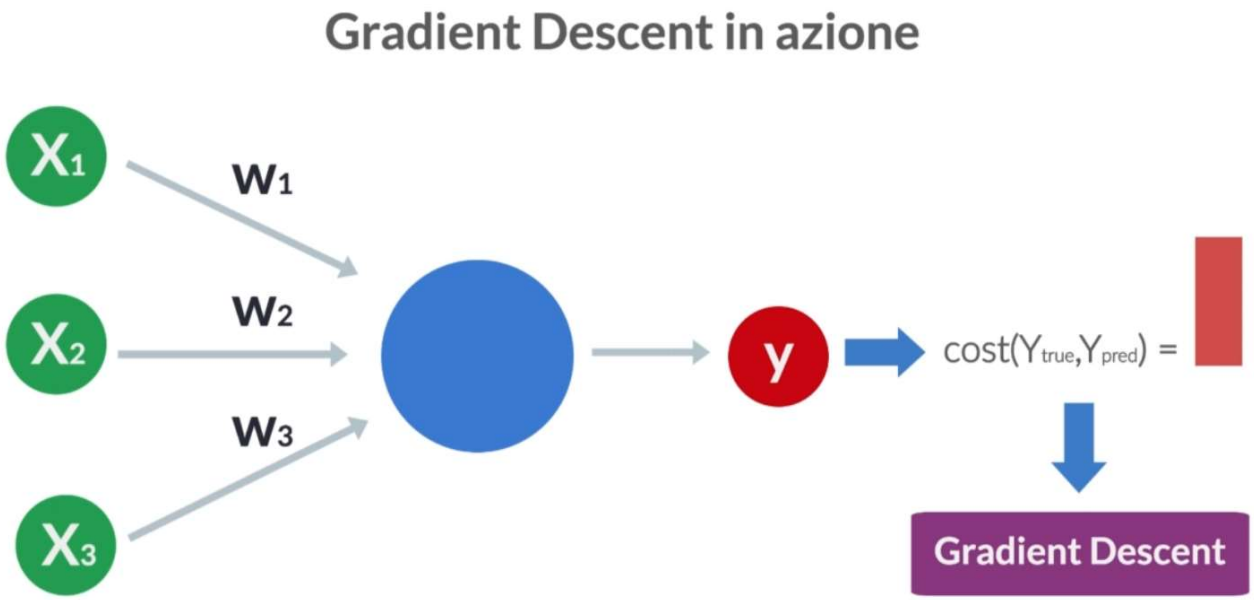
E' sufficiente caricare un esempio per volta

