Grado en ingeniería informática
Artificial Intelligence 2020/2021

Neural networks for classification

Lara Quijano Sánchez

Universidad Autónoma de Madrid

Readings, sources & resources

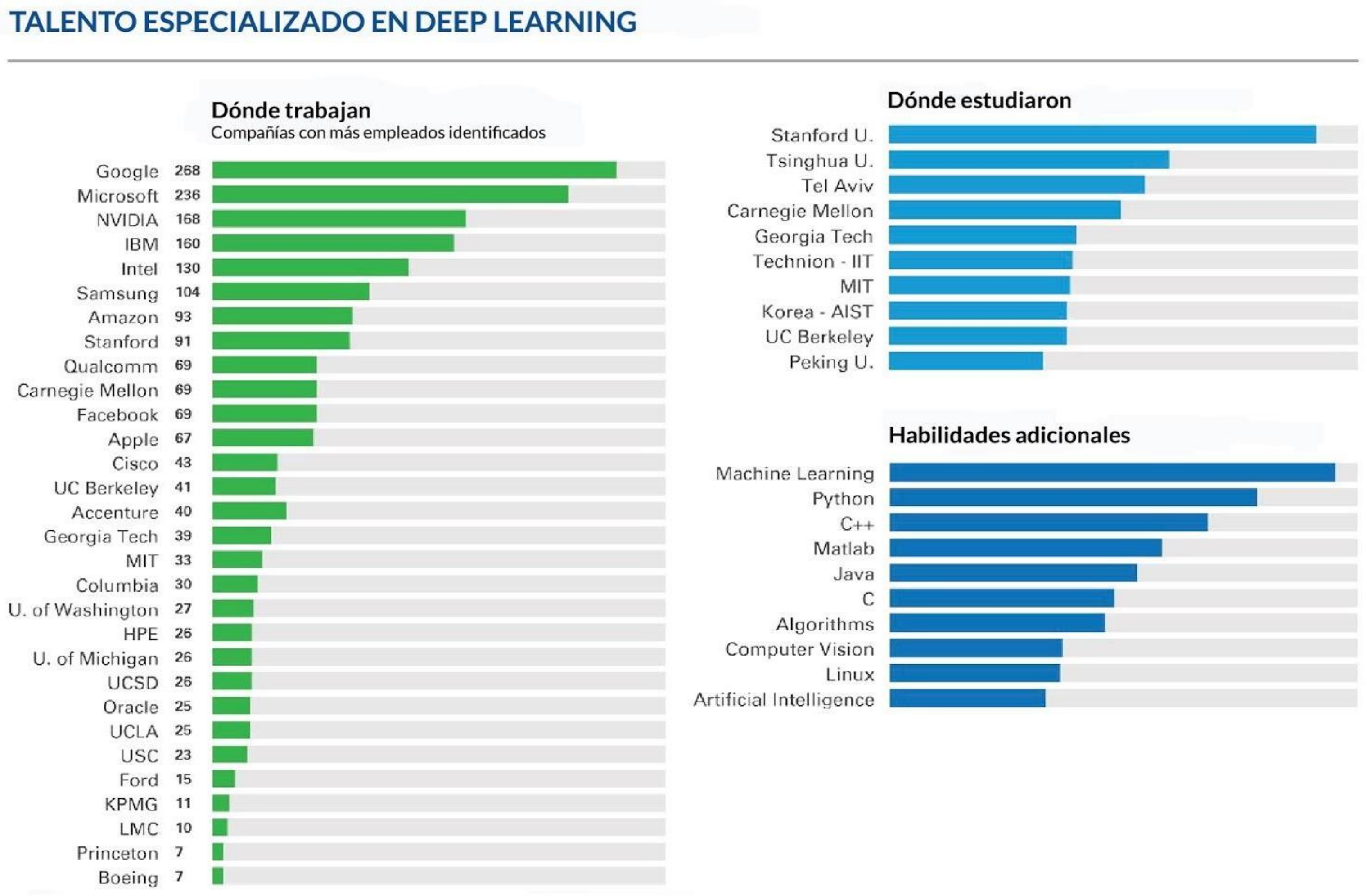
- Multilayer perceptron with TensorFlow
  - https://playground.tensorflow.org/
- Scikit learn documentation
  - Neural networks
    - http://scikit-learn.org/stable/modules/neural_networks_supervised.html
- Simon O. Haykin "Neural Networks and Learning Machines, 3rd edition", Pearson (2009)
- Ian Goodfellow and Yoshua Bengio and Aaron Courville "Deep Learning", MIT Press (2016)
- Michael Nielsen: Introducing neural networks and deep learning(2019). https://goo.gl/Zmczdy
- Learning Representations by Back-propagating Errors. D. Rumelhart, G. Hinton, and R. Williams. Nature 323 (6088): 533--536 (1986)
- Python Data Science Handbook
- Kaggle courses: intro to deep learning
- Dan Becker's deep learning tutorials
- ¿Pero qué "es" una Red neuronal? | aprendizaje profundo, Parte 1
- ¿Qué es una Red Neuronal? Parte 1 : La Neurona

What is deep learning?

- "Deep learning networks are a revolutionary development of neural networks, and it has been suggested that they can be utilized to create even more powerful predictors"

- Main goal
  - Learn from the data to predict a response variable using a group of attributes.

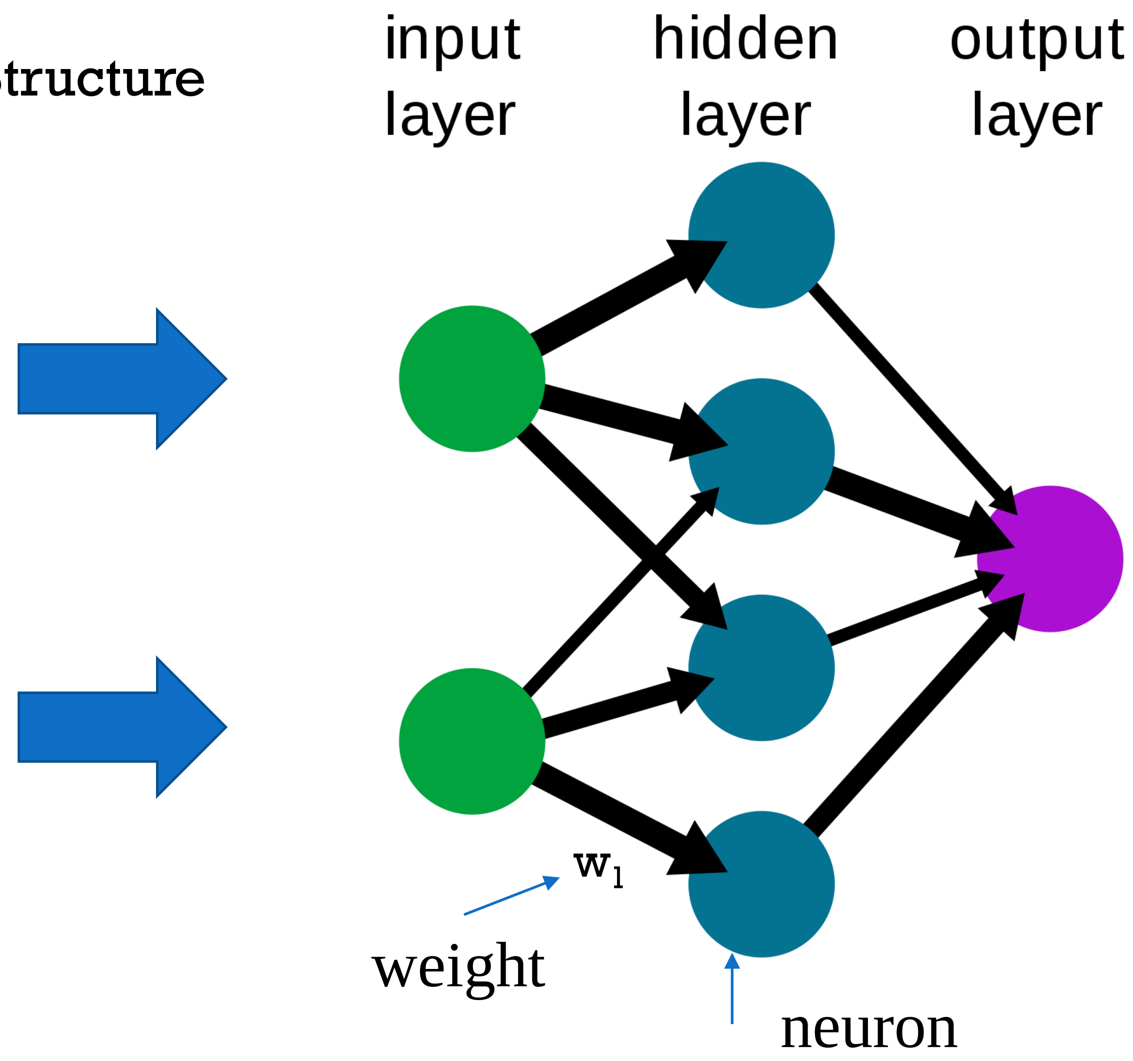
Motivation



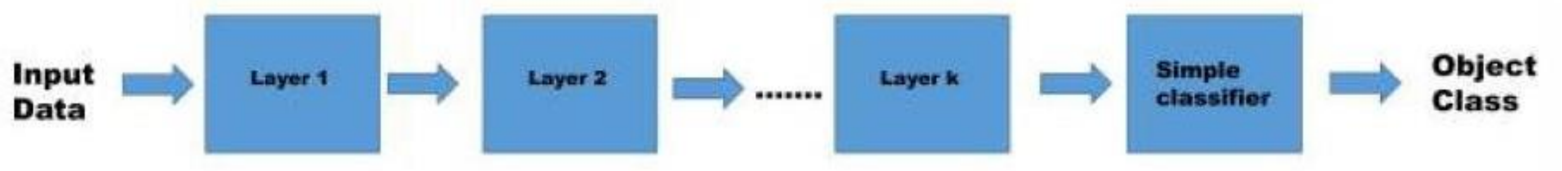
Búsqueda en LinkedIn de personas identificadas con la habilidad "deep learning" capturando dónde trabajan, dónde estudiaron y qué otras habilidades tienen. (realizado el 1 de junio de 2016)

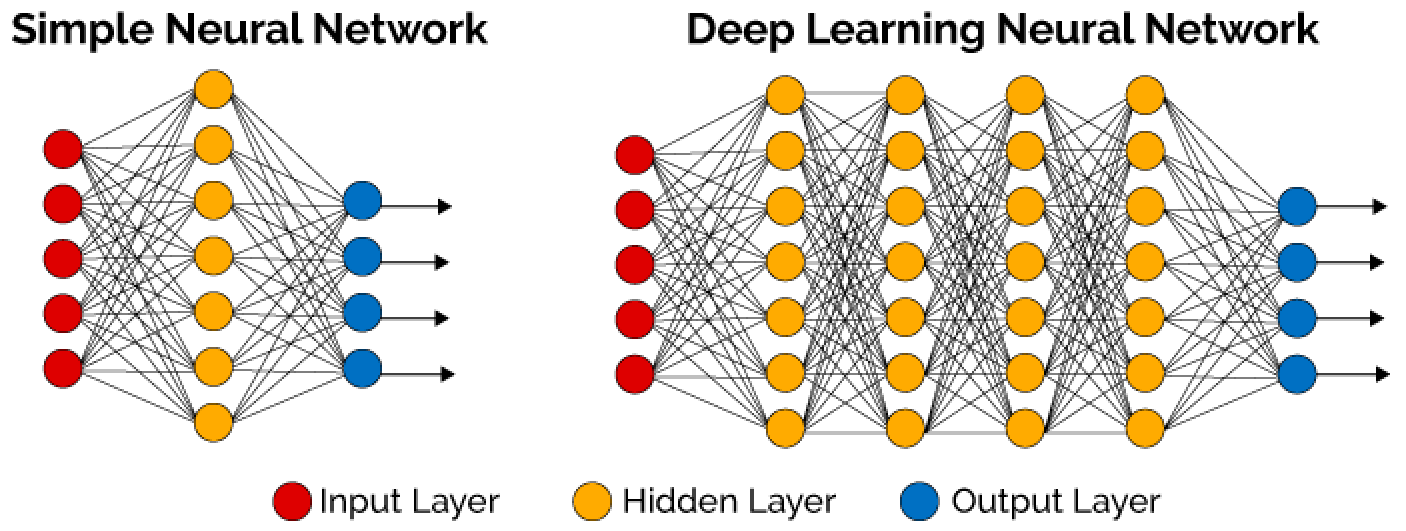
Neural network (not deep)

- Structure



Neural network (deep)

