

Biologically Inspired Computation

Dr Marta Vallejo
m.vallejo@hw.ac.uk

Lecture 2

Gradient Descent

- Learning Algorithms in Neural Networks
- Gradient Descent: General Idea
- Cost Function
 - For linear regression
 - For logistic regression (classification)
- Logistic Regression/Linear Regression Architecture
- Gradient Descent: Learning Rules
- Gradient Descent: The Learning Rate
- How to code Gradient Descent
 - For a single sample
 - For m samples
- Online – Offline Learning

Learning Algorithms in Neural Networks

A neural network is only a model, a way of representing information. In order to add intelligence to our model, we use different types of learning algorithms:

Historical Learning Paradigms:

- Hebbian Learning
- Delta Learning Rule
- Widrow-Hoff Learning Rule

} More in Additional Material 2

Recurrent Neural Networks:

- Hopfield

Feed-forward networks:

- Backpropagation (based on Gradient Descent)
- Neuroevolution (Evolutionary Algorithms)

Non-Supervised methods: (not covered in this class)

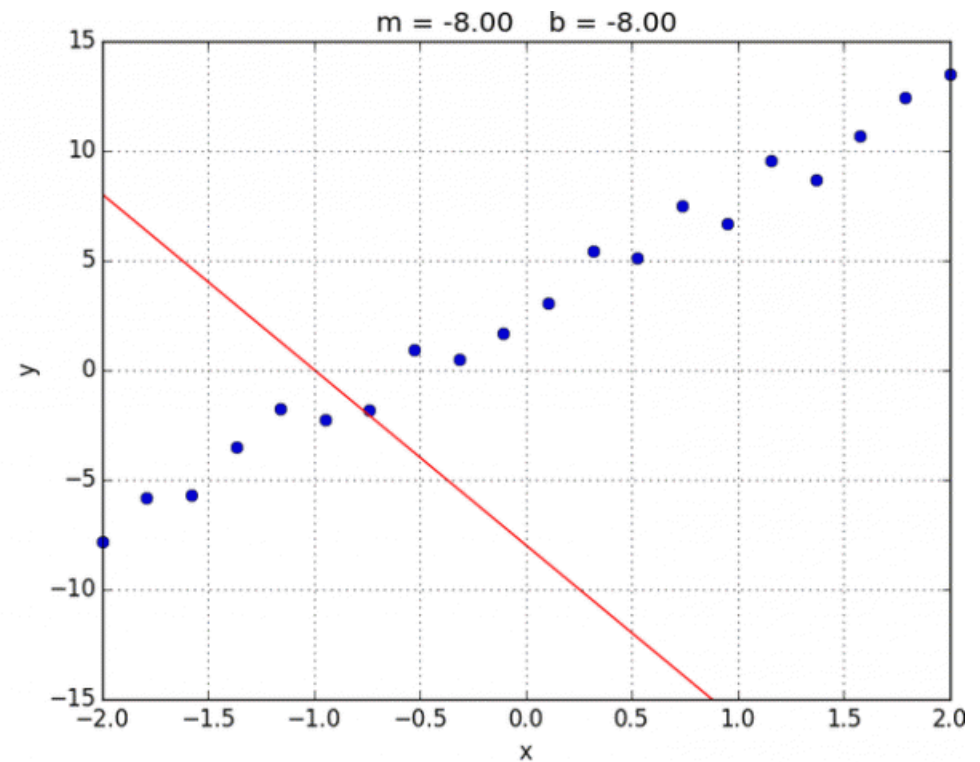
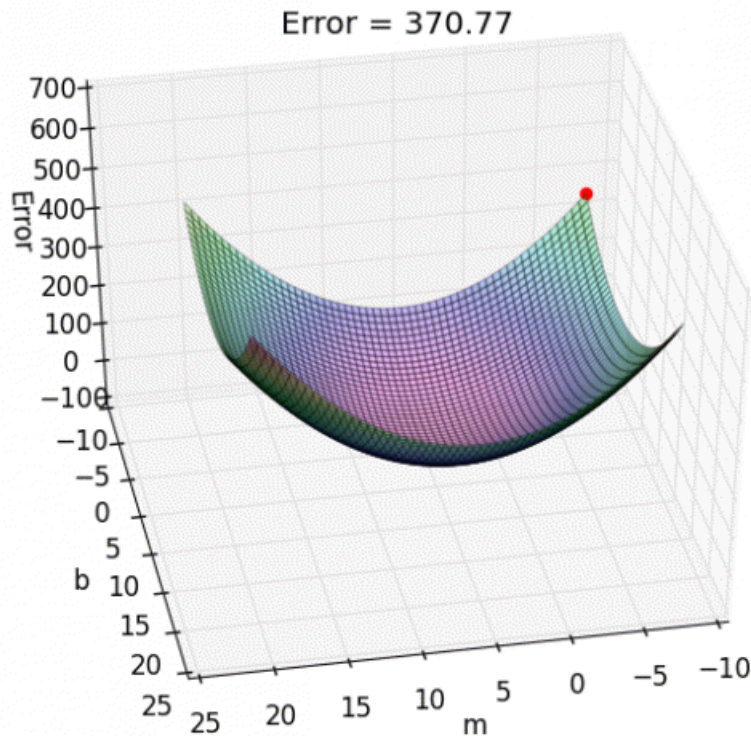
- Reinforcement Learning
- Kohonen's Learning Laws

Gradient Descent: General Idea

Gradient descent is a **local search** optimisation method that allows us to find a local minimum of a function. It is used as a basis for backpropagation. For example, in a regression problem, we want to find the linear model that best fit our data.

Cost Function: is a measure of how wrong the model is.

$$y = mx + b$$



We want to find the parameters \mathbf{m} (we call it w later on) and \mathbf{b} to minimise the distance between each point and our prediction. For that, we use gradient descent.

Cost Function

Given a training set with m samples

$$\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$$

where the superscripts
are the number of
each sample:

Our **cost function** J will be:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m L(y^{(i)}, \hat{y}^{(i)})$$

The averaged error **L** between the data **y** and the predictor **\hat{y}** This is called the **loss function**.

The loss function measures the discrepancy between the prediction $\hat{y}^{(i)}$ and the desired output $y^{(i)}$, but the way is calculated depends on the type of problem into consideration.

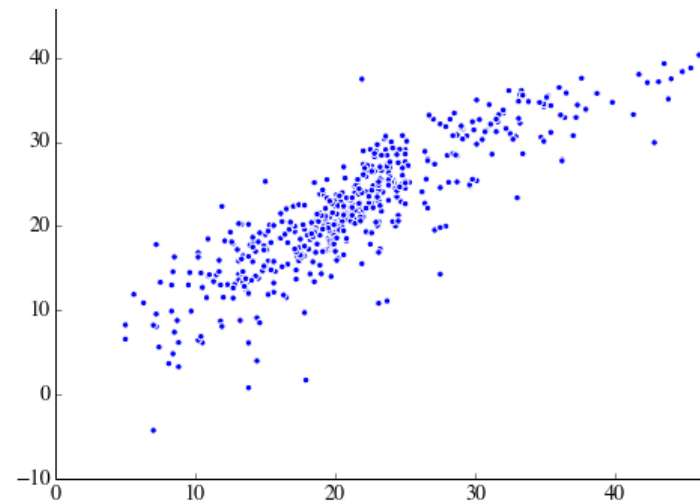
- Regression
- Classification

Linear Regression Problem

Let's say that we want to predict the housing cost in function of the property size. To do that we have a dataset with prices and the corresponding size.

House Price in \$1000s (y)	Square Feet (x)
245	1400
312	1600
279	1700
308	1875
199	1100
219	1550
405	2350
324	2450
319	1425
255	1700

If we plot each sample: (note that this is a visual example, the values of the axis don't match the table)



Taken a look to the plot, we decide that the model that best fits our data is a linear model. Something like that:

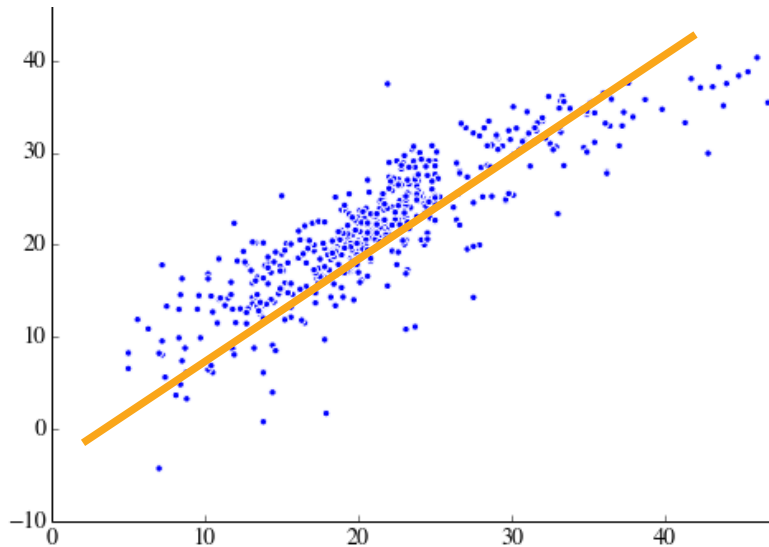
$$\hat{y} = w x + b$$

x: feature (size)
 \hat{y} : predictor(price)