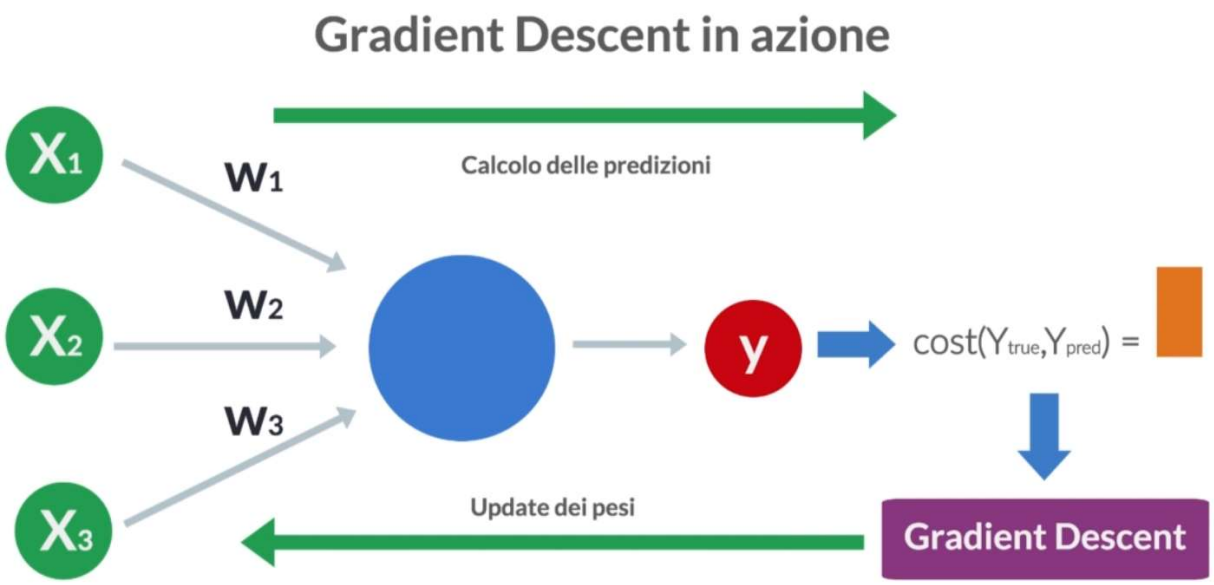
Molto dinamico

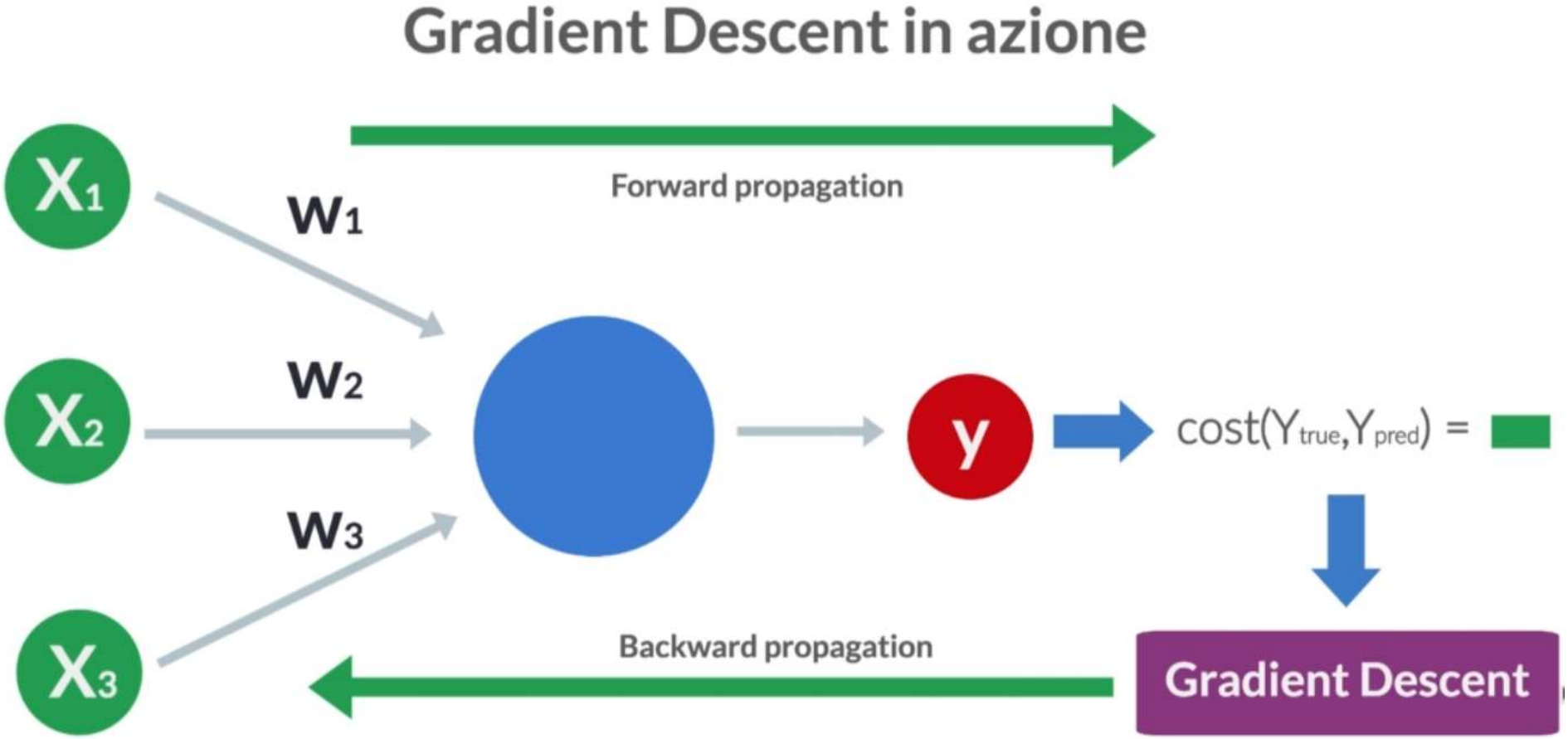
Per aggiornare il modello con nuovi dati è possibile eseguire uno step del SGD solo su questi dati