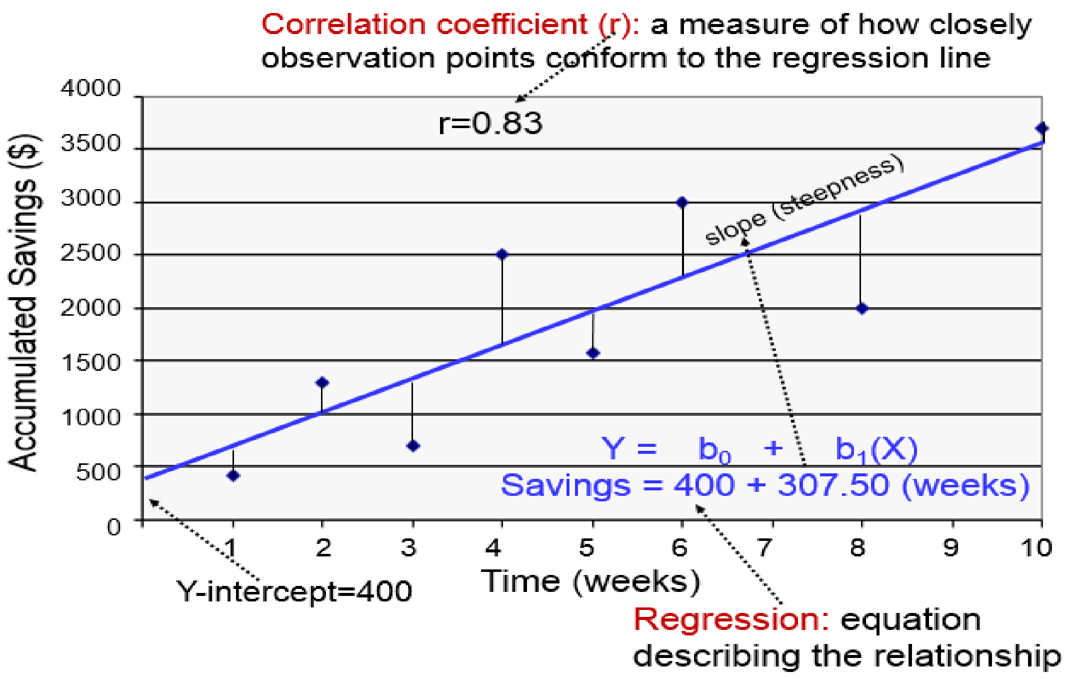




Source: The Data Scientist . What deep learning is and isn't. 2018

Basic Concepts: simple regression

- A linear **regression** line has an **equation** of the form **Y = aX + b**, where X is the explanatory variable and Y is the dependent variable. The slope of the line is a, and b is the intercept.
  - Slope(a) -> how much you can expect Y to change as X increases (units of the Y variable per units of the X variable)
  - Intercept (b) -> is the place where the regression line y = ax + b crosses the y-axis (where x = 0)
  - The Interpretations of the slope and y-intercept are over the range of x values



<u>Interpretation examples</u>:

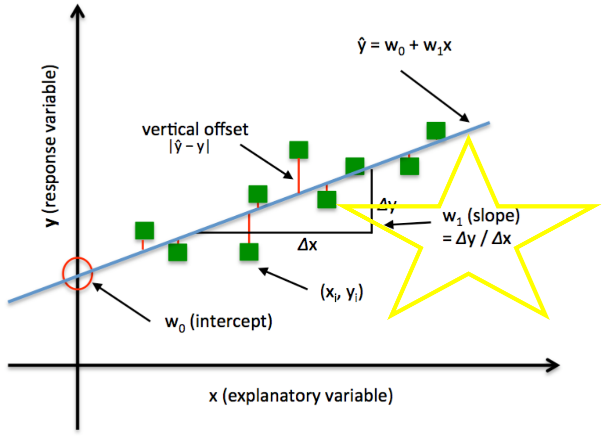
1. **Slope**: if *Y* is an exam score and *X* = study time, and you find the slope of the equation is 5, what does this mean? Not much without any units to draw from. Including the units, you see you get an increase of 5 points (change in *Y*) for every 1-hour increase in studying (change in *X*).

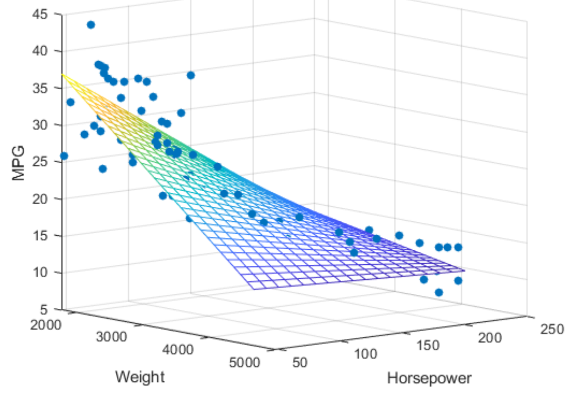
2. **Intercept**: if you're predicting coffee sales (Y) at football games, using temperature (X), some games get cold enough to have temperatures at or even below 0 degrees, so predicting coffee sales at these temperatures makes sense. (they sell more and more coffee as the temperature dips.)

Basic Concepts: Multiple linear regression

- In simple regression we model **Y = aX + b**. That is a direct correlation between feature X and class to predict Y. This model is trained finding the line that minimizes the error (ex. Least squares) for given observations

(rows in our dataset)





- If our class to predict Y correlates (linearly) with **multiple** variables. Then instead of the line we try to find the hyperplane the minimizes the error. We add one dimension for each correlated variable (columns in our dataset)

Formula= $\mathbf{Y = w_1X_1 + w_2X_2 + \ldots + w_nX_n + b}$

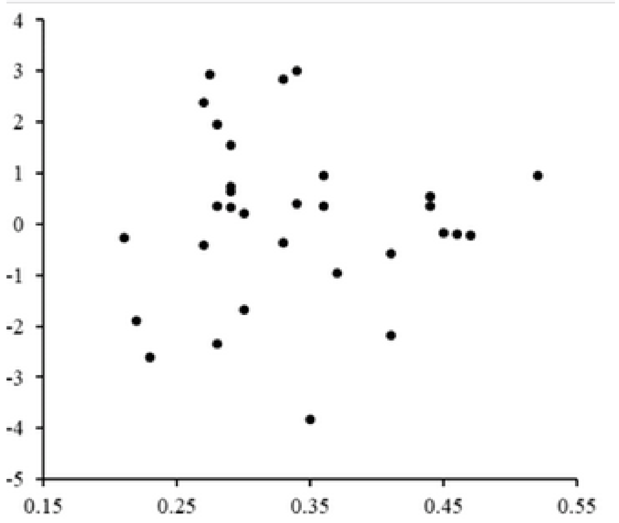
- Example.
  - Initially we want to predict the prize of a house (Y) that correlates linearly with the number of rooms it has. We study the market look for examples (rows) and estimate a linear model
  - Actually the prize of a house (Y) depends/correlates with more variables: x_1= number of rooms, x_2=criminality in the neighbourhood, x_3=city/town, …..thus we compute a multiple linear regression through our observations (rows) of given parameters and the results

Source: Medium. New Aspects to consider while moving from Simple Linear Regression to Multiple Linear Regression.. Lekha Priya. 2019

More on Least squares…

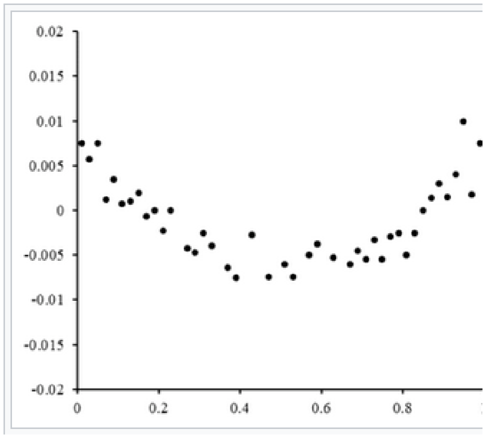
- Minimize the sum of residuals
- N = rows
- $\sum_i^n r_i{}^2$
- Residuals: $r_i = y_i - f(x_i, B)$

- Finding the minimum can be achieved through setting the gradient of the loss to zero and solving the weights



Good for linear regression

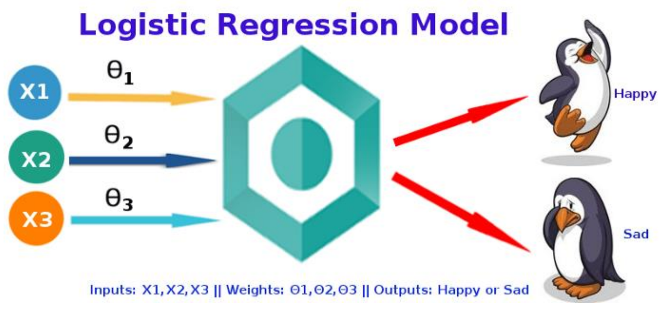
Residuals plotted against the corresponding x values. Parabolic shape → better use a parabolic model

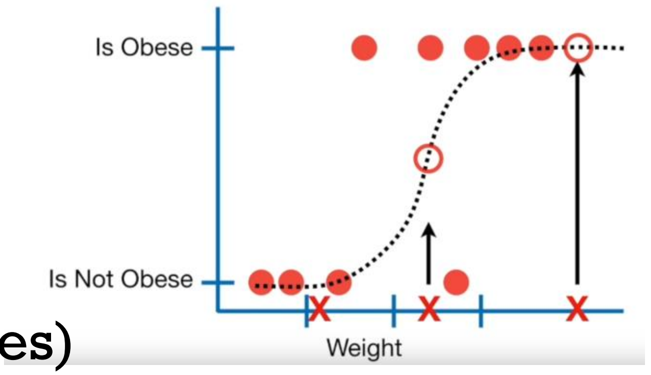


Bad for linear regression

Basic Concepts: Logistic regression

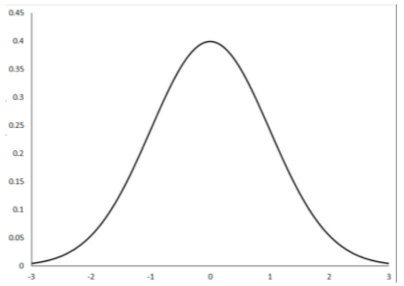
- ❑ Logistic Regression is used when the dependent variable(target) is categorical.
  - ❑Example
    - ❑To predict whether an email is spam (1) or (0)
    - ❑Whether the tumor is malignant (1) or not (0)
- ❑Instead of a line it fits an "S" shaped logistic function (that goes from 0 to 1)
  - ❑Example
    - ❑Tell the probability that a mouse is obese given weight light mouse's not obese, medium mouse's 50% obese, heavy mouse's highly probable obese
- ❑We fit the function (instead of with least squares) with maximum likelihood


Logistic Regression Model
X1
X2
X3
θ_1
θ_2
θ_3
Happy
Sad
Inputs: X1,X2,X3 || Weights: Θ1,Θ2,Θ3 || Outputs: Happy or Sad



Source: Towards data scienc eLogistic Regression — Detailed Overview. Saishruthi Swaminathan 2018; https://www.youtube.com/watch?v=yIYKR4sgzI8; Medium. Logistic Regression. Ratik Puri. 2018.

Basic Concepts: Maximum likelihood



- Given observations $x_1, x_2, ..x_n$ of variable X
  - Assumed to be independent identically distributed
  - Assumed to follow a normal distribution
  - These values are constant
- Given a given moving value θ
  - That can be the mean, standard deviation
- Given a function $f(x_i|\theta)$ that is the probability ($\hat{p}$) of observing x_i given θ
  - that can be that the mean/standard deviation of the normal distribution is θ
- We find the value θ that for all the observations maximices the probability
  - Product rule, all observations assumed to be independent => factorizes:

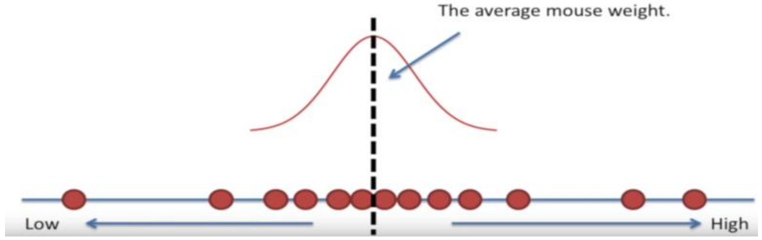
$$\mathcal{L}(\theta) = \hat{p}\,(x_1, x_2, ..x_n|\theta) = \hat{p}\,(x_1|\theta)\,\hat{p}(x_2|\theta)\,\hat{p}(x_n|\theta) = \prod_{i=1}^{n} \hat{p}(xi|\theta)$$

  - Maximum likelihood = $\theta^* = \underset{\theta}{\arg\max}\, \mathcal{L}(\theta)$
  - In practice, depending on the distribution that generated the data, the logarithm of this function is usually used
    - Hence $\ln \mathcal{L}(\theta) = \sum_{i=1}^{n} \ln \hat{p}(xi|\theta)$
    - $\theta^* = \underset{\theta}{\arg\max} \ln \mathcal{L}(\theta)$

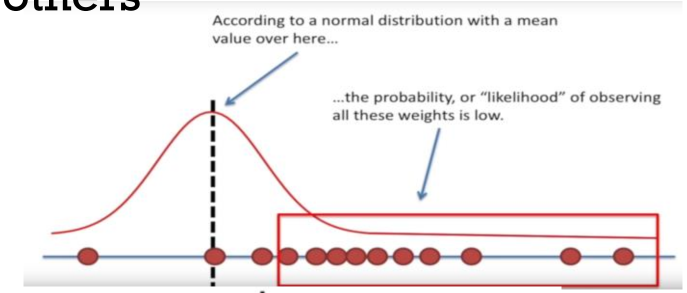
The log of the product of probabilities is the sum of the logs of the probabilities

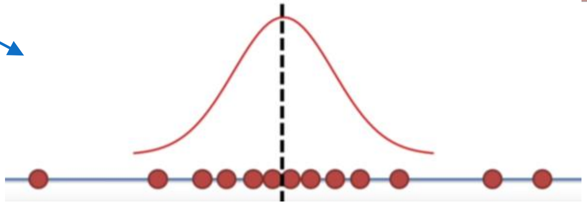
Basic Concepts: Maximum likelihood intuitively

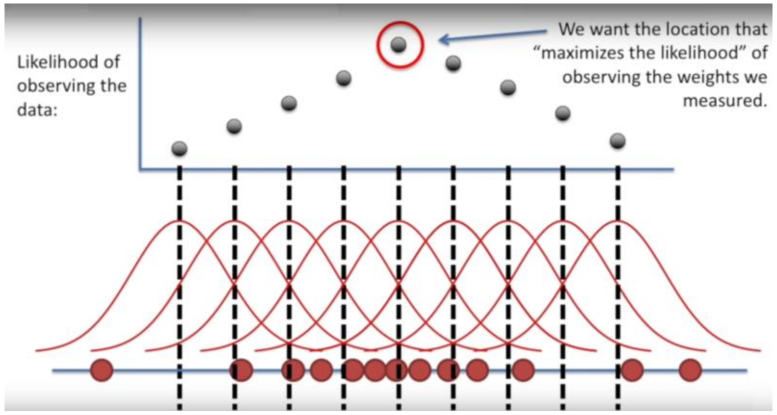
- ❑Normally distributed
  - ❑We expect most of our measurements X_i (mouse weights) to be close to the mean (average)
  - ❑We expect the measurements to be relatively symmetrical around the mean



- ❑Once we settle a bell shape we have to figure out given our observations where to center the "thing". Some locations will be better than others
  - ❑We try different mean of the distribution values
    - ❑ For each new mean value we compute the probability of observing all the X_i values
      - ❑ Ex. Shifted the normal distribution so that is mean is the same as the average of the observations
  - ❑Plot the likelihood of observing the data over the location of the center of the distribution


According to a normal distribution with a mean value over here...
...the probability, or "likelihood" of observing all these weights is low.