By solving the equation the model gives us a predicted price when we select a value for the size of the property

Linear Regression Problem

We are looking for a linear model with this behaviour:



To do that, we should find the values of **w** and **b** that minimise the distance of each sample to our predict model

Our goal is:

$$y \approx \hat{y}$$

Minimise the distance between y and \hat{y}

Note that for real problems we normally deal with a much large number of features, like number of bedrooms, bathrooms, type of heating system....

$$w_0 x_0 + w_1 x_1 + \dots + w_n x_n + b$$

The representation of our model will be:



Recap

Remember that our goal is to choose a loss function L to calculate our cost J :

$$\underbrace{J(w, b)}_{\text{2nd}} = \frac{1}{m} \sum_1^m \underbrace{L(y^{(i)}, \hat{y}^{(i)})}_{\text{1st}}$$

The loss L : For each sample

The cost J : The average of all the losses

Loss Functions for Linear Regression

1.- Mean Squared Error/Squared Loss

Easy to understand and implement.
Differentiable. Very sensitive to outliers

$$L(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{2} (y^{(i)} - \hat{y}^{(i)})^2$$

2.- Absolute Loss

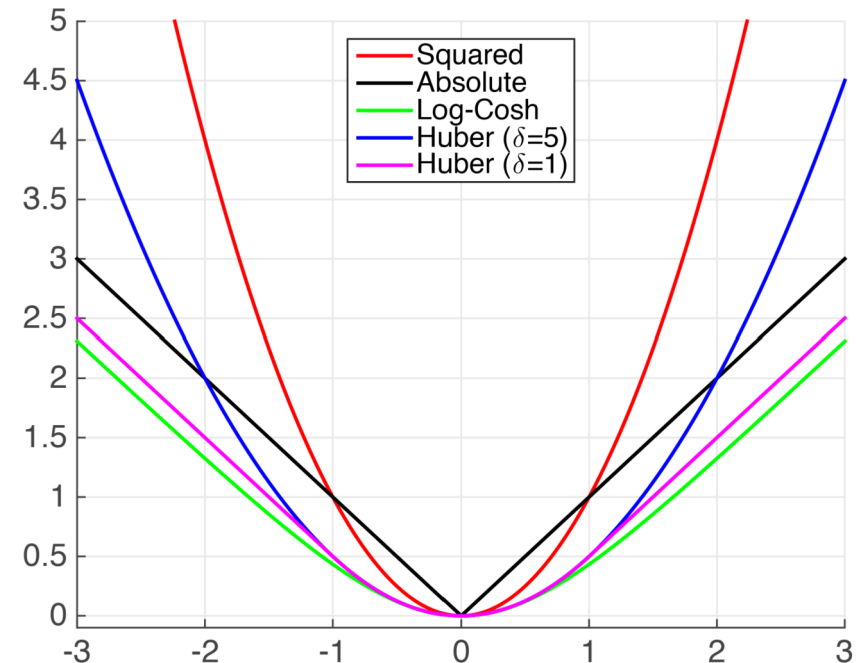
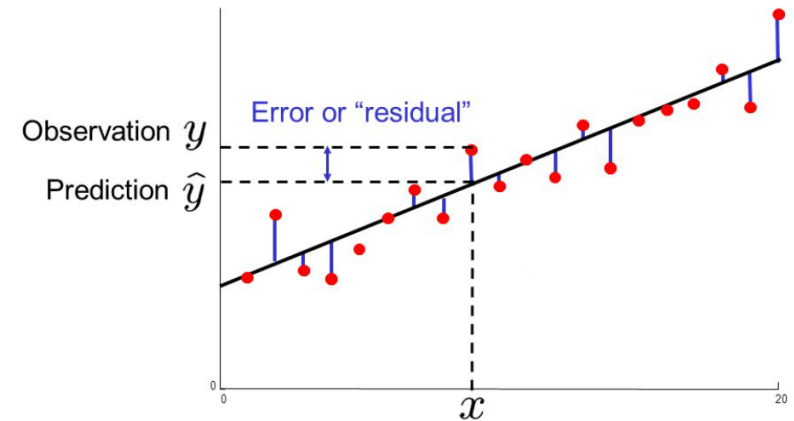
Robust to the presence of outliers but not differentiable (no suitable for Gradient Descent)

$$L(y^{(i)}, \hat{y}^{(i)}) = |y^{(i)} - \hat{y}^{(i)}|$$

3.- Huber Loss

Differentiable approximation to absolute loss
Quadratic for values $\leq \delta$ and linear for $> \delta$
Robust to the presence of outliers.

$$L(y^{(i)}, \hat{y}^{(i)}) = \begin{cases} \frac{1}{2} (y^{(i)} - \hat{y}^{(i)})^2 & \text{for } |y^{(i)} - \hat{y}^{(i)}| \leq \delta \\ \delta |y^{(i)} - \hat{y}^{(i)}| - \frac{1}{2} \delta^2 & \text{otherwise} \end{cases}$$



Classification Problem: Logistic Regression

Let's say that we want to discriminate between photos of cats and dogs

To do that we have a database file with the photos and the corresponding label (cat/dog)

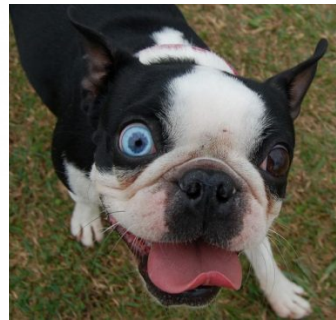


Cat



Dog

If we have an unseen sample:

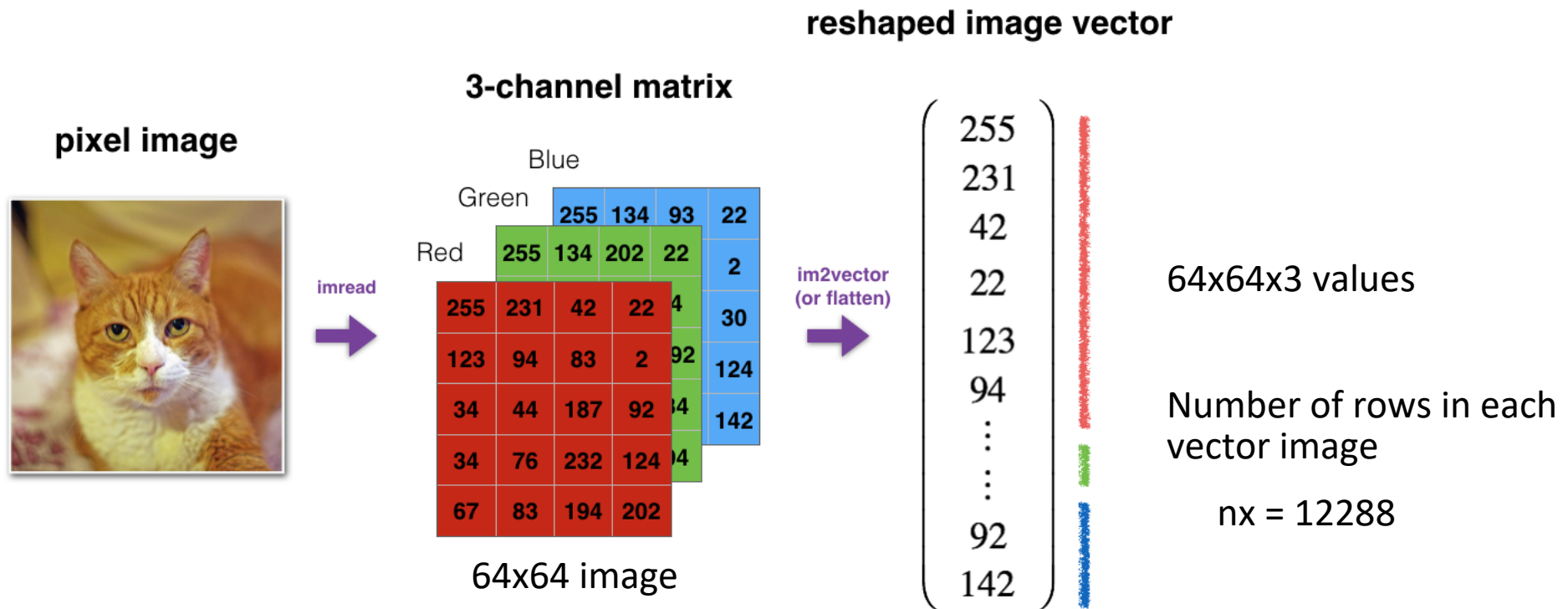


We have to decide if it is a cat or a dog

Our answer is **categorical**

Logistic Regression for Image Recognition

Given an image we transform it in a feature vector x by unrolling its pixel intensity values in a single column vector x .