Meno soggetto al problema dei minimi locali

Grazie alle fluttuazioni della funzione di costo potrebbe sfuggire ai minimi locali

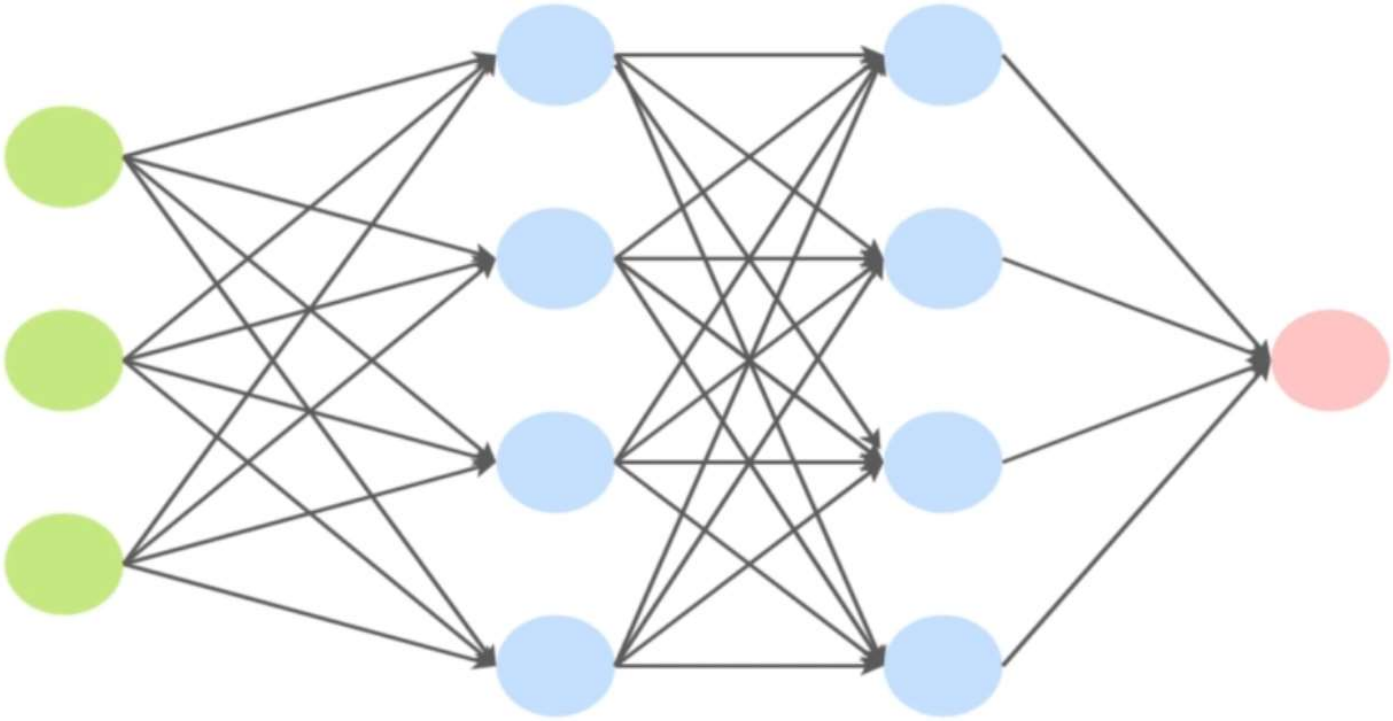

Gradient Descent in azione
X1
X2
X3
W1
W2
W3
y
cost(Ytrue,Ypred) =
Gradient Descent


Gradient Descent in azione
Calcolo delle predizioni
X1
X2
X3
W1
W2
W3
y
cost(Ytrue,Ypred) =
Update dei pesi
Gradient Descent