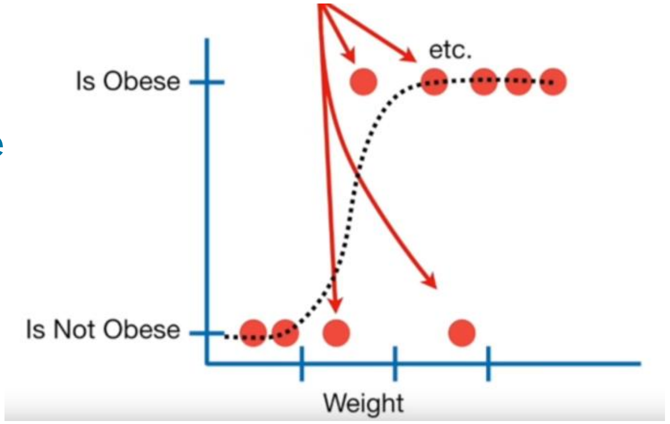



Likelihood of observing the data:
We want the location that "maximizes the likelihood" of observing the weights we measured.

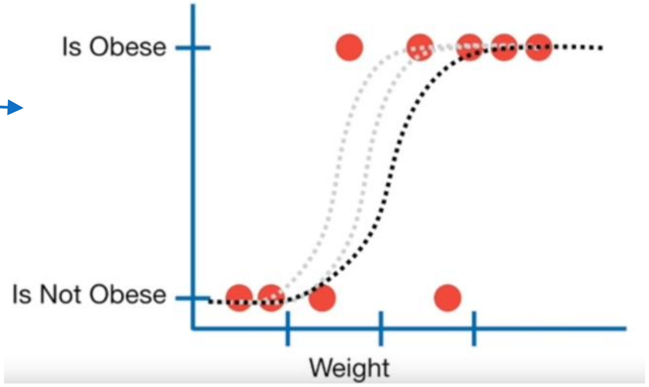
Source: https://www.youtube.com/watch?v=XepXtl9YKwc

Basic Concepts: Logistic regression intuitively

- ❑Fitting our S curve with maximum likelihood
  - ❑Pick a probability (S curve) of observing an obese mouse
    - ❑Use the curve to compute the likelihood of observing mouse$_i$ that weights X$_i$
    - ❑ repeat for 1-n observations, multiply the likelihoods
      - ❑ We get the likelihood of the data given that line



- ❑ Shift the line and get new likelihoods of the data



- ❑The curve with the maximum value for the likelihood is selected
- ❑Types of logistic regression
  - ❑Binary (classify to categories)
  - ❑Multinomial logistic regression (more than 2)

Source: Towards data scienc eLogistic Regression — Detailed Overview. Saishruthi Swaminathan 2018; https://www.youtube.com/watch?v=yIYKR4sgzI8; Medium. Logistic Regression. Ratik Puri. 2018.

Basic Concepts: Logistic regression

- How we get the sigmoid function
  - Y-axis is confined to probability values between zero to one
    - to solve this problem the y-axis in logistic regression is transformed to log(odds) so that the y-axis like in linear regression can have values from -infinity to +infinity
    - Odds are the ratio of something happening to something not happening
      - Odds of an event = p/(1-p) where p denotes the probability of the event
- the logarithm of odds of data
  - Log(odds)=Log(p/1-p)
    - here Log(p/1-p) forms the Logit function
  - If log_odds($\hat{p}$ (y=1|x))=$w_o+w_1x_1+w_2x_2+\ldots+w_nx_n$

  and we call $w_o+w_1x_1+w_2x_2+\ldots+w_nx_n$ simply z(x) then,

  log_odds($\hat{p}$ (y=1|x))=z(x)

  Taking the inverse,

  $\hat{p}\ (y=1|x)=\sigma(z)=\frac{1}{1+e^{-z}}$

  Hence we get the sigmoid function

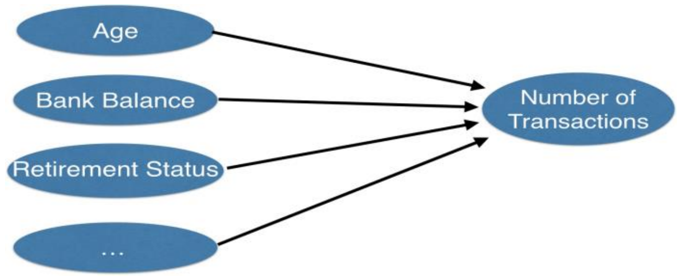
Note when training our model

- Given our function $y = z(x) = \sum_i w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$
- Where estimate the optimal weights w_i
- The bias parameter $b = w_0x_0$ is sometimes referred to as w_0
  - Where we assume and additional static entry in our model of $x_0 = 1$

Going back to neural networks….

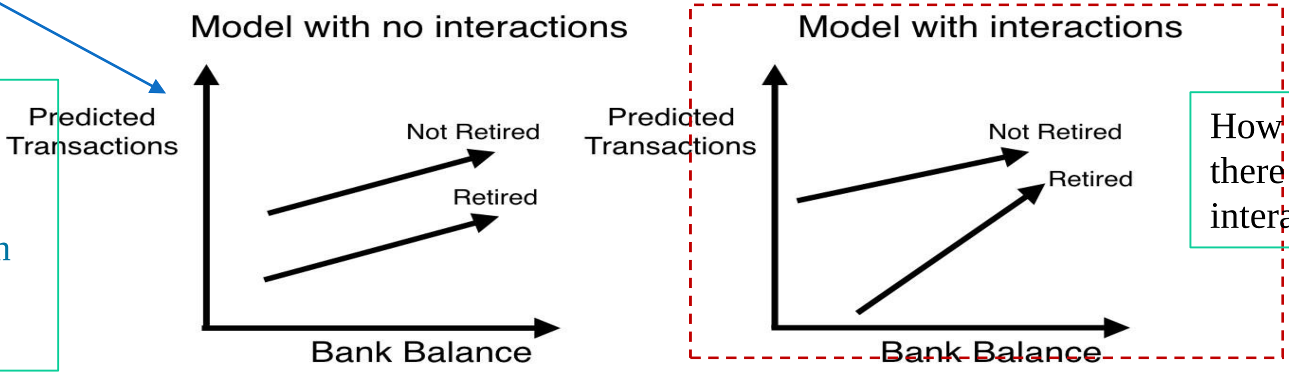
Motivation example

- Imagine you work for a bank
  - You need to predict how many transactions each customer will make next year
  - We have data on the costumers'
    - Age, bank balance, retirement status, etc.…



- Example as seen by a linear regression
  - The outcome= n° of transactions is the sum of individual parts
    - It starts by saying what is the average, then it adds the effective age, bank balance..etc
- The linear regression is not identifying the interactions between these parts and how they affect banking activity
  - If we plot predictions from this model
    - One line for predictions of retired people, another line for not retired…
      - Add the effect of retirement status to bank balance and see the effect on transactions

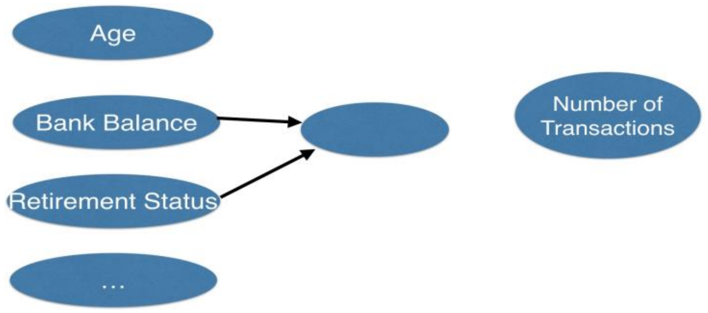
Lack of interaction between two variables. Lines are parallel. Assumption of linear regression model



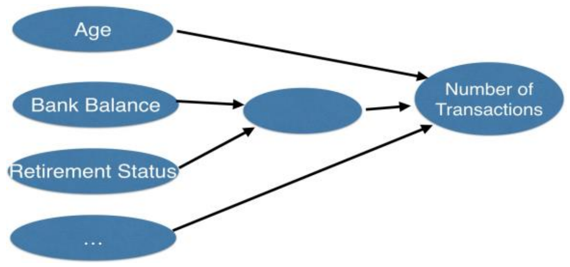
How we model, when there are interactions?

Deep learning models capture interactions

- ❑Deep learning uses especially powerful neural networks
  - ❑ Text, Images, Videos, Audio, Source code
- ❑In the example
  - ❑We saw an interaction between retirement status and bank balance



  - ❑Instead of them separately affecting the income we can calculate a function of these variables that accounts their interaction and use it to predict the outcome
  - ❑ A very simple neural network structure could be:



Motivation example structure

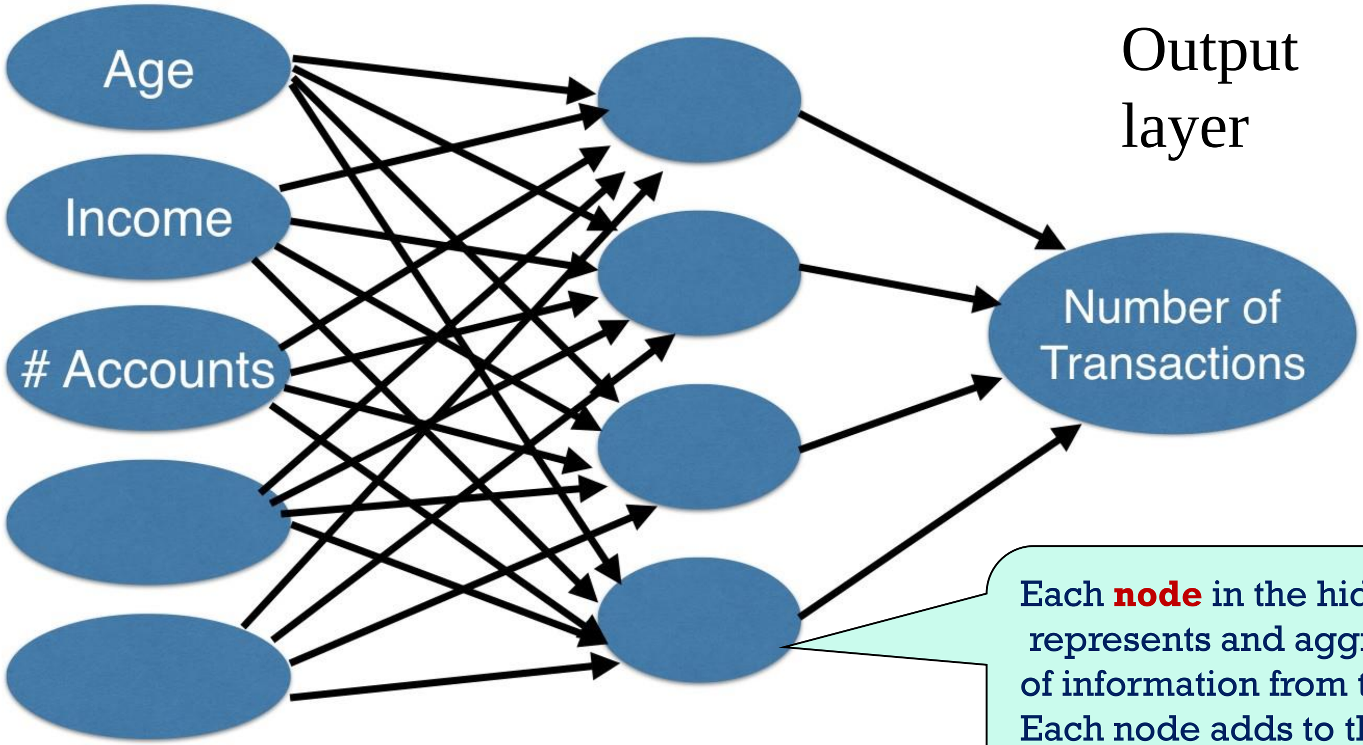
Input layer

Input and **output** are values that happen in the world and can be stored as known data

Hidden layer

The values in the hidden layer Are not something we have data about or anything we have observed directly in the world