We add all the column vectors (one per image in our training data) together in a matrix. If we have m images in our training data, the size of our **input data** will be: $X_{(n_x, m)}$

$$X = \begin{bmatrix} | & & | & & | \\ x^{(1)} & \dots & x^{(m-1)} & & x^{(m)} \\ | & & | & & | \end{bmatrix}$$

Definition of the Problem

The algorithm will evaluate the **probability** of a dog being in the image. Remember that a probability value is a number between 0 and 1, where 1 represents that the image is 100% a dog and 0 that is 0% probable.

Given x :

$$\hat{Y} = P(y=1|x) \quad \text{where } 0 \leq \hat{y} \leq 1$$

$P(y=1|x)$ tells us what is the probability of y being equal to 1 (a dog) given the input vector of features x

This is called
conditional probability

Parameters:

X is the input feature vector where n_x is the number of features (in our case the number of pixels intensity values)

y the training label $y \in 0,1$ (0 dog, 1 cat)

\hat{Y} is the predicted output that depends on the parameters w and b .

$$X = \begin{bmatrix} | & & | & & | \\ x^{(1)} & \dots & x^{(m-1)} & & x^{(m)} \\ | & & | & & | \end{bmatrix}$$

Definition of the Problem

To model \hat{Y} (conditional probability), we can think in using something similar to the regression problem:

$$w_0 x_0 + w_1 x_1 + \dots + w_n x_n + b$$

where $w = \{w_0, w_1, \dots, w_n\}$ and b tell us how important is the associate pixel x_i for deciding if the image is a dog. If we express this as a matrix multiplication

$$z = W^T X + b$$

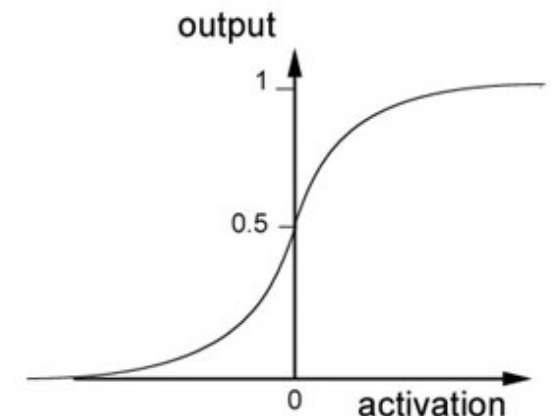
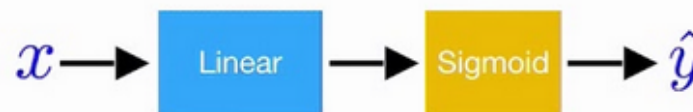
In order to be able to multiply these two matrices, we need to use the transpose of W

However this gives us a real number and not a value between 0 and 1. Then we need to use a function to transform this output to a value in the range of 0 and 1

Sigmoid/Logistic function:

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

$$\hat{y} = \sigma(W^T X + b) = \sigma(z)$$



Recap again

Remember that our goal is to choose a loss function L to calculate our cost J :

But now for approaching a **classification** problem. J is the same, but we need to use other types of loss function

$$\underbrace{J(w, b)}_{\text{2nd}} = \frac{1}{m} \sum_1^m \underbrace{L(y^{(i)}, \hat{y}^{(i)})}_{\text{1st}}$$

The loss L : For each sample

The cost J : The average of all the losses

Cross Entropy Loss Function

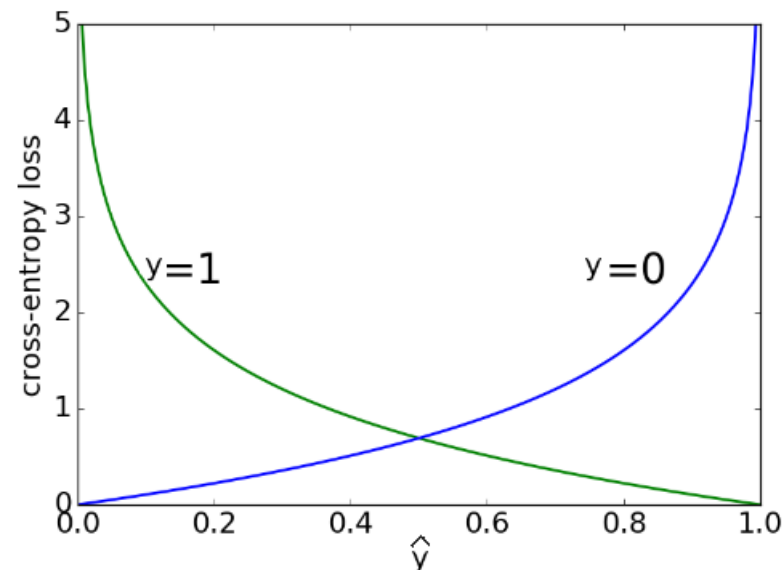
1.- Cross Entropy / Log Loss

For a two-class classification problem, the cross entropy for a sample i is:

$$L(y^{(i)}, \hat{y}^{(i)}) = -(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))$$

It returns a probability value between 0 and 1.

A perfect model would have a log loss of 0. This is the behaviour of the function:



Behaviour of Cross Entropy Loss Function

$$L(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

- If $y=1$ then \hat{y} should be close to 1. If we substitute y by 1

$$L(y, \hat{y}) = -\log(\hat{y}) = -\log(\approx 1) \approx 0$$

- If $y=1$ but \hat{y} is close to 0

$$L(y, \hat{y}) = -\log(\hat{y}) = -\log(\approx 0) \approx \infty$$

- If $y=0$ then \hat{y} should be close to 0. If we substitute y by 0

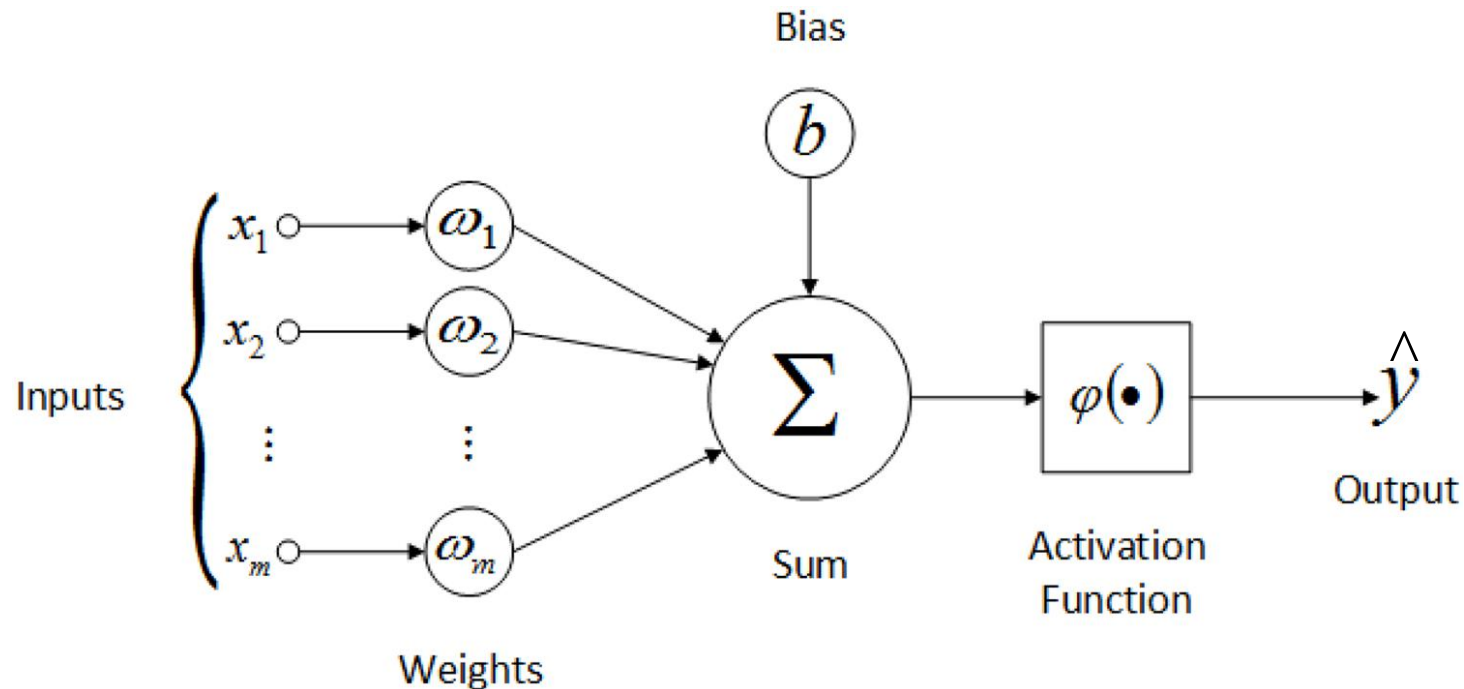
$$L(y, \hat{y}) = -\log(1 - \hat{y}) = -\log(\approx 1) \approx 0$$