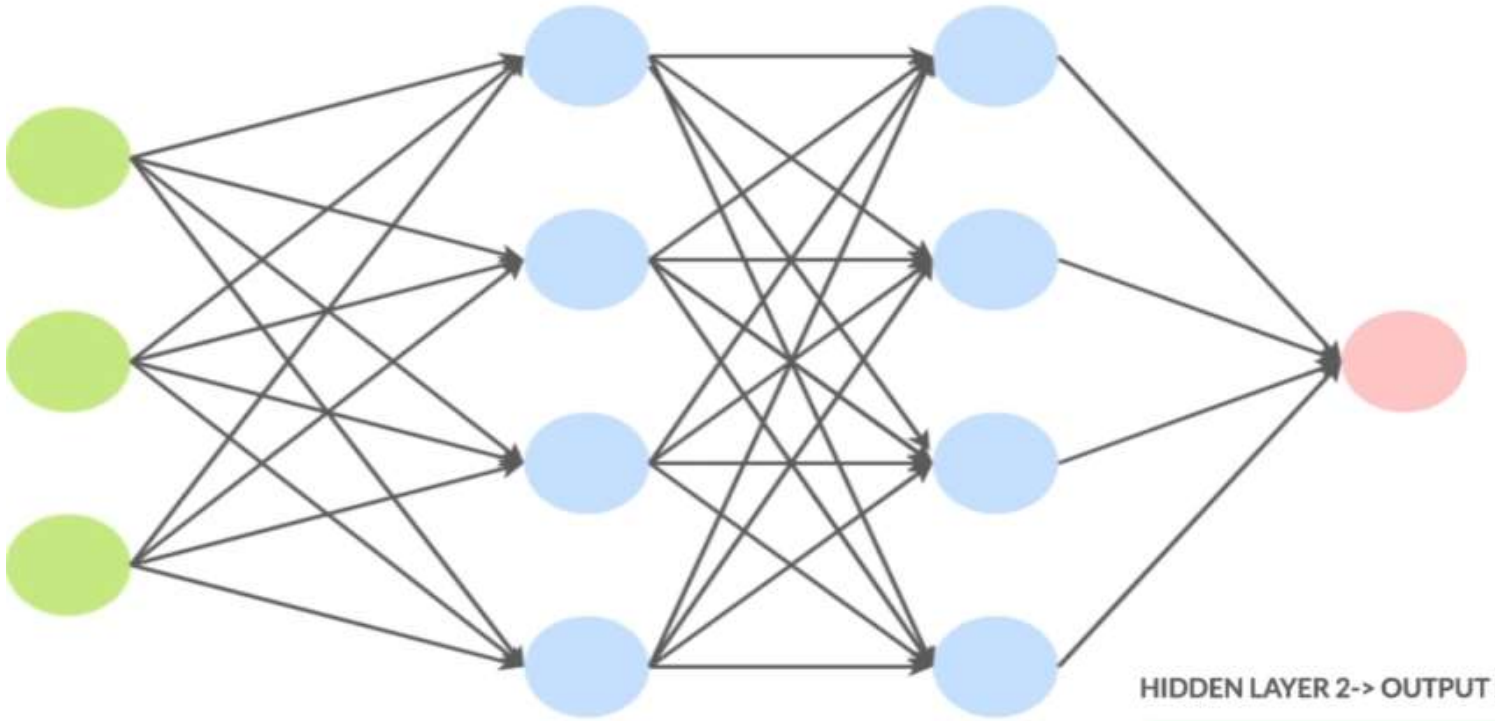


Backpropagation

Permette di sapere quanto ogni peso di ogni strato
ha contribuito all'errore del modello
propagando l'errore dell'ultimo strato all'indietro.



Una rete neurale è formata da più funzioni composte,
ognuna con i propri coefficienti, che mappano uno strato al successivo