Output layer

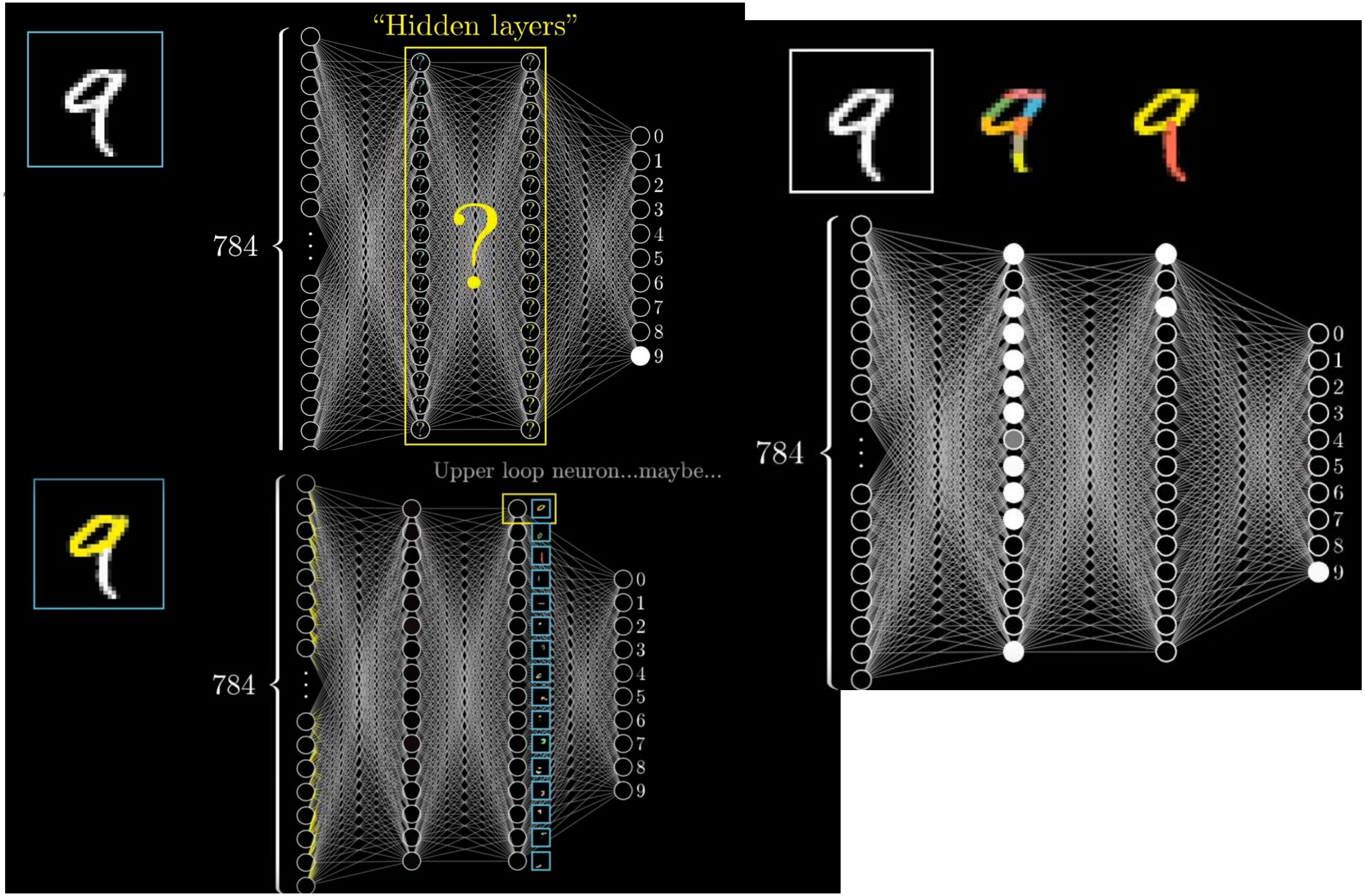


Each **node** in the hidden layer represents and aggregation of information from the input data. Each node adds to the model abilities to capture of interactions

Layer structure

- The input has a neuron for each possible value that represents the current state to evaluate
  - All the attributes in the class
    - Each attribute can have a different value
      - When we train each neuron has each value in the example of each attribute
    - If we are recognizing numbers and have a grid of pixels, each pixel that can be white, black, a tone of grey is a neuron in the input layer
- The output has a neuron for each possible value
  - If is a binary or continuous attribute to predict only one neuron
  - If we can predict different outcomes, one neuron for each different value, and we will get the probability of that value
    - Ex. if we are recognizing numbers 0-9 output neurons, if we are recognizing different crimes 1-ntypes_ofcrime
- The number of hidden layers and neurons in each hidden layer
  - Are specific to the problem
    - Arbitrary
    - Need to be trained /motivated
      - Eg. First layer to recognice bits of drawing, to pass to the second layer that recognices circles top, circle down,, straight vertical lines, straight horizontal lines, final layer composes the results of the previous layer into a number.

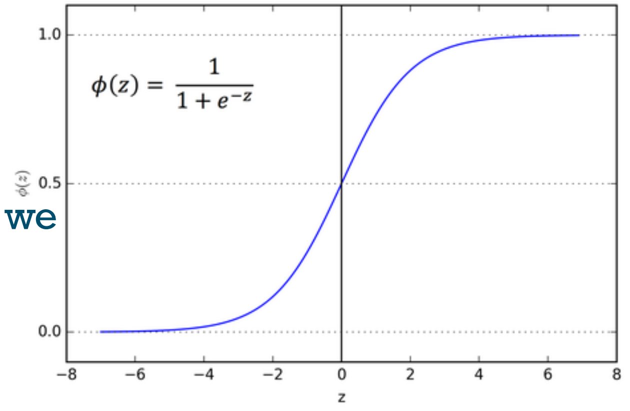
Layer structure

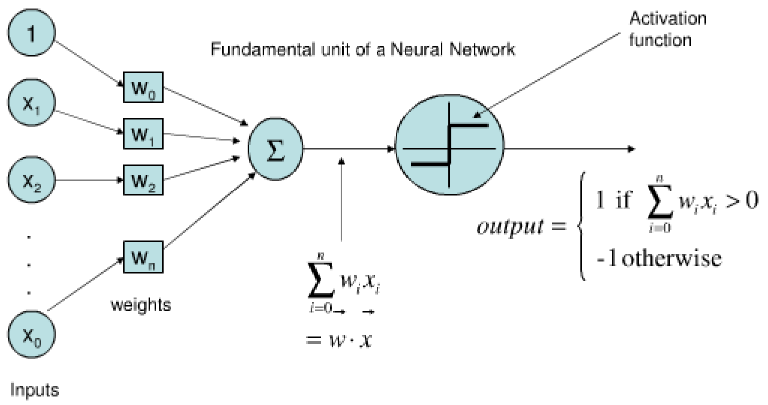


Source:
https://www.youtube.com/watch?v=aircAruvnKk&feature=youtu.be

One neuron structure: is a function

- ❑Input data: $a_1, a_2, a_3, \ldots a_n$
- ❑Weights: $w_1, w_2, w_3, \ldots w_n$
- ❑Weighted sum: $w_1a_1 + w_2a_2 + w_3a_3 + \ldots w_na_n$
  - ❑This weighted sum can take many possible values $(-\infty, \infty)$
    - ❑It might interest us given the domain to narrow the values through and activation function
      - ❑ Ex. We need result values to be between 0-1
        - ❑ Use sigmoid = logistic curve
          - Very negative values close to 0
          - Very positive values close to 1
      - ❑ ActivationFunction $(w_1a_1 + w_2a_2 + w_3a_3 + \ldots w_na_n)$
        - ❑ Sigmoid= $\varphi(w_1a_1 + w_2a_2 + w_3a_3 + \ldots w_na_n)$
    - ❑It might interest us to change the activation point.. maybe we do not want it to activate/say yes when the computed value is greater than 0 maybe we want to set it to 10. That is the BIAS for inactivity
    - ❑ActivationFunction $(w_1a_1 + w_2a_2 + w_3a_3 + \ldots w_na_n$ -bias)



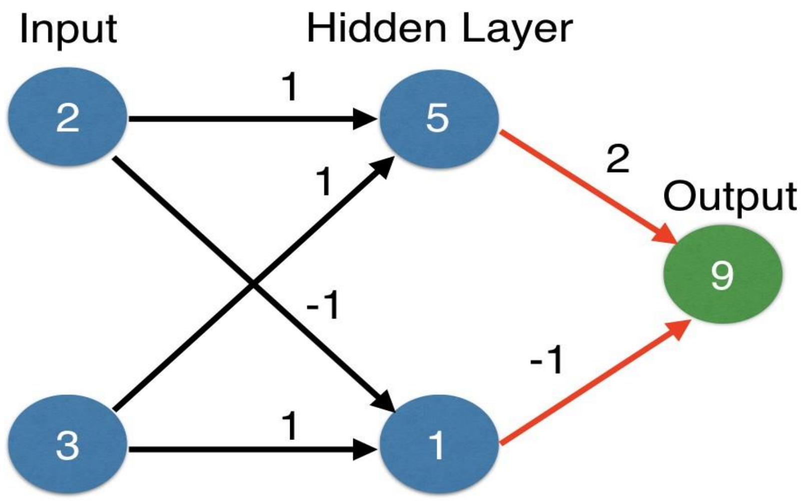


Fundamental unit of a Neural Network
Activation function
1
x_1
x_2
x_0
w_0
w_1
w_2
w_n
weights
Inputs
Σ
$\sum_{i=0}^{n} w_i x_i = \vec{w} \cdot \vec{x}$
$output = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Source: Mohamed, Z. E. (2019). Using the artificial neural networks for prediction and validating solar radiation. Journal of the Egyptian Mathematical Society

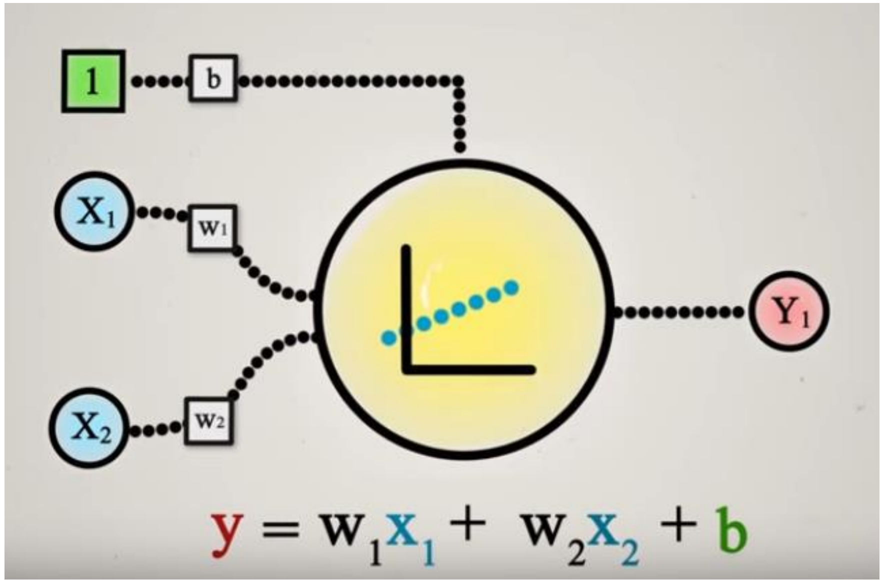
Intuitively: Forward propagation

- In the example of Bank transactions
  - Make predictions on number of transactions based on
    - Number of children
    - Number of existing accounts
  - In the example graph
    - Inputs
      - A costumer with 2 children and 3 accounts
    - Output
      - Number of transactions
    - We try with 2 neurons and 1 hidden layer
      - Each line reflects the weight for that neuron of each of the inputs = set of weights = parameters we train



Simplification

- The neuron takes all the input values to make a weighted sum of them.
- The weighting of each of the inputs is given by the weight assigned to each of the input connections.
  - In other words, each connection that reaches our neuron will have an associated value that serves to define how strongly each input variable affects the neuron
- We have already seen this structure before… it looks like a linear regression
  - that is, a line or hyperplane to which we can vary the inclination according to our parameters.
  - Recall the presence of the bias/intercept parameter

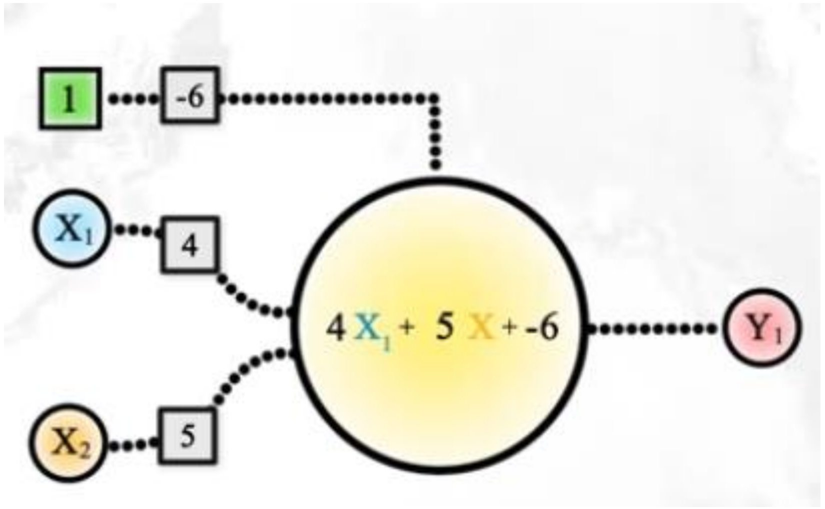


$$y = w_1x_1 + w_2x_2 + b$$

Source: https://www.youtube.com/watch?v=MRIv2IwFTPg

To classify: Single layer perceptron

- ❑With a single neuron
- ❑Given a truth table
- ❑We will put some initial weights
- ❑We will modify those weights so that no matter what value of *X* enters our truth table is fulfilled
- ❑And *Y* returns the expected value

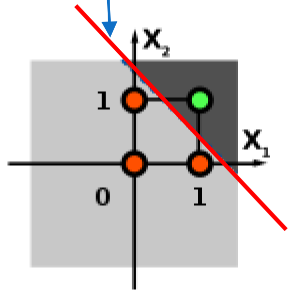


X1	X2	Target	Y
0	0	0	-6
1	0	0	-2
0	1	0	-1
1	1	1	3

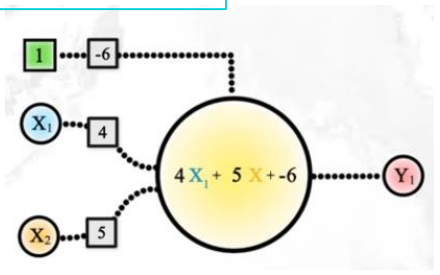
Linearly separable models

- ❑A linearly separable problem is one that can be divided into two clearly differentiated areas by a line
  - ❑Example
    - ❑Function AND
    - ❑Function OR
    - ❑Function XOR (not linearly separable)