- If $y=0$ but \hat{y} is close to 1

$$L(y, \hat{y}) = -\log(1 - \hat{y}) = -\log(\approx 0) \approx \infty$$

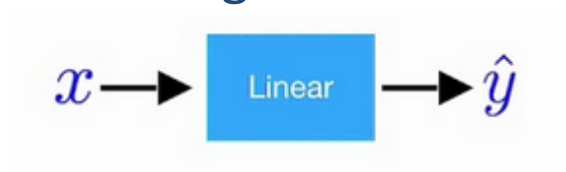
Logistic Regression Architecture

As a summary of previous slides, this is a schema of the entire process, which corresponds to a Neural Network with a single node

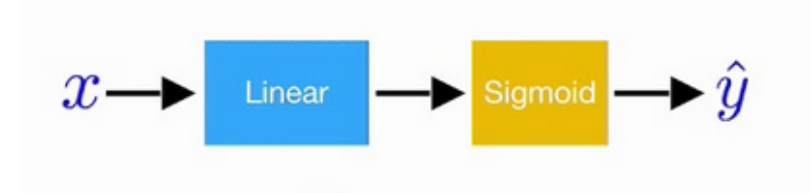


However, we saw that the steps are different according to the problem into consideration

Regression



Classification

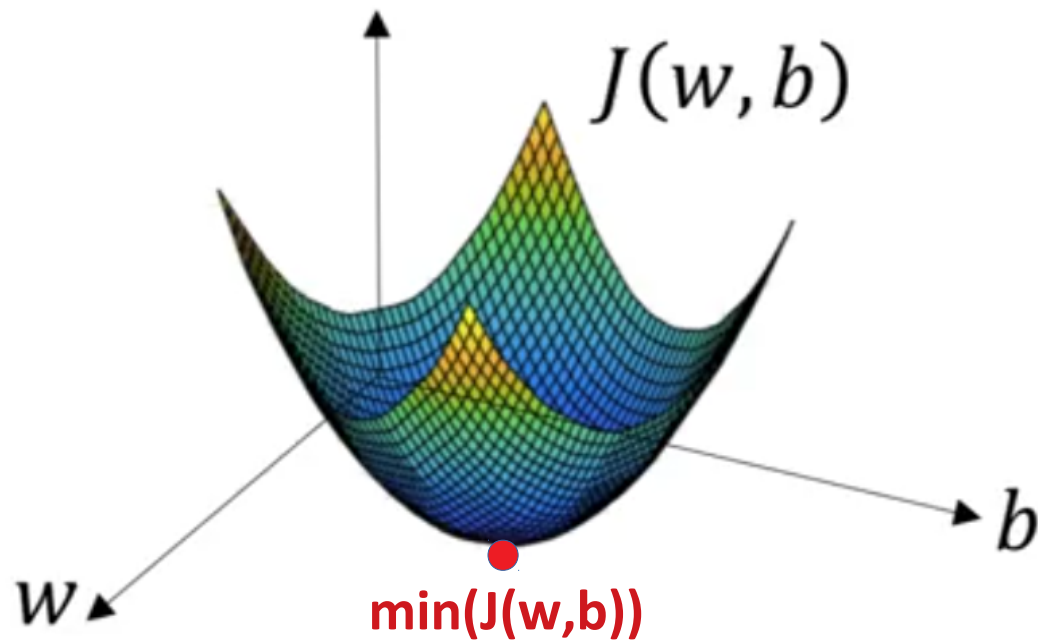


This is because the transformation can be different depending on different factors. In regression will use a linear function and in classification a sigmoid function

Gradient Descent

Supposing that we are going to solve a classification problem using the cross-entropy loss function

If this is a graphical representation of the cost $J(w,b)$, considering w as a single number



We want to find this place

Initially it evaluates a random point (random w and b values)

Gradient descent is a **local search** algorithm. This means that, it can only find the next **local optima**.

However since the cross-entropy loss function is always convex, there is only a single global minima. Then, does not matter where we start the search, the algorithm would always leads us towards the minimum.

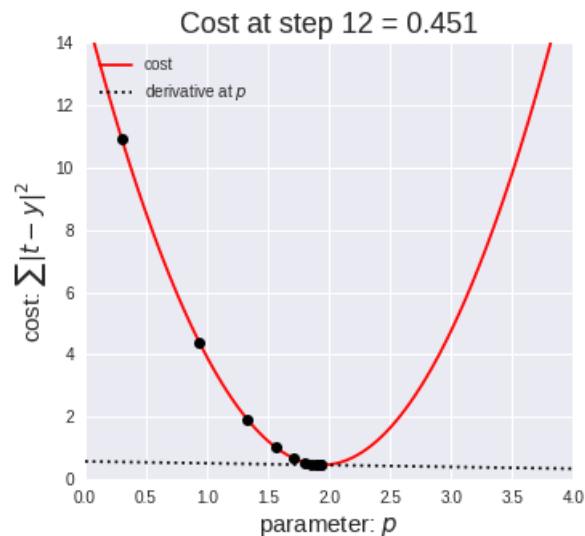
Gradient Descent: Learning Rules

Algorithm:

```
repeat until convergence{  
    minimising the cost by updating w and b  
}
```

But which rule are we going to follow to update the parameters w and b?

It uses the **gradient (derivative)** to decide the direction in which the function tends to a minimum: the steepest down direction. One per iteration.



Learning Rule for each parameter:

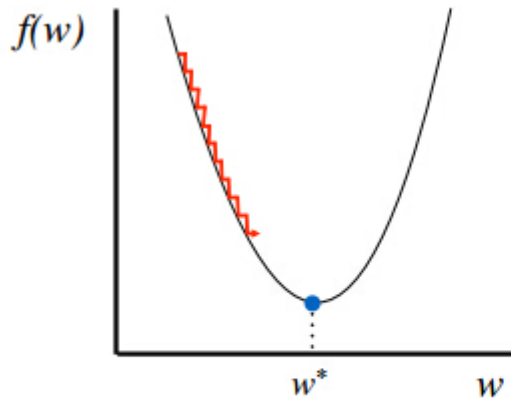
$$w_1 := w_1 - \alpha \frac{\partial J(w, b)}{\partial w}$$

$$b := b - \alpha \frac{\partial J(w, b)}{\partial b}$$

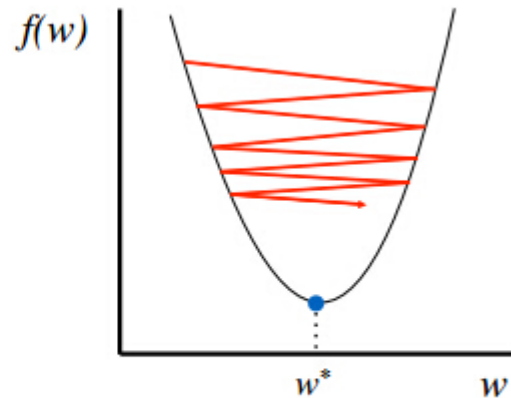
α Learning rate: controls the size of the step in each iteration

Gradient Descent: α learning rate

The learning rate is a hyperparameter of Gradient Descent



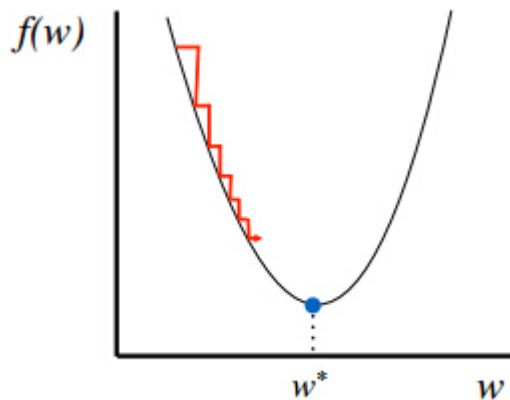
Too small: converge very slowly



Too big: overshoot and even diverge

- The learning rate determine sthe size of the jump between iterations. It is important to adjust appropriately its value to avoid both slowness and divergence.

Values from 0.1 to 0.0001

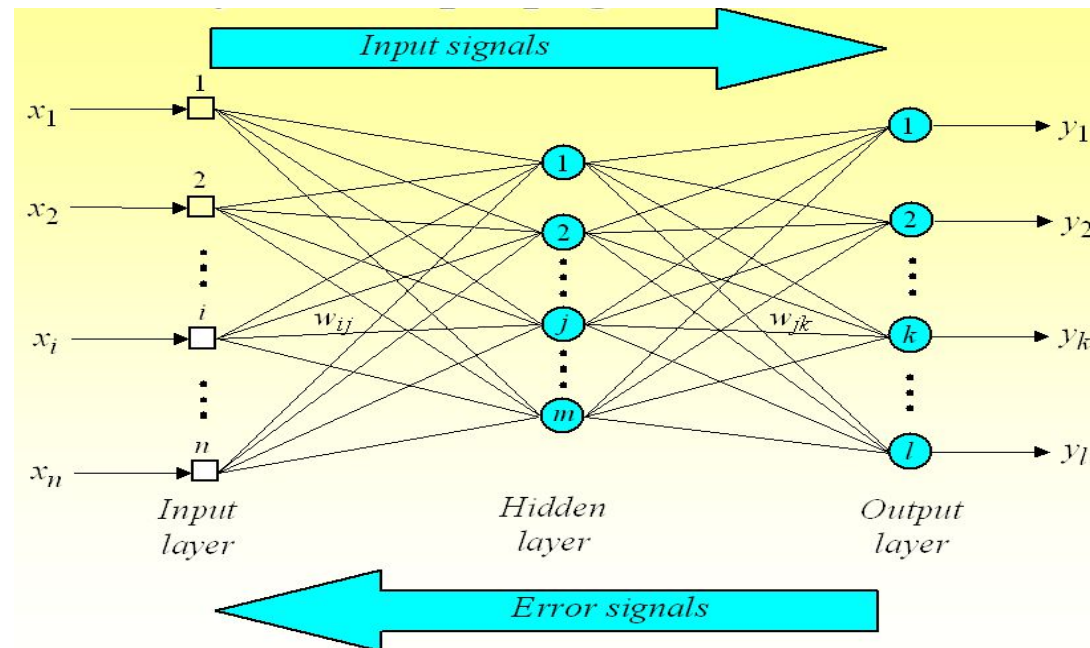


Reduce size over time

The learning rate can be reduced over the time, proportional the gradient or the iteration to be more accurate when we are close to the minimum.