INPUT LATER -> HIDDEN LAYER 1

$$\mathbf{z_1} = \mathbf{W}_1\mathbf{x} + \mathbf{b_1}$$

$$\mathbf{a_1} = \text{relu}(\mathbf{z}_1)$$

HIDDEN LAYER 1 -> HIDDEN LAYER 2

$$\mathbf{z_2} = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b_2}$$

$$\mathbf{a_2} = \text{relu}(\mathbf{z}_2)$$

HIDDEN LAYER 2-> OUTPUT LAYER

$$\mathbf{z_3} = \mathbf{W}_3\mathbf{a}_2 + \mathbf{b_3}$$

$$\mathbf{a_3} = \text{sigmoid}(\mathbf{z}_3)$$

$$\hat{y} = \text{step}(\mathbf{a_3})$$

$$\mathbf{z_1} = \mathbf{W}_1\mathbf{x} + \mathbf{b_1}$$

$$\mathbf{a_1} = \text{relu}(\mathbf{z}_1)$$

$$\mathbf{z_2} = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b_2}$$

$$\mathbf{a_2} = \text{relu}(\mathbf{z}_2)$$

$$\mathbf{z_3} = \mathbf{W}_3\mathbf{a}_2 + \mathbf{b_3}$$

$$\mathbf{a_3} = \text{sigmoid}(\mathbf{z}_3)$$

Chain rule

Permette di calcolare derivate di funzioni annidate

$$f(x) = f(g(x))$$

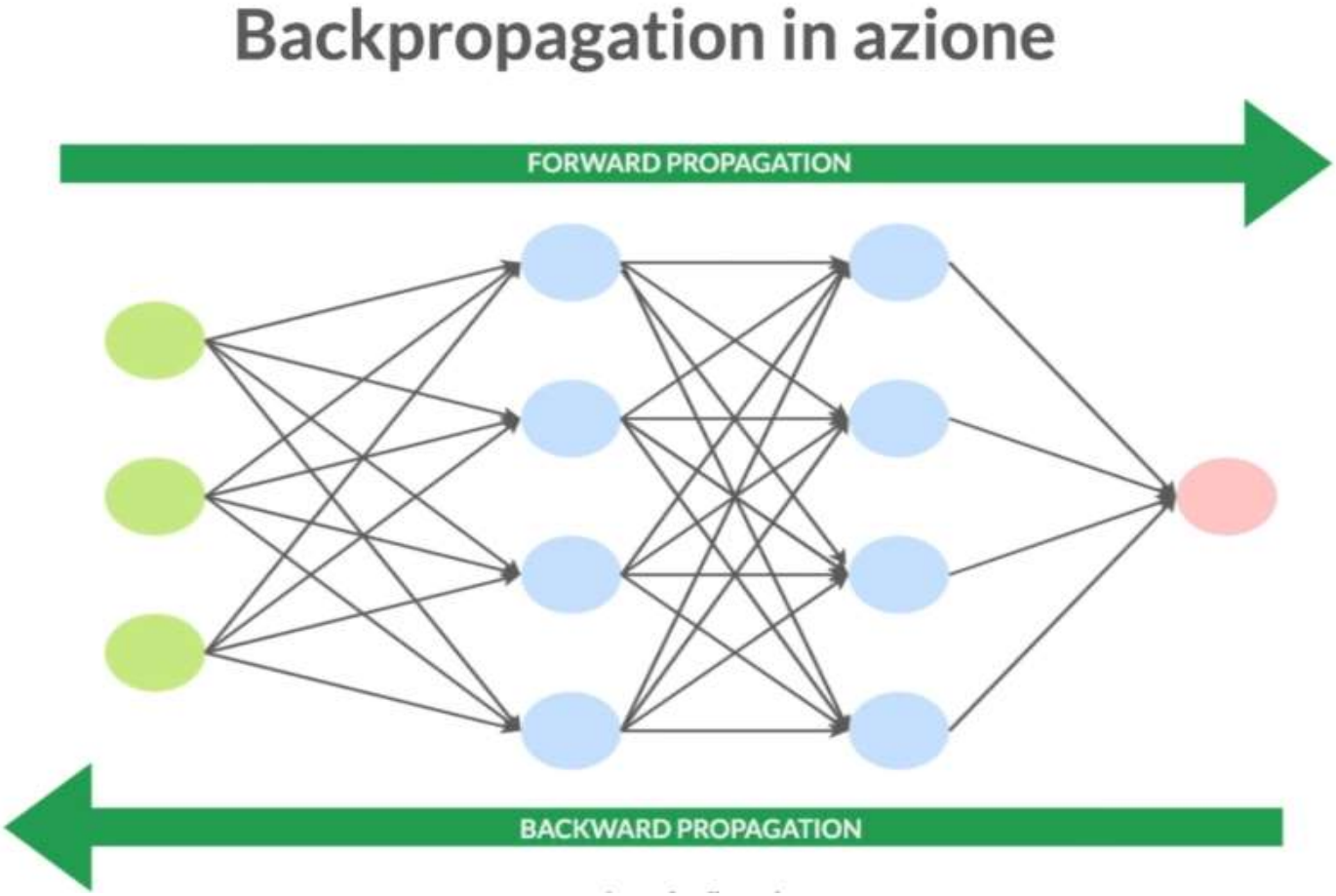
$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

$$\mathrm{J}(w, b) = J(\phi(wx + b))$$

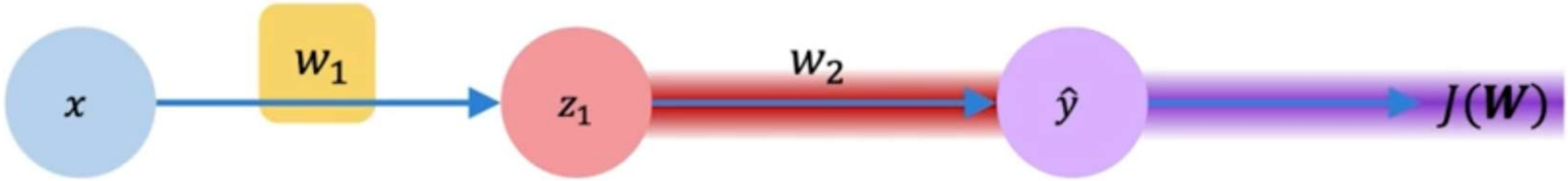
$$\nabla \mathrm{J}(w, b) = [\frac{d\mathrm{J}}{dw}, \frac{d\mathrm{J}}{db}]$$

$$\frac{d\mathrm{J}}{dw} = \frac{dJ}{d\phi}\frac{d\phi}{d\mathrm{z}}\frac{d\mathrm{z}}{dw}$$

$$\frac{d\mathrm{J}}{db} = \frac{dJ}{d\phi}\frac{d\phi}{d\mathrm{z}}\frac{d\mathrm{z}}{db}$$