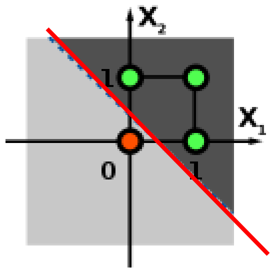
The line we have defined



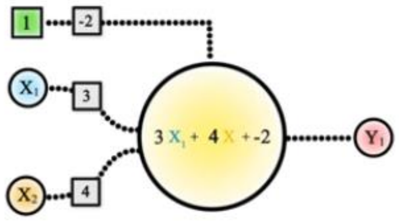
AND $(X_1 \cap X_2)$



X1	X2	Target	Y
0	0	0	-6
1	0	0	-2
0	1	0	-1
1	1	1	3

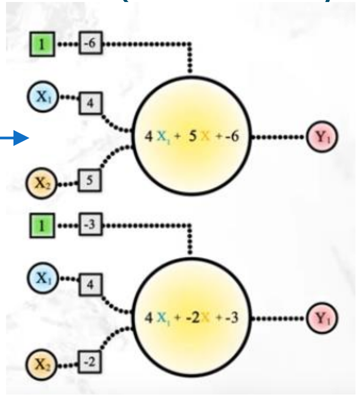


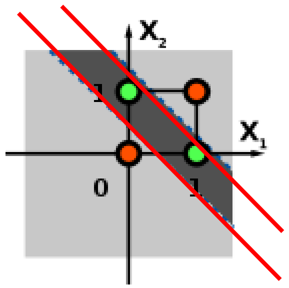
OR $(X_1 \cup X_2)$



X1	X2	Target	Y
0	0	0	4
1	0	1	-2
0	1	1	-1
1	1	1	-3

Cannot be done with a single layer perceptron. We need multi-layer perceptron



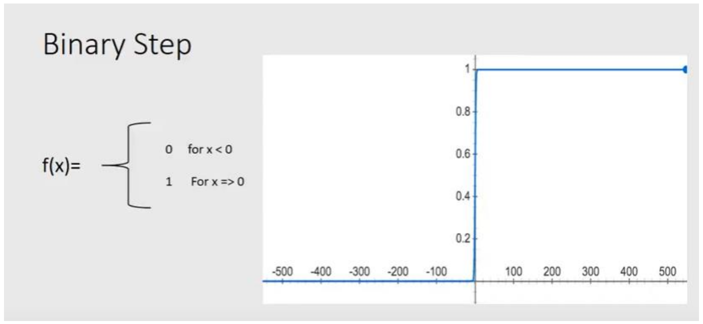


XOR

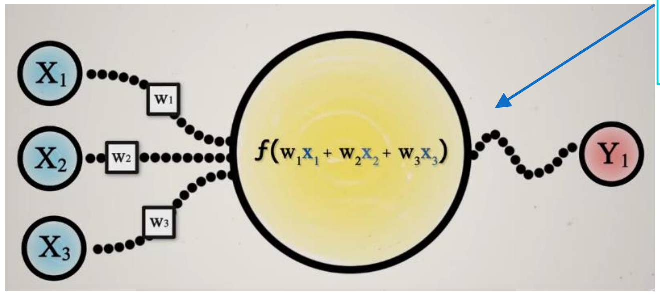
x_1	x_2	h
0	0	0
0	1	1
1	0	1
1	1	0

Motivation on activation functions

- In the previous examples
  - Our activation function was a binary step

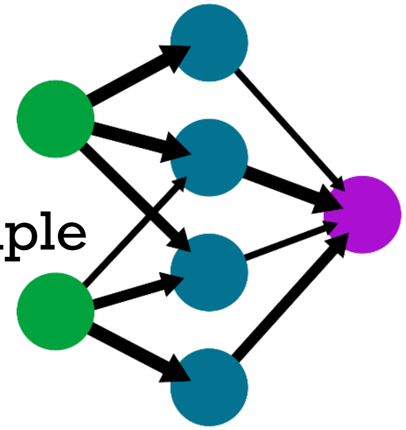


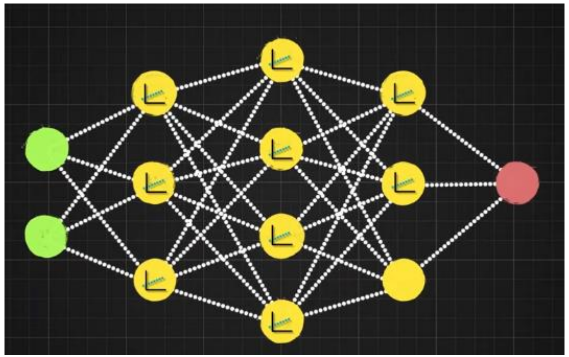
- The concept of activation functions is important



adds nonlinear deformations to our output function

- To achieve our deep learning we want to connect multiple neurons sequentially
  - That is, concatenate different linear regression operations

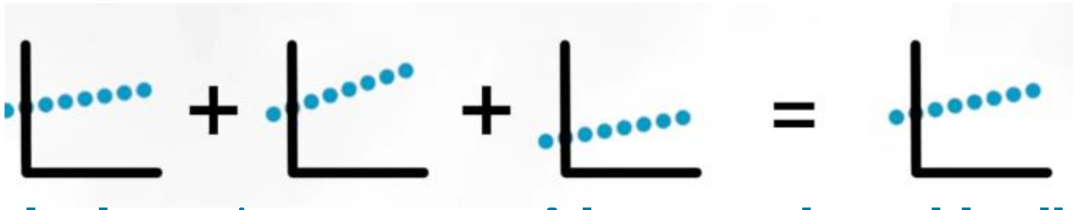




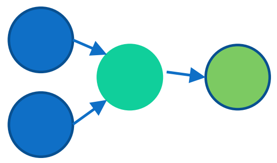
Source: https://www.youtube.com/watch?v=uwbHOpp9xkc

Motivation on activation functions

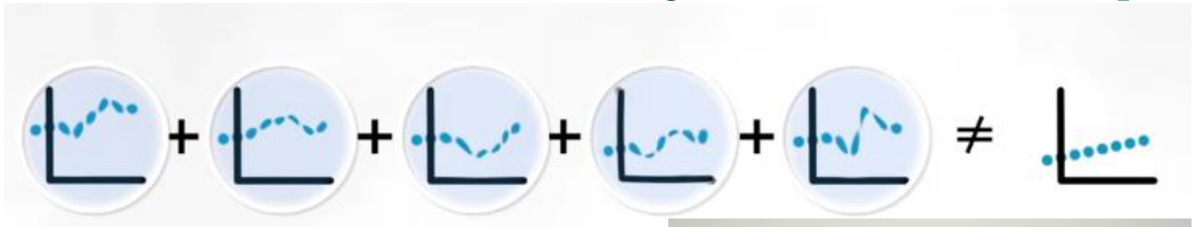
- Mathematically it can be verified that the effect of adding many linear regression operations is equivalent to having done a single operation
  - that is, it results in another straight line



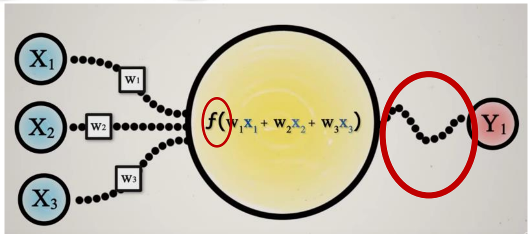
  - In other words, the entire structure of the network would collapse until it was equivalent to having a single neuron



  - to ensure that the structure does not collapse we need our sum of neurons to result in something other than a straight line
    - for that we need each of our lines to undergo some non-linear manipulation that distorts them



  - We need an activation function
    - to distort our output value



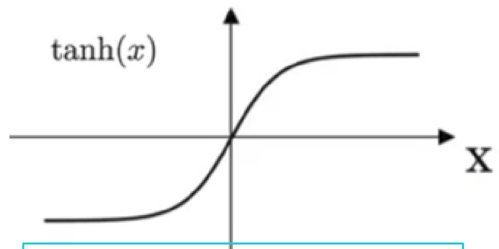
Source: https://www.youtube.com/watch?v=uwbHOpp9xkc

Activation functions

- ❑Finding appropriate architecture can be difficult
  - ❑Choose appropriate activation function
  - ❑Choose appropriate n° of layers and neurons in them

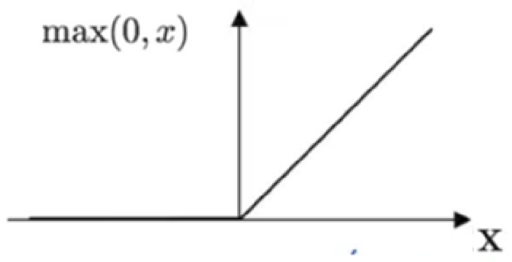
Linear function when positive and constant at 0 when negative

Hyper Tangent Function

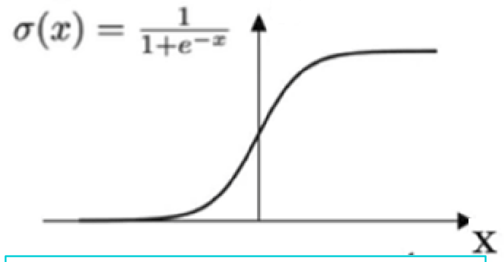


Range from -1 to 1.

ReLU Function
Rectified linear unit

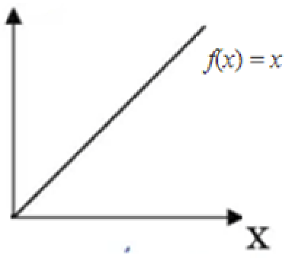


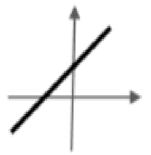
Sigmoid Function

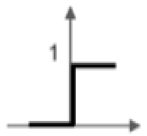


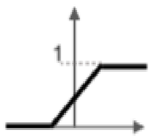
Range from 0 to 1. Useful to represent probabilities

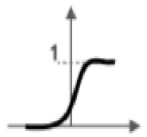
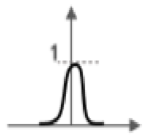
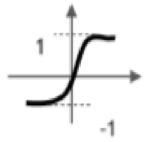
Identity Function



Name	Graphic	Function	
Linear		[illegible]([illegible]) = [illegible] · [illegible] + [illegible]	

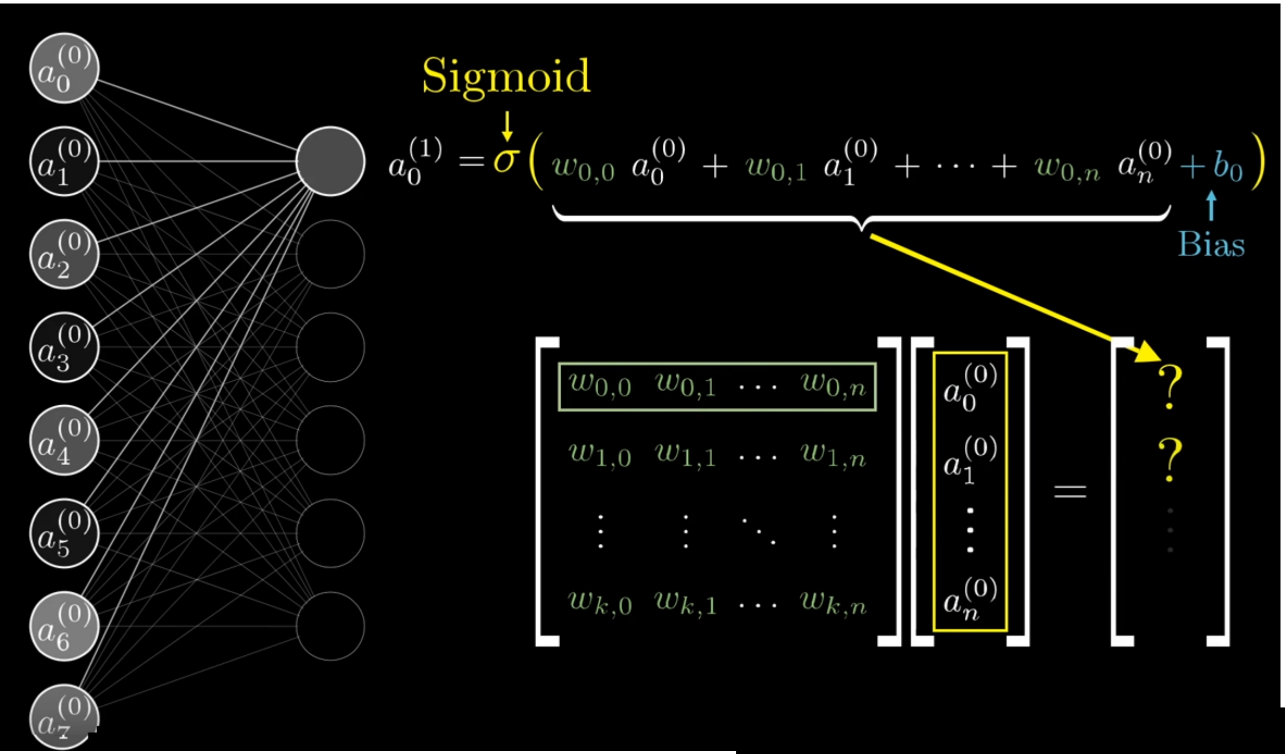
Binary step		if [illegible] ≥ 0,
		if [illegible] < 0,
			then [illegible]([illegible]) = 1,
			then [illegible]([illegible]) = 0,

Piecewise linear		if [illegible] ≥ [illegible],
		if [illegible] > [illegible] > [illegible],
		if [illegible] ≤ [illegible],
			then [illegible]([illegible]) = 1,
			then [illegible]([illegible]) = [illegible] · [illegible] + [illegible],
			then [illegible]([illegible]) = 0,

Sigmoid		[illegible]([illegible]) = $\frac{1}{1+[illegible]}$,	interval (0,1)
Gaussian		[illegible]([illegible]) = [illegible],	interval (0,1]
Hyperbolic tangent		[illegible]([illegible]) = $\frac{2}{1+[illegible]} - 1$,	interval [-1,1]

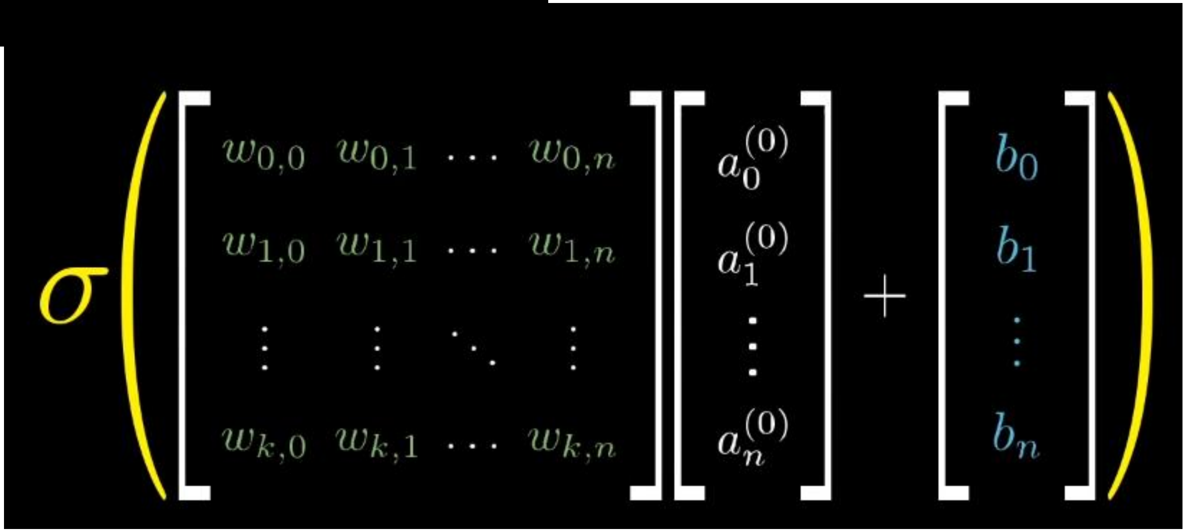
Source: Avinash Navlani. Neural Network Models in R. 2019.