Backpropagation

Forward propagation: An input vector propagates through the network.

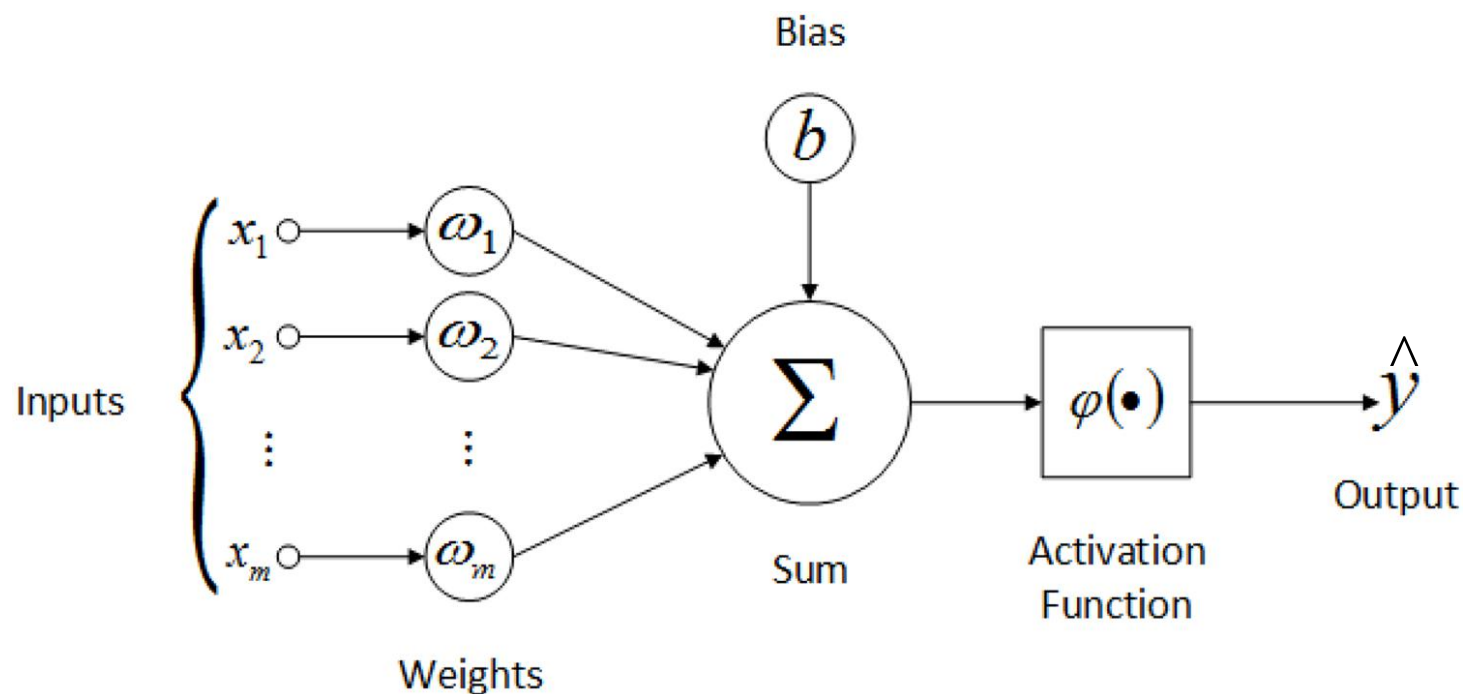


Weight update (**backpropagation**): the weights of the network will be changed in order to decrease the signal error of each neuron

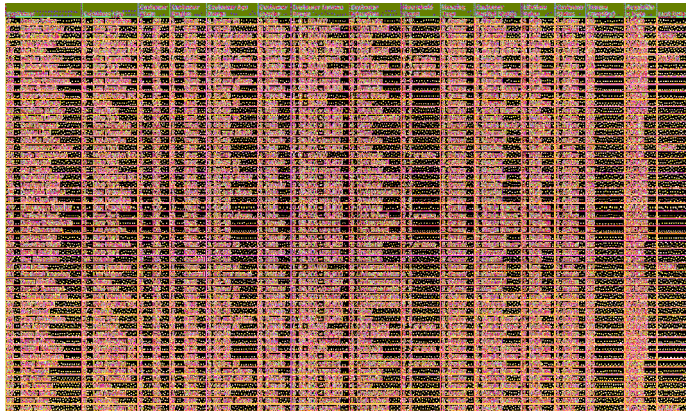
Backpropagation is a **gradient descent algorithm**. We are going to convert the training problem of an ANN into an optimisation problem.

Before some code details... a summary about what we need/have

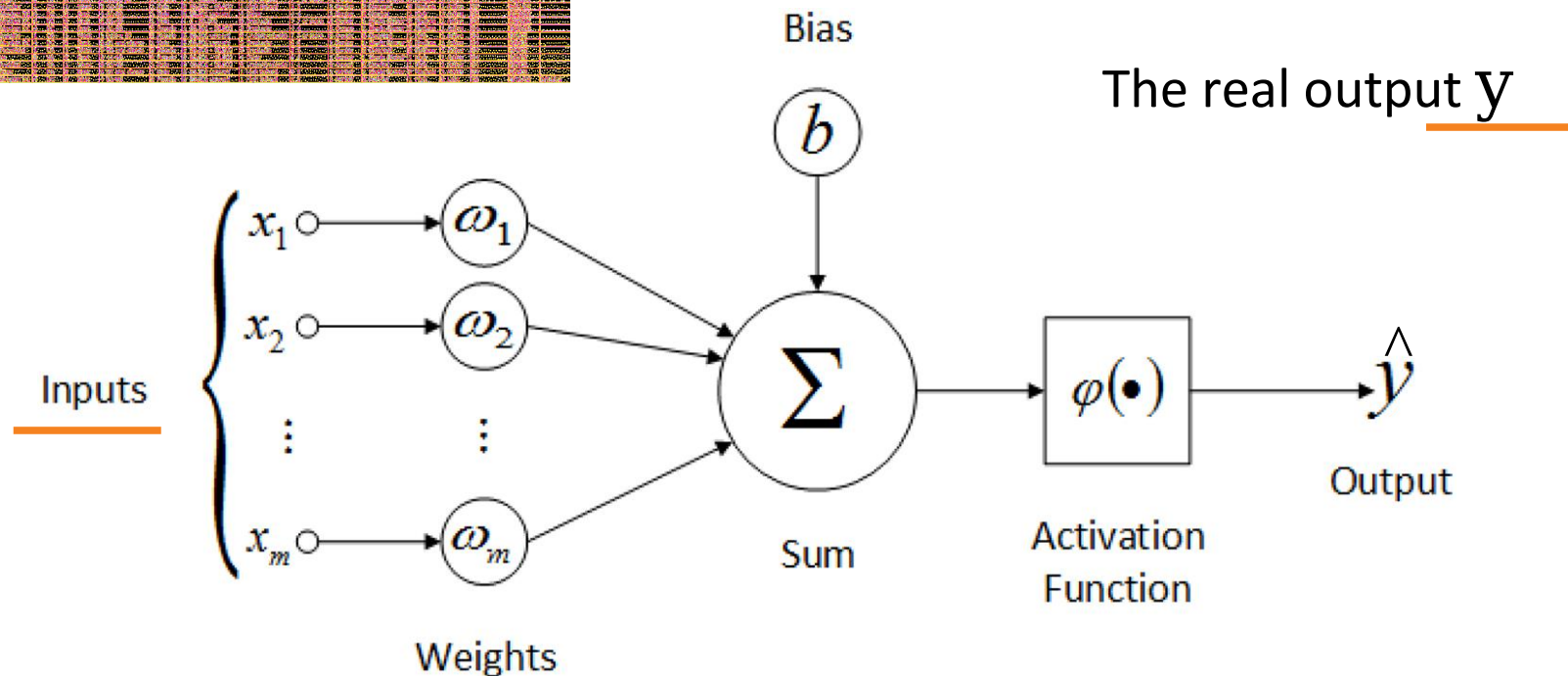
Taking a look again to our architecture. Which information we have?