Computing Gradients: Backpropagation

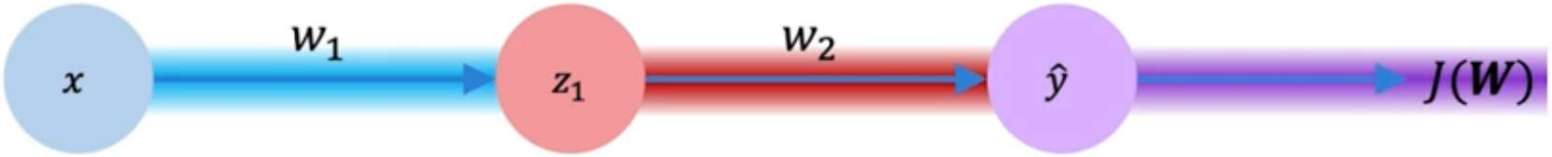


$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Repeat this for ***every weight in the network*** *using gradients from later layers*

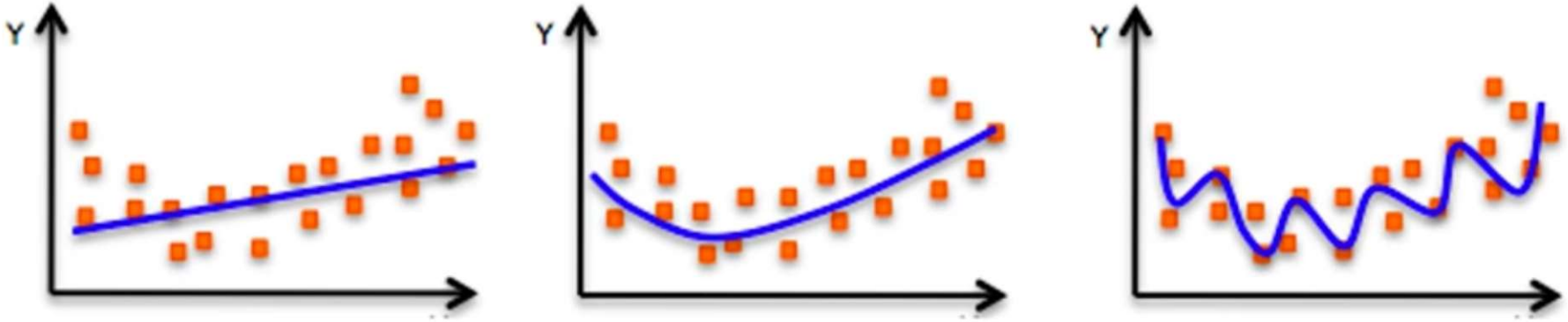
```
def forward_propagation(X):

    z1 = dot(self.W1,X.T)+self.b1
    a1 = relu(z1)
    z2 = dot(self.W2,a1)+self.b2
    a2 = sigmoid(z2)

    return a2
```

```
def backward_propagation(X,a2,y):

    m = a1.shape[1]
    dz2 = a2-y
    dW2 = dot(a1.T,dz2)/m
    db2 = sum(dz2,axis=0)

    m = X.shape[1]
    dz1 = dot(dz2,self.W2.T)*relu_derivative(z1)
    dW1 = dot(X.T,dz1)/m
    db1 = sum(dz1,axis=0)

    return dW1, db1, dW2, db2
```

Gradient Descent Algorithms

Algorithm	TF Implementation	Reference
• SGD	`tf.keras.optimizers.SGD`	Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.
• Adam	`tf.keras.optimizers.Adam`	Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.
• Adadelta	`tf.keras.optimizers.Adadelta`	Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.
• Adagrad	`tf.keras.optimizers.Adagrad`	Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.
• RMSProp	`tf.keras.optimizers.RMSProp`	

The Problem of Overfitting



Underfitting
Model does not have capacity to fully learn the data

Ideal fit

Overfitting
Too complex, extra parameters, does not generalize well