Matrix notation


Sigmoid
Bias
$a_0^{(0)}$ $a_1^{(0)}$ $a_2^{(0)}$ $a_3^{(0)}$ $a_4^{(0)}$ $a_5^{(0)}$ $a_6^{(0)}$ $a_7^{(0)}$

$$a_0^{(1)} = \overset{\text{Sigmoid}}{\sigma}\left(\underbrace{w_{0,0}\,a_0^{(0)} + w_{0,1}\,a_1^{(0)} + \cdots + w_{0,n}\,a_n^{(0)}} + \underset{\text{Bias}}{b_0}\right)$$

$$\begin{bmatrix} \boxed{w_{0,0} \quad w_{0,1} \quad \cdots \quad w_{0,n}} \\ w_{1,0} \quad w_{1,1} \quad \cdots \quad w_{1,n} \\ \vdots \qquad \vdots \qquad \ddots \qquad \vdots \\ w_{k,0} \quad w_{k,1} \quad \cdots \quad w_{k,n} \end{bmatrix} \boxed{\begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix}} = \begin{bmatrix} ? \\ ? \\ \vdots \\ \end{bmatrix}$$



$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right)$$

Source:
https://www.youtube.com/watch?v=aircAruvnKk&feature=youtu.be

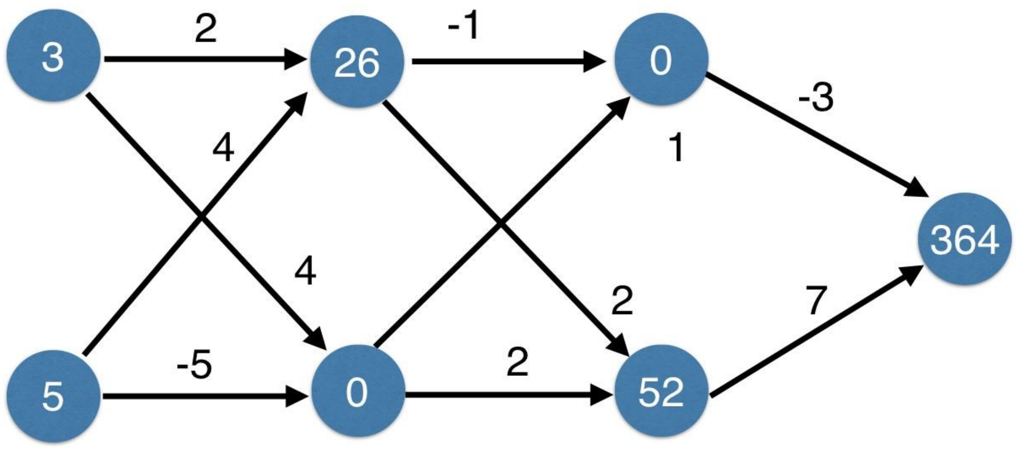
Computation Explosion

- Setting all these weights and biases by hand trying to make each layer work and perform as you want can be a nightmare
  - Ex. A network of 784 input layers, 2 hidden layers of 16 neurons and 10 output layers has
    - ((784*16+16) *16 +16) *10) = 13.002 weights and biases to compute
- Interesting to know the math, and how it works to:
  - Not have black box algorithms
  - Be able to debug
  - Understand the results

Lets play with different configurations of neural networks

Intuitively: Forward propagation with multiple hidden layers and ReLu as activation function

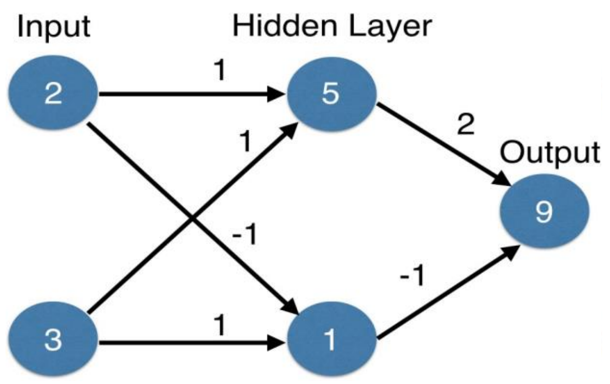
- ❑In the example of Bank transactions
  - ❑Make predictions on number of transactions based on
    - ❑Number of children
    - ❑Number of existing accounts
  - ❑In the example graph
    - ❑Inputs
      - ❑A costumer with 2 children and 3 accounts
    - ❑Output
      - ❑Number of transactions
    - ❑ We try with 2 neurons and 2 hidden layers
      - ❑Each line reflects the weight for that neuron of each of the inputs = set of weights = parameters we train

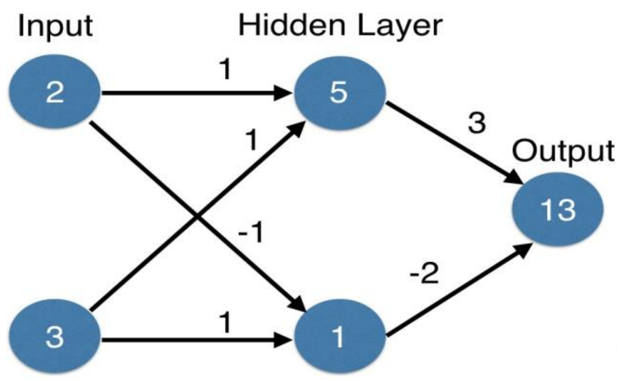


Calculate with ReLU Activation Function

How to compute the best weights?

- The mere fact that a model has the structure of a neural network does not guarantee that it would make good predictions
- Motivation example
  - Activation function = Identity
  - Actual value of target = 13
    - Ex 1. Weights 2° layer = 2 & -1
      - Error = Predicted – Actual = -4
    - Ex 2. Weights 2° layer = 3 & -2
      - Error = Predicted – Actual = 0
- Making accurate predictions gets harder with more data points
- At any set of weights, there are many values of the error
  - Corresponding to the many points we make predictions for





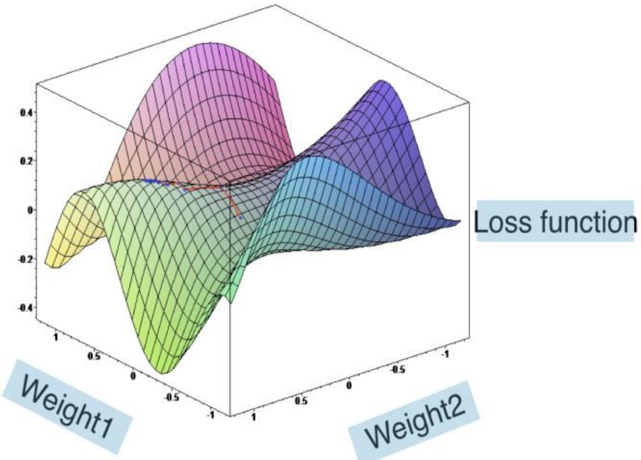
Lets remember: predictive metrics

- Aggregates errors in predictions from many data points into a single number
- Measure of model's predictive performance
- Ex: Mean Squared Error:
  - Total squared error = 150
  - Mean squared error = 50
  - MSE= $\frac{1}{n}\sum_{i=1}^{n}(y_i - \overline{y_i})^2$

Prediction	Actual	Error	Squared Error
10	20	-10	100
8	3	5	25
6	1	5	25

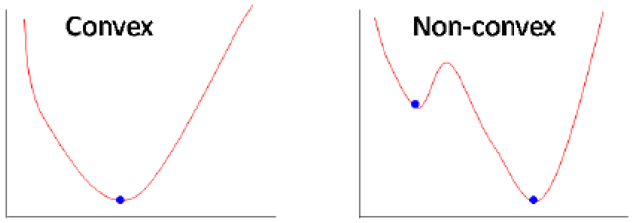
Average cost for all the training examples n

- Loss function
  - Plot the models performance for each set of weights
    - X-axis values of the weights
    - Y-axis value of the loss function
  - Lower loss function value means a better model
  - Goal: Find the weights that give the lowest value for the loss function
  - How?
    - Least squares? -> computational costly with too many parameters
    - Gradient descent

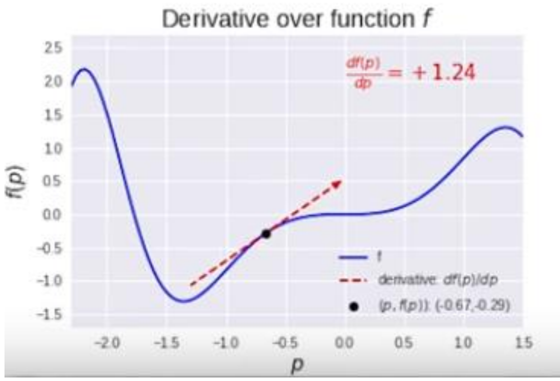


Motivation gradient descent

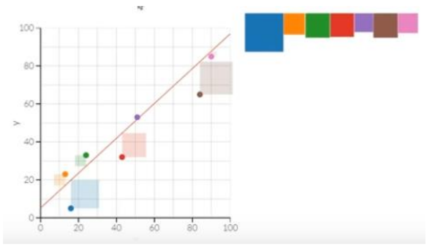
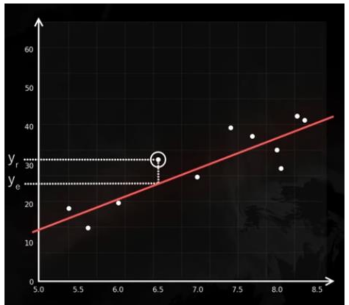
- Starting point:
  - In any model (linear regression, etc) we have some parameters and if we modify those parameters we can vary the error of our model
  - The cost function is the one that tells us what the error is for each of the combinations of our parameters
- Math (cost function is a function…so properties of functions..)
  - In a convex function (U)
    - Property: finding a minimum point will be a global minimum of the function, that is, we will not find a point of the function lower than that
  - Non-convex functions
    - Cannot apply this property: it is possible to find a minimum point that is not the global minimum of the function



- What happens when our error function is non-convex and we need to find the minimum 🤦🏾‍♀️ (yuck)
  - Math
    - We know that if we derive a function, the derivative tells us the slope of the function at that point => we can measure the inclination in each section of our function
    - We set it equal to zero, and we find the point f '(x) = 0 where the slope is zero
    - With convex functions there is only one number => we only have to solve one equation
    - But…
    - With non-convex functions we can find multiple minimum points => multiple equations to solve
      - + local maximums, plateaus / inflection points, saddle points etc
      - => a HUGE system of equations that is very difficult to solve

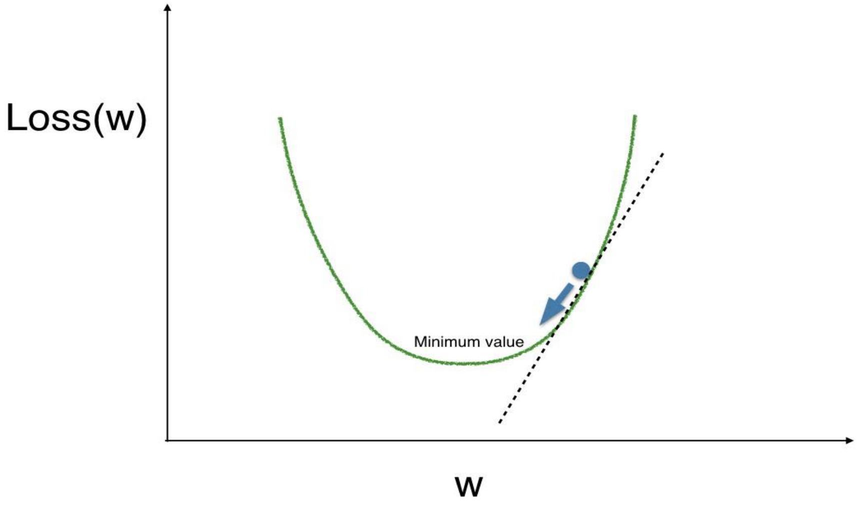


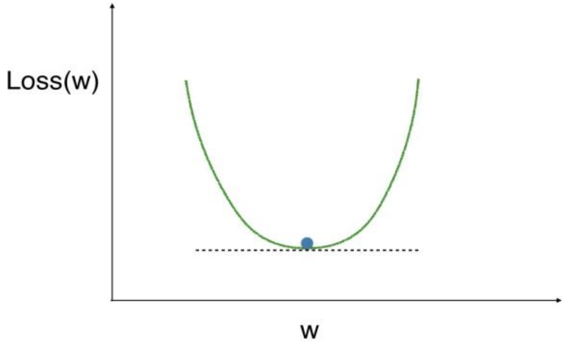
  - When we solved the linear regression problem we forced the cost function to be the least squares so we had a function with convex shape and easy to calculate … but … the diversity of models and error functions forces us to find another solution