Before some code details... a summary about what we need/have



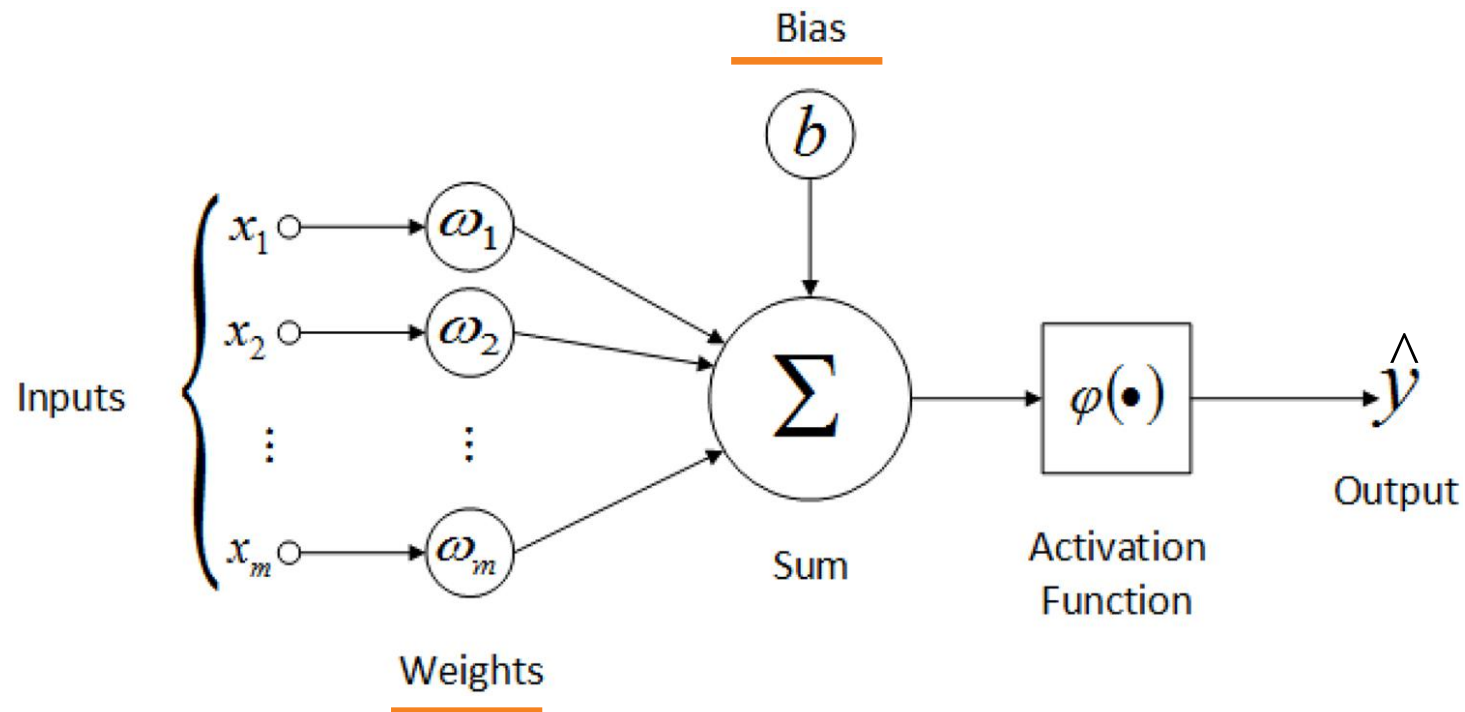
Beforehand, we only have the information included into our training data



One set of inputs and one output for each row in our dataset

Before some code details... a summary about what we need/have

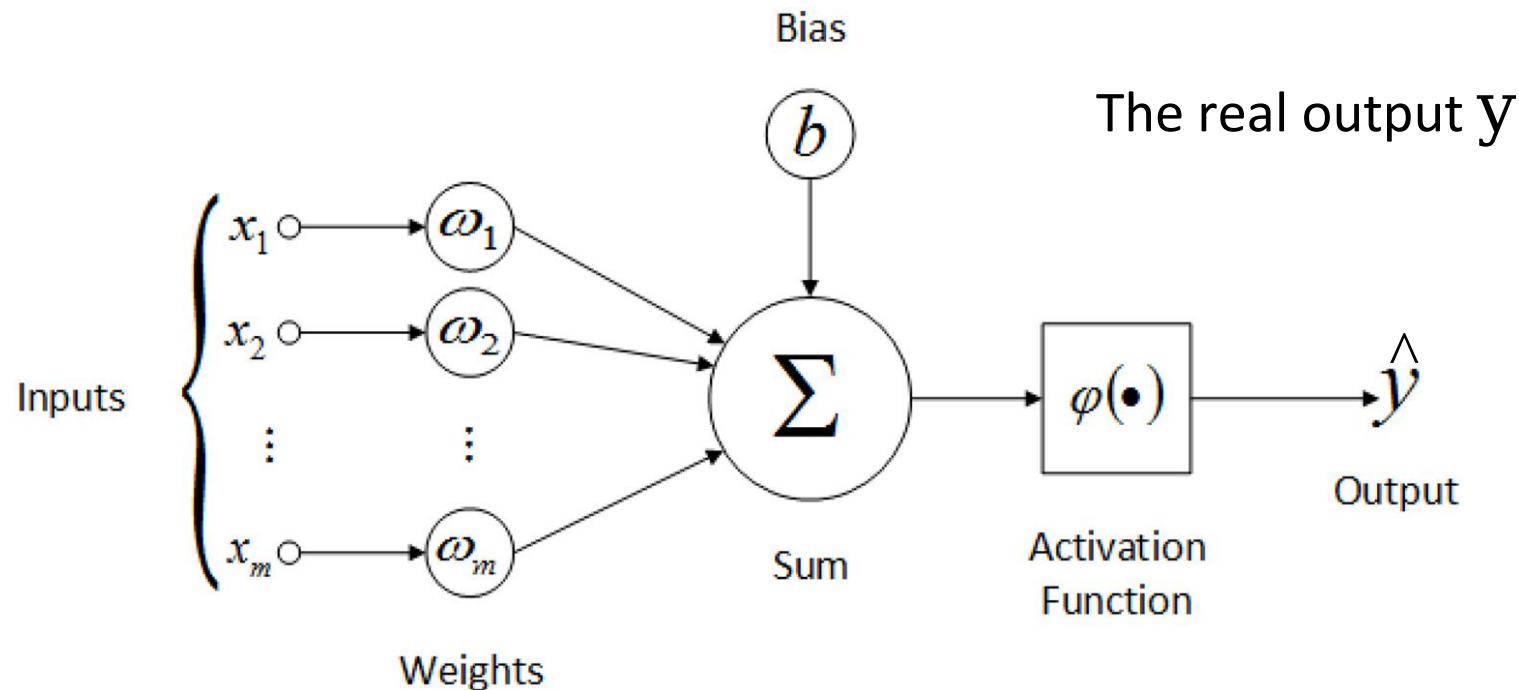
What we need to calculate?



How should be the values of these parameters?

Before some code details... a summary about what we need/have

How should be the values of these parameters?



$$y \approx \hat{y}$$

And then, applying Gradient Descent, what we need to calculate in each iteration?

Before some code details... a summary about what we need/have


And then, applying Gradient Descent, what we need to calculate in each iteration?

The learning rules:

$$w_1 := w_1 - \alpha \frac{\partial J(w, b)}{\partial w}$$

One per each input feature

$$b := b - \alpha \frac{\partial J(w, b)}{\partial b}$$

- 
- Deciding the value for the learning rate α
 - Deciding the best loss function to use
 - Initialising the weight and the bias randomly
 - Calculating the derivatives of the loss function
 - Running the loop until convergence