Gradient descent (i)

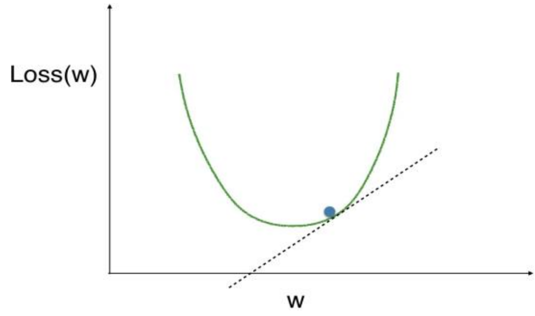
- Motivation
  - Imagine you are in a pitch dark held.
  - You want to find the lowest point
  - Feel the ground to see how it slopes
  - Take a small step downhill
  - Repeat until it is uphill in every direction
- Steps
  - Start at random point
  - Until you are somewhere flat:
    - Find the slope
    - Take a step downhill
- Graphically
  - One single weight
    - Curve, loss = y-axis
    - Different weight values =x-axis
    - We are looking for the lowest point of the curve
      - That means our model is the most accurate
    - We draw a tangent line on the current studied point
      - The slope of the line captures the slope of the loss function at the current weight
        - The slope corresponds to the **derivative** of the loss function
      - We use the slope to decide in what direction we step
      - We keep going to the next weight values in the pointed direction and we arrive to the minimum value
        - We cannot go downhill anymore





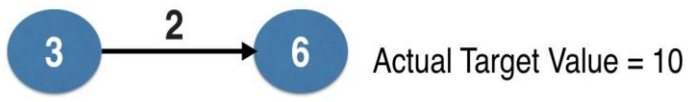
Remember in maths: The minimum point of a function is computed with the derivative and equalling to 0. $\frac{\partial}{\partial x}$ f(x) =0. In our case $\frac{\partial C}{\partial w}$

Gradient descent (ii)



- Compute the tangent line of the loss curve at the current weight
- If the slope is positive:
  - Going opposite the slope means moving to lower numbers
    - How?
      - Subtract the slope from the current weight value
      - Note: Too big a step might lead us astray (we do not find the minimum point…to big steps…or we go to slowly..to costly for the algorithm…)
        - Solution: So instead of directly subtracting the slope we multiply the slope by a small number = learning rate
      - Update each weight' = weight – learning rate*slope
- How to compute the slope/gradient for

Neural networks. Maths:



- To calculate the slope for a weight, we need to multiply 3 things:
  1. Slope of the loss function with respect to the value at the node we feed into
     - Ex. **Slope** of mean-squared loss function with respect to prediction value =

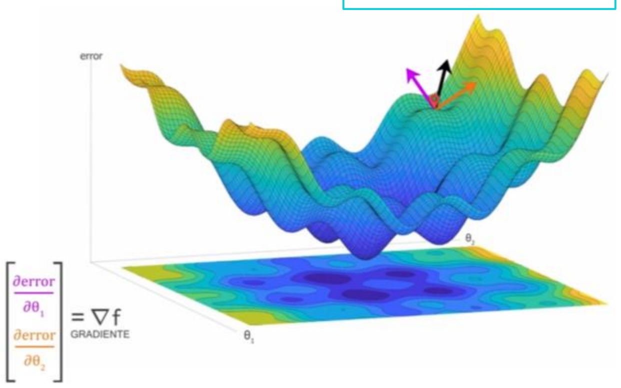
     Derivative of mean-squared **function** = $\frac{\partial}{\partial x}$ f(x) = $\frac{\partial}{\partial x}$ Error^2 = 2 * Error
       - 2 * (Predicted Value - Actual Value) = 2* Error = 2*(6-10)= 2* -4 = -8
  2. The value of the node that feeds into our weight
     - Ex. 3
  3. Slope of the activation function with respect to the value we feed into
     - Ex. If the activation function is the identity = 1
  - Total_slope: (2 * -4) *3 * 1= -24
- Weight' = 2 – learning_rate * total_slope = 2- 0.01*(-24)= 2.24

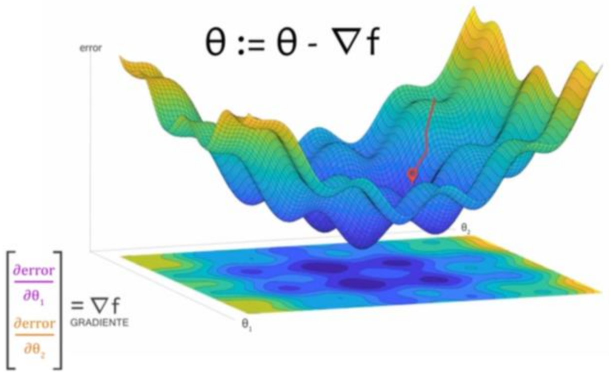
Gradient descent (iii)

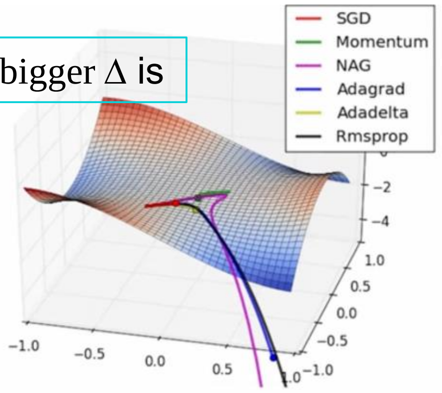
- For multiple weights
  - We apply the same process in parallel separately for each weight
  - Update weights simultaneously using their respective derivatives
- That is…
  - As we have many weights/points we have to calculate the partial derivatives of all of them.
  - Together all these directions (the partial derivatives) indicate a direction towards which the slope ascends =gradient vector
  - To descend we take the opposite direction
  - We continue descending until moving no longer involves a notable variation in cost that is, the slope is close to null and it is most likely that we are at a local minimum, we have minimized the cost of the model
  - Remember the direction we move is appeased with the learning rate (η)
    - **$\theta' := \theta - \eta\Delta C(W)$**
    - There are different techniques that can be used to adjust this hyperparameter dynamically.
  - Yes….And the maths of computing the gradient with multiple weights for neural networks???
    - Lets see..

We only paint 2 weight dimension

Which of these changes is more important = The bigger Δ is


error
θ_2
θ_1
$\begin{bmatrix} \frac{\partial error}{\partial \theta_1} \\ \frac{\partial error}{\partial \theta_2} \end{bmatrix} = \nabla f$
GRADIENTE

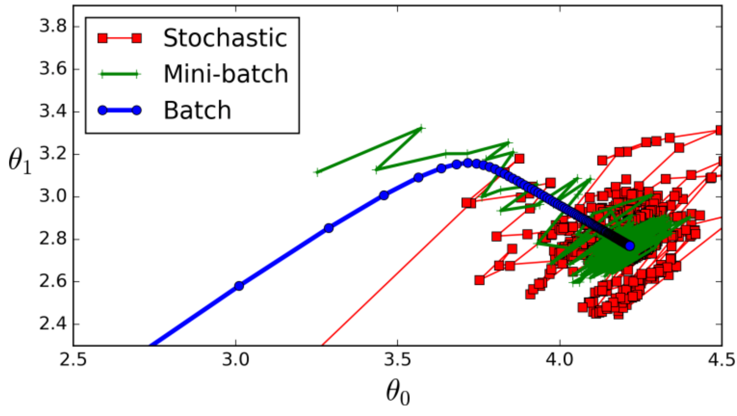

error
$\theta := \theta - \nabla f$
θ_2
θ_1
$\begin{bmatrix} \frac{\partial error}{\partial \theta_1} \\ \frac{\partial error}{\partial \theta_2} \end{bmatrix} = \nabla f$
GRADIENTE



Source: https://www.youtube.com/watch?v=A6FiCDoz8_4

Stochastic gradient descent

- For computational efficiency it is common to calculate slopes on only a subset of the data (a batch) for each update of the weights
- Use a different batch of data to calculate the next update
- Start over from the beginning once all data is used
- Each time through the training data is called an epoch
- When slopes are calculated on **one batch** at a time (rather than on the full data): **stochastic gradient descent**
- Mini-batches
  - The **batch size** defines the number of samples that will be propagated through the network
    - Advantages of using a batch size < n° of all samples
      - It requires less memory. Since you train the network using fewer samples, the overall training procedure requires less memory.
      - Typically networks train faster with mini-batches. That's because we update the weights after each propagation.
        - Example if we propagated 11 batches (10 of them had 100 samples and 1 had 50 samples) and after each of them we updated our network's parameters. If we used all samples during propagation we would make only 1 update for the network's parameter.
    - Disadvantages of using a batch size < n° of all samples:
      - The smaller the batch the less accurate the estimate of the gradient will be. In the figure below, you can see that the direction of the mini-batch gradient (green color) fluctuates much more in comparison to the direction of the full batch gradient (blue color)

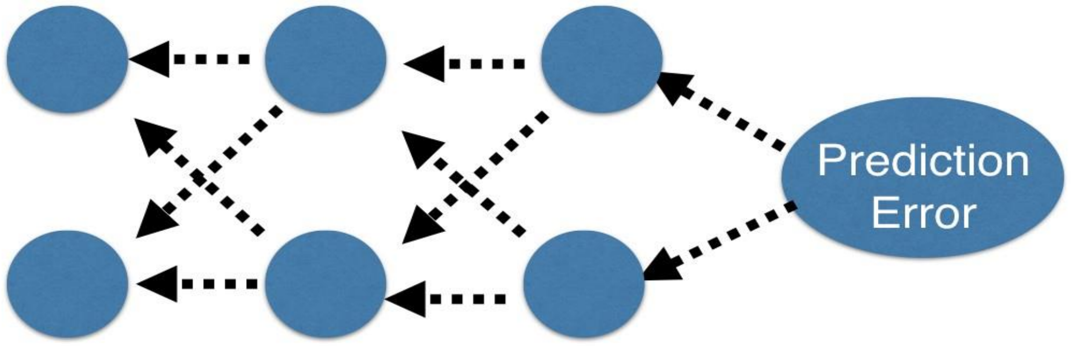


Stochastic is just a mini-batch with batch_size equal to 1. In that case, the gradient changes its direction even more often than a mini-batch gradient

Source: https://stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network

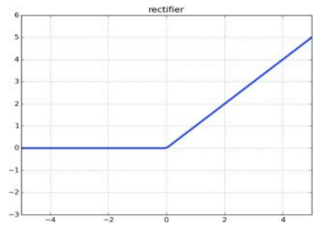
Intuitively: Backpropagation (i)

- We have used gradient descent to optimize weights in a single neuron model
- Now we add a technique called backpropagation to calculate the slopes.. we need to optimize more complex deep learning models
  - That is… calculate the partial derivatives of our algorithm with respect to the cost to optimize our neural network algorithm making use of gradient descent
- Now we will go from output to input layers
  - It calculates the necessary slopes sequentially from the weights closest to the prediction through the hidden layers back to the input layer
  - Then use the slopes to update the weights as previously seen
    - Allows gradient descent to update all weights in neural network (by getting gradients for all weights)
    - Comes from chain rule of calculus

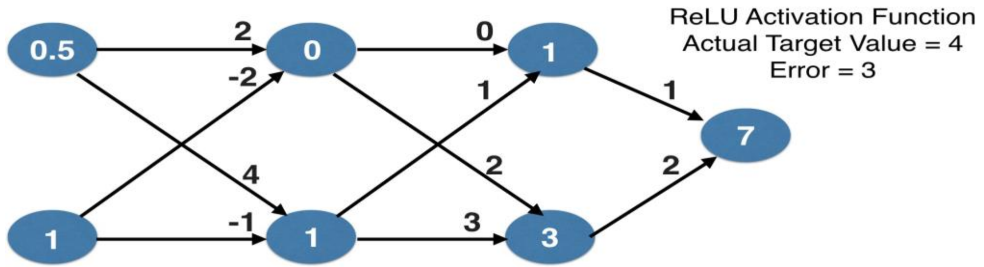


Intuitively: Backpropagation (ii)

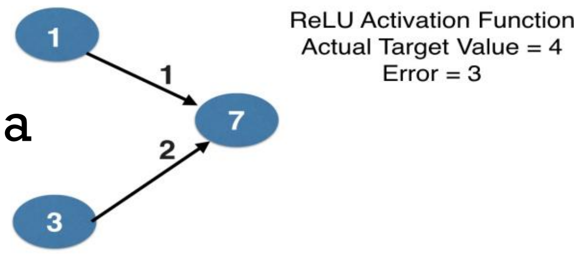
- Trying to estimate the slope of the loss function w.r.t each weight in our network
- We use prediction errors to calculate the slopes
  - So…
  - Do forward propagation to calculate predictions and errors
  - Before we do backpropagation
- Go back one layer at a time
  - Compute Gradients for each weight as the product of:
    1. The value of the weights' input node
       - We know it either because it is an input or we have calculated it first when we first do forward propagation
    2. The slope from plotting the loss function against that weight's value
    3. Slope of activation function at the weight's output
       - In case of ReLU slope= 0 if output is negative slope = 1 otherwise



- Need to also keep track of the slopes of the loss function w.r.t node values
- Slope of node values are the sum of the slopes for all weights that come out of them