We will start seeing how to compute the derivatives

Gradient Descent: for a single sample

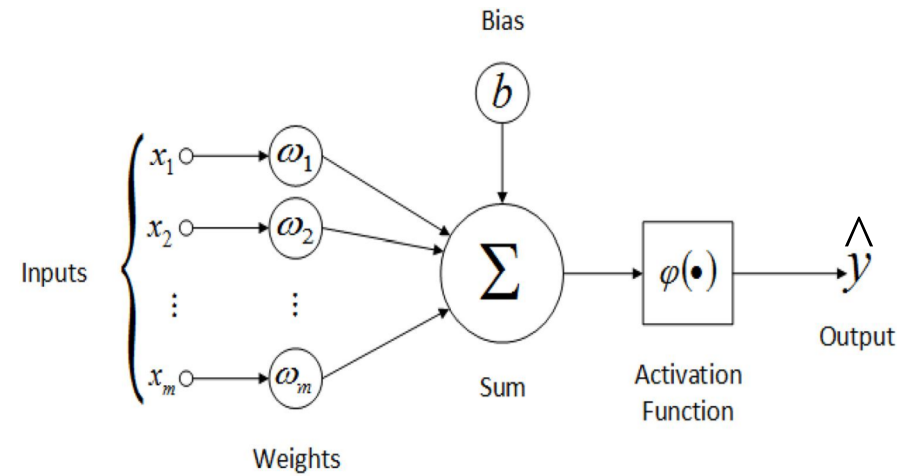
We know that for one sample in a classification problem:

$$z = W^T X + b$$

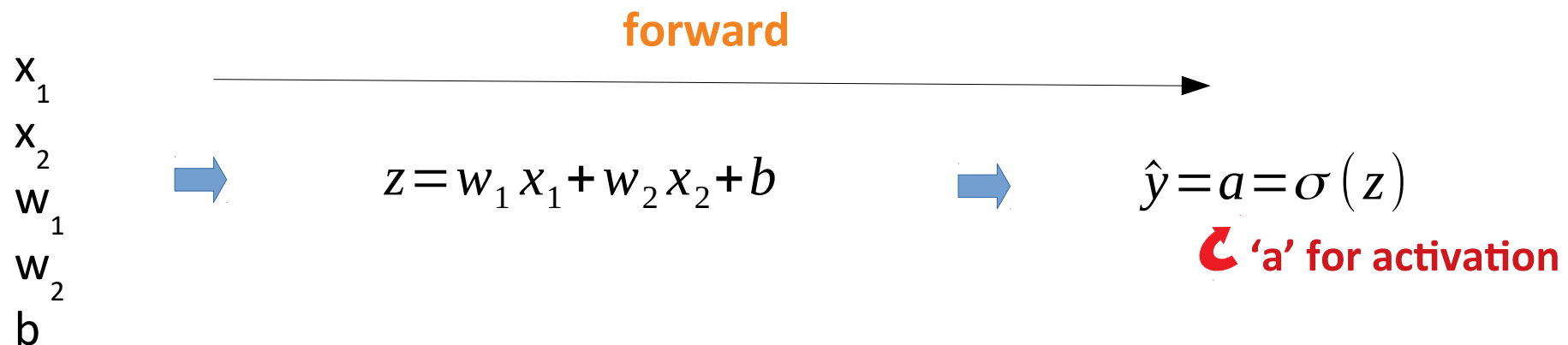
$$\hat{y} = a = \sigma(z)$$

$$L(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

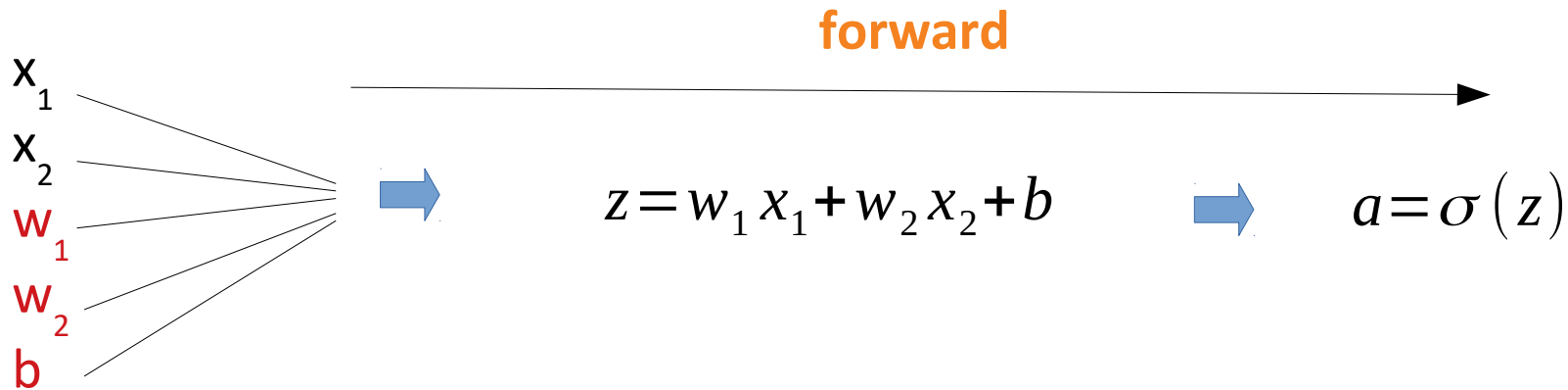
For this example, we are going to have only two features x_1 and x_2 . Then, this means that we need to compute w_1 , w_2 and b .



In this forward pass we are going to compute \hat{Y}



Gradient Descent: for a single sample



After calculating \hat{Y} , that in this case is also a , we need to see how far is our predict from the real one Y . To do that we calculate the loss $L(y, \hat{y})$

Notice that for one sample the loss function L is equivalent to the cost function J

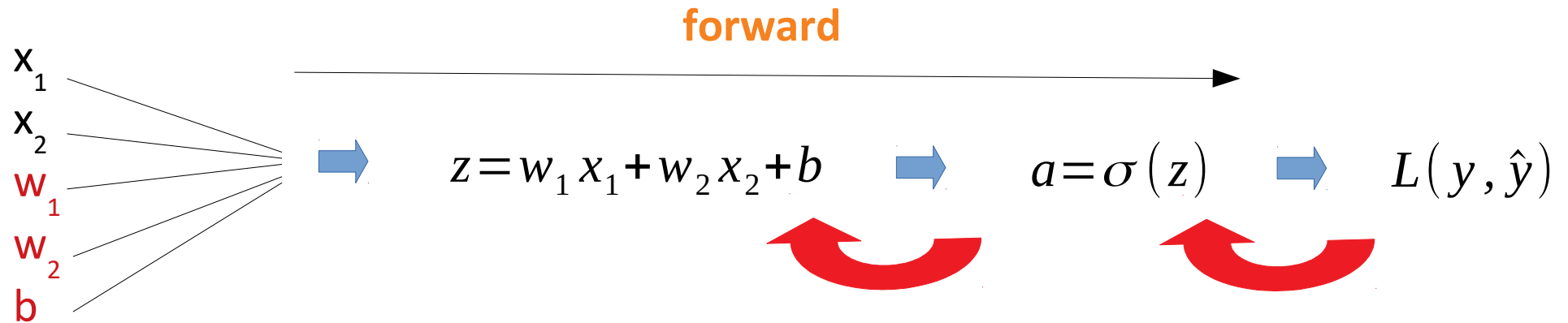
$$J(y, \hat{y}) = J(y, a) = -(y \log(a) + (1 - y) \log(1 - a))$$

Based on this value, we need to go backwards and calculate how much each factor w_1 and w_2 contribute to this difference. To do that we use the derivatives.

$$dw_1 = \frac{\partial J(y, a)}{\partial w_1} \quad dw_2 = \frac{\partial J(y, a)}{\partial w_2}$$

However, we don't know how much the cost function varies in function of w_1 , w_2 and b .

Gradient Descent: for a single sample



With the information we have, we can calculate how the cost function varies in relation to our predict \hat{Y} , that in this case is the same than a .

$$J(y, \hat{y}) = J(y, a) = -(y \log(a) + (1 - y) \log(1 - a))$$

$$da = \frac{\partial J(y, a)}{\partial a} = \frac{\partial -(y \log(a) + (1 - y) \log(1 - a))}{\partial a}$$

$$da = \frac{-y}{a} + \frac{1 - y}{1 - a}$$

Applying the chain rule once:

$$dz = \frac{\partial J(y, a)}{\partial z} = \frac{\partial J(y, a)}{\partial a} \frac{\partial a}{\partial z} = da \frac{\partial a}{\partial z}$$

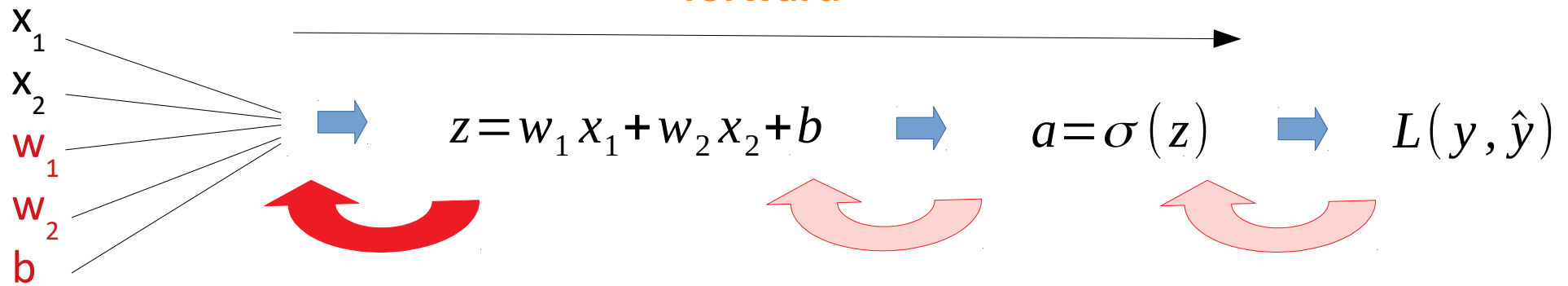
$$\frac{\partial a}{\partial z} = a(1 - a)$$

$$dz = \left(\frac{-y}{a} + \frac{1 - y}{1 - a} \right) (a(1 - a))$$

$$dz = a - y$$

Gradient Descent: for a single sample

forward



$$dz = \left(\frac{-y}{a} + \frac{1-y}{1-a} \right) (a(1-a))$$

$$dz = a - y$$

Applying the chain rule once:

$$dw_1 = \frac{\partial J(y, a)}{\partial w_1} = \frac{\partial J(y, a)}{\partial z} \frac{\partial z}{\partial w_1} = dz \frac{\partial z}{\partial w_1}$$

$$dw_1 = x_1 dz$$

$$dw_2 = \frac{\partial J(y, a)}{\partial w_2} = \frac{\partial J(y, a)}{\partial z} \frac{\partial z}{\partial w_2} = dz \frac{\partial z}{\partial w_2}$$

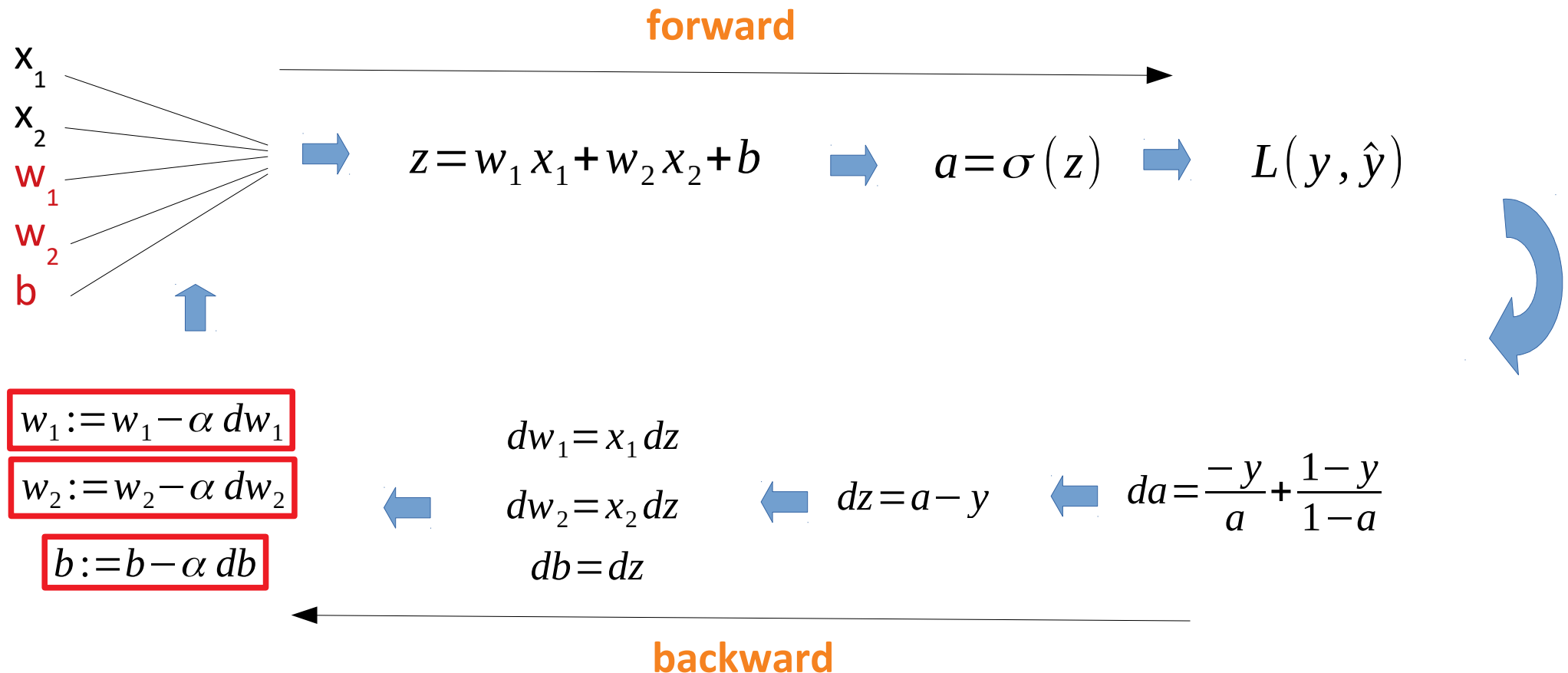
$$dw_2 = x_2 dz$$

$$db = \frac{\partial J(y, a)}{\partial b} = \frac{\partial J(y, a)}{\partial z} \frac{\partial z}{\partial b} = dz \frac{\partial z}{\partial b}$$

$$db = dz$$

Gradient Descent: Putting all together

And the entire process:



Gradient Descent with m samples

When we are dealing with m samples, our cost is:

$$J(w, b) = \frac{1}{m} \sum_1^m L(y^{(i)}, a^{(i)})$$

Then, instead of calculating the derivatives from the loss function, we need to do it for the cost function

$$\frac{\partial J(w, b)}{\partial w_1} = \frac{1}{m} \sum_1^m \underbrace{\frac{\partial L(y^{(i)}, a^{(i)})}{\partial w_1}}_{d(w_1)^{(i)}}$$

$$d(w_1)^{(i)}$$

We know how to calculate this value from the previous slide

Gradient Descent Algorithm

$$J=0, \quad dw_1=0, \quad dw_2=0, \quad db=0$$

for $i = 1$ to m (training samples)

$$z^{(i)} = W^T x^{(i)} + b$$

$$a^{(i)} = \sigma(z^{(i)})$$

$$L+ = -(y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)}))$$

$$dz = a^{(i)} - y^{(i)}$$

$$dw_1+ = x_1 dz$$

$$dw_2+ = x_2 dz$$

$$db+ = dz$$

Forward pass

Loss accumulation

One per input
feature

Gradient/Derivatives calculation

end for

$$L /= m, \quad dw_1 /= m, \quad dw_2 /= m, \quad db /= m$$

Loss accumulation

$$w_1 = w_1 - \alpha dw_1$$

$$w_2 = w_2 - \alpha dw_2$$

$$b = b - \alpha db$$

Learning Rules

Gradient Descent Algorithm

Remember that you need to vectorise your code, computing your matrices using dot products and not going element by element using for loops. This will be crucial when you deal with more complex architectures.

To do that I recommend you write the size dimension of your matrices/vector and be sure that you are computing the proper set of operations by transposing when necessary.

Size of our main matrices/vectors:

$$\begin{matrix} & X = nx, m \\ \begin{bmatrix} z^{(1)} & z^{(2)} & \dots & z^{(m)} \end{bmatrix} & = w^T X + \begin{bmatrix} b & b \dots b \end{bmatrix} = \begin{bmatrix} w^T x^{(1)} + b & \dots & w^T x^{(n)} + b \end{bmatrix} \\ \begin{matrix} 1, m \\ W = nx, m \\ W^T = m, m \end{matrix} & \begin{matrix} 1, m \\ 1, m \end{matrix} \end{matrix}$$

Another element we can improve:

Gradient Descent algorithm

$J=0$, $dw_1=0$, $dw_2=0$, $db=0$

for $i = 1$ to m (training samples)

$$z^{(i)} = W^T x^{(i)} + b$$

$$a^{(i)} = \sigma(z^{(i)})$$

$$L+ = -(y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)}))$$

$$dz = a^{(i)} - y^{(i)}$$

$$dw_1+ = x_1 dz$$

$$dw_2+ = x_2 dz$$

$$db+ = dz$$

One per input
feature

Forward pass

Loss accumulation

Gradient/Derivatives calculation

Create a dw vector and
avoid another for loop here

end for

$$L /= m , \quad dw_1 /= m , \quad dw_2 /= m , \quad db /= m$$

Loss accumulation

$$w_1 = w_1 - \alpha dw_1$$

$$w_2 = w_2 - \alpha dw_2$$

$$b = b - \alpha db$$

Learning Rules

Gradient Descent Algorithm

Finally:

- Once we have implemented the algorithm, we can train this network.
- The initialisation of weights: initial weights are randomly chosen, with typical values between -1.0 and 1.0 or -0.5 and 0.5.
- To ensure a good performance, we need many training samples as possible, so the network can keep adjusting the weights until it has a high enough accuracy.
- When achieved that, we can give it a brand new set of inputs to get the predicted value and see if our model is able to generalise.

Online and Batch learning for MLP

Batch Learning:

In each iteration, process all the training data: for each sample of your training data performs a forward pass and compute the error. Sum all the errors and divide them into the number of samples. Then, proceed to calculate the gradients with the average error and modify the weights accordingly.

Online Learning:

The data is processed sample by sample. Take the first sample and calculate the error (forward pass) only for this sample, then calculate the gradients and modify the weights. Do the same with the second sample, third sample and so on. After you finish with all the samples, repeat the process until convergence.

In neural networks, **one epoch** is a full pass through the dataset

Gradient Descent: Coursework 1

This year my part of the coursework will be focused on neural computation. We will work in a progressive coursework, applying the same concepts for different types of Neural Networks:

- **Coursework 1a:** Gradient Descent for a single neuron (5%)
- **Coursework 1b:** Backpropagation for Shallow and Deep Learning architectures (20%)

Even if it is only 5% of the mark, coursework 1a will be the basis for coursework 1b