Backpropagation example

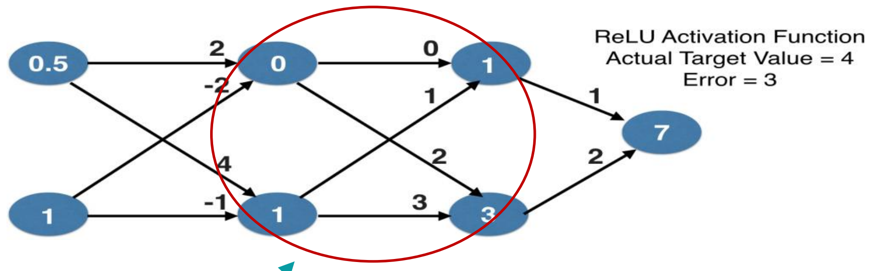


- Starting randomly weights to $w_1=1$ $w_2=2$
- For each weight Multiply 3 things = Gradients for a weight
  1. Node value feeding into that weight
     - Inputs = 1 & 3
  2. Slope of loss function w.r.t output node is 2*MSE= 2* (7-4)=6
  3. Slope of activation function for the node being fed into = 1 since the output is positive

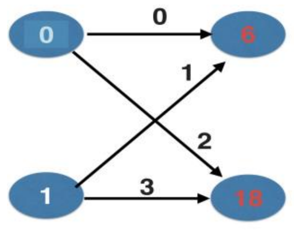
δ { 2., 3. }

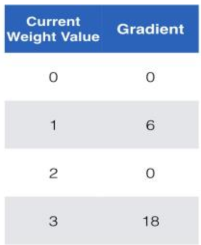
- slope for W_1 (slope_{w1})$\rightarrow$1*6*1=6.
  - $W_1{}^* = W_1 - \text{learning\_rate} * \text{slope}_{w1}$
- slope for $W_2 \rightarrow$ 3*6*1=18.
  - $W_2{}^* = W_2 - \text{learning\_rate} * \text{slope}_{w2}$
- The slopes we have just calculated feed into

the formula back in the network. So lets go back with them = one layer back



- We hid previous layer values & farther away layers
- We keep calculating gradient Backwards. So now (multiplication of our 3* values): input *(W_{1or2}* δ) *slope activation function
  - slope for W_3 = 0 * 1 * 6 * 1=0. $W_3' = W_3 - \text{learning\_rate} * \text{slope}_{w3}$
  - slope for W_4 = 1* 1 * 6 * 1= 6. $W_4' = W_4 - \text{learning\_rate} * \text{slope}_{w4}$
  - slope for W_5 = 0 * 2 * 6 *1 = 0. $W_5' = W_5 - \text{learning\_rate} * \text{slope}_{w5}$
  - slope for W_6 = 1 * 2 * 6 *1 = 18. $W_6' = W_6 - \text{learning\_rate} * \text{slope}_{w6}$

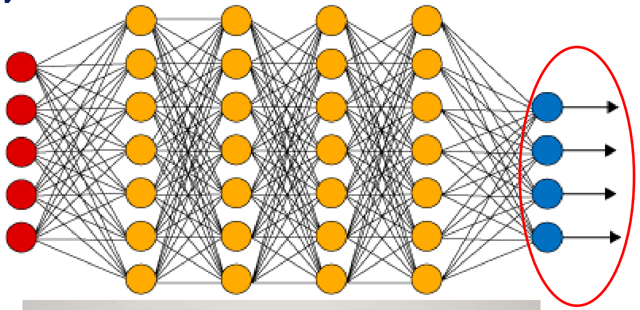


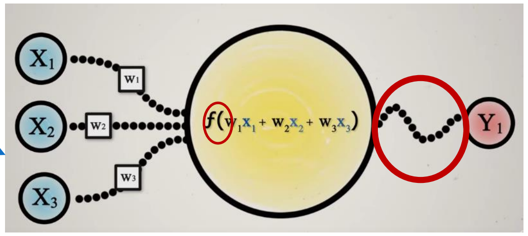


Current Weight Value	Gradient
0	0
1	6
2	0
3	18

Backpropagation recap

- Start at some random set of weights
- What we want to calculate for each parameter of the neural network is the partial derivative of the cost function with respect to each of the parameters (weights and bias) of the network $\frac{\partial C}{\partial w}$ (actually we compute $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$)
- We start with the derivatives of the last layer
  - what parameters are affected?
    - Remember Each neuron has a W set that belongs to a weighted sum:
      - $Z = w_1a_1 + w_2a_2 + w_3a_3 + \ldots w_na_n + b$
    - That is passed to an activation function a
      - $a(Z)$
    - That is then evaluated by function cost C
      - $C(a(Z)) = \text{Error}$
  - To calculate the derivative of a composition of functions in math we have to apply the chain rule
    - Multiply each of the partial derivatives
      - Intuition
        - A bunny is 3 times faster than a dog $= \frac{\partial bunny}{\partial dog} = 3$
        - A dog is 3 times faster than a turtle$= \frac{\partial dog}{\partial turtle} = 5$
        - How fast is the bunny than the turtle? $\frac{\partial bunny}{\partial turtle} = \frac{\partial bunny}{\partial dog} * \frac{\partial dog}{\partial turtle} = 15$





Source: https://www.youtube.com/watch?v=M5QHwkkHgAA

Backpropagation math

- Gradient descent in a neuron
  - $Z = w_1a_1 + w_2a_2 + w_3a_3 + \dots w_na_n + b$
  - $C(a(Z))$
  - $\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a} * \frac{\partial a}{\partial z} * \frac{\partial z}{\partial w}$
  - $\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a} * \frac{\partial a}{\partial z} * \frac{\partial z}{\partial b}$

Cross-entropy error

$$CE = -\sum_i y_i \log \overline{y_i}$$

Gradient of the cross-entropy error

$$\frac{\partial}{\partial z} CE = y - \overline{y}$$

- Is the product of 3 elements (As we said before!) ~ Easy to compute derivatives

1. $\frac{\partial C}{\partial a}$ derivative of the cost with respect to the activation function. How the cost function varies when we vary the output of the neural network

- Ex. Derivative of MSE $\frac{1}{n}\sum_{i=1}^{n}(y_i - a(Z))^2 = \frac{2}{n} * (y - a(Z)) = \frac{2}{n} *$ Error

2. $\frac{\partial a}{\partial z}$ derivative of the activation function with respect to z. How the output of the neuron varies when we vary the weighted sum of the neuron, i.e derivative of the activation function

- Ex. Derivative with an sigmoid activation function $\sigma(z) = \frac{1}{1+e^{-z}}$

$$\sigma'(z) = \frac{d\sigma(z)}{dz} = \sigma(z)\big(1 - \sigma(z)\big)$$

3. $\frac{\partial z}{\partial w}$ or $\frac{\partial z}{\partial b}$

- how the weighted sum varies when we vary the weights.
  - $\frac{\partial z}{\partial w}$ = a the input of the neuron (output of the previous layer)
- how the weighted sum varies when we vary the bias. $\frac{\partial z}{\partial b} = 1$ because b is a constant. Derivatives of constants are 1.

Backpropagation math (ii)

- Gradient descent in the a last layer neuron
  - $Z^L = W^L * a^{L-1} + b^L = w_1a_1+w_2a_2+w_3a_3+\ldots w_na_n + b$
  - $C(a^L(Z^L))$
  - $\frac{\partial C}{\partial w^L} = \underbrace{\frac{\partial C}{\partial a^L} * \frac{\partial a^L}{\partial z^L}} * \underbrace{\frac{\partial z^L}{\partial w^L}}$

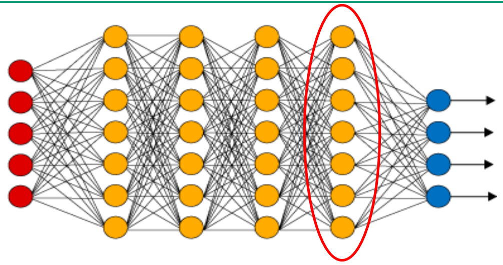
error imputed to the neuron $\delta^L = \frac{\partial C}{\partial z^L}$ $\quad a_i^{L-1}$

  - $\frac{\partial C}{\partial b^L} = \underbrace{\frac{\partial C}{\partial a^L} * \frac{\partial a^L}{\partial z^L}} * \underbrace{\frac{\partial z^L}{\partial b^L}}$

error imputed to the neuron $\delta^L = \frac{\partial C}{\partial z^L}$ $\quad 1$

- How the error varies as a function of the value of Z (the calculated weighted sum).
  - Ie. To what degree the error/cost is modified when the sum of the neuron is modified.
    - If it is large, a small change will affect the final result.
    - If the derivative is small, it does not matter how we vary the value of the sum since it will not affect the error of the network.
  - That is, it tells us what responsibility the neuron has in the final result / in the error

Input previous layer

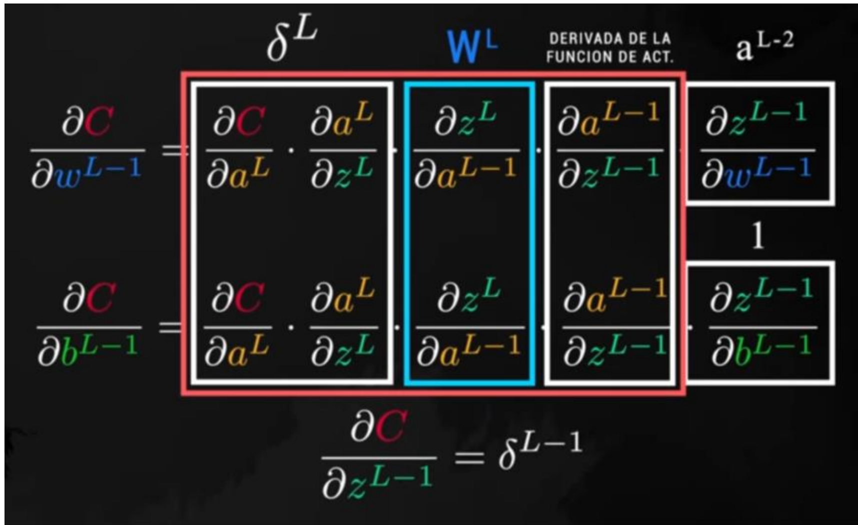


- Then $\frac{\partial C}{\partial w^L} = \Delta f = \delta^L * a_i^{L-1}$ and $\frac{\partial C}{\partial b^L} = \delta^L$

$\text{weight}_\theta^* = \text{weight}_\theta - \eta\Delta f$

- The parameters for layer L-1
  - Chain rule $C(a^L(W^{L-1}a^L(W^{L-1}a^{L-2} + b^{L-1}) + b^{L-1}))$
  - We repeat the process
    1. Slope cost function= **previous** 3 step values = δ^L
       1. Derivative cost
       2. Derivative activation action
       3. Instead of Input parameters = **New**

       Derivative =how the weighted sum of a layer varies when the output of a neuron in the previous layer is varied= the weight of the input in L= W^L
    2. New: Derivative activation function w.r.t new output of the function in that layer
    3. New: Input parameters



$$\begin{array}{cccccc} & \delta^L & & W^L & \text{DERIVADA DE LA FUNCION DE ACT.} & a^{L-2} \\ \frac{\partial C}{\partial w^{L-1}} = & \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} & \cdot & \frac{\partial z^L}{\partial a^{L-1}} \cdot & \frac{\partial a^{L-1}}{\partial z^{L-1}} & \frac{\partial z^{L-1}}{\partial w^{L-1}} \\ & & & & & 1 \\ \frac{\partial C}{\partial b^{L-1}} = & \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} & \cdot & \frac{\partial z^L}{\partial a^{L-1}} \cdot & \frac{\partial a^{L-1}}{\partial z^{L-1}} & \frac{\partial z^{L-1}}{\partial b^{L-1}} \end{array}$$

$$\frac{\partial C}{\partial z^{L-1}} = \delta^{L-1}$$

Backpropagation: Recap

- ❑Start at some random set of weights
- ❑Use forward propagation to make a prediction
- ❑Use backward propagation to calculate the slope of the loss function w.r.t each weight
- ❑Multiply that slope by the learning rate, and subtract from the current weights
- ❑Keep going with that cycle until we get to a flat part

Two ways of learning: Online learning ~ gradient descent

- Randomly initialize weights
- $n_{epoch} = 0$
- While convergence criteria are not met
  - Increment epoch counter: $n_{epoch} = n_{epoch} + 1$
  - For each input entry (new row in the dataset)
    - Perform forward propagation
    - Perform backpropagation
      - For each neuron
        - Compute gradient on each neuron
        - Update weight of the neuron (**$\theta' := \theta - \eta\Delta f$**)

- That is, weight changes with each input hence the gradient computation of the next input is affected by those weight changes

Two ways of learning: Batch learning ~ Stochastic gradient descent

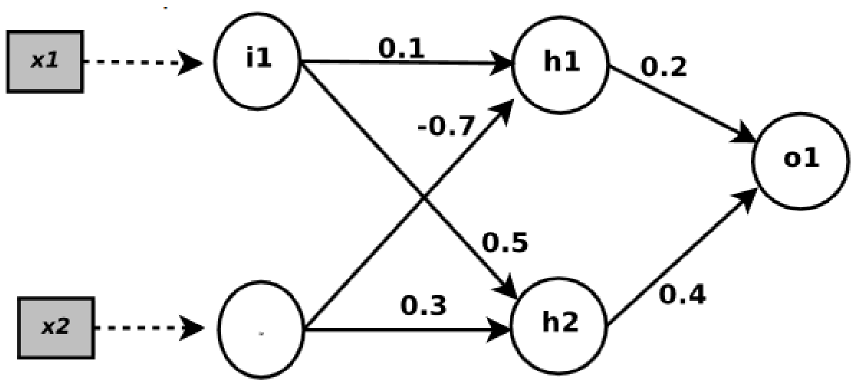
```
❑Randomly initialize weights
❑$n_{epoch} = 0$
❑While convergence criteria are not met
    ❑Increment epoch counter: $n_{epoch} = n_{epoch} + 1$
    ❑For each batch with size m (group of rows in the dataset)
        ❑Perform forward propagation
        ❑Perform backpropagation
            ❑For each neuron
                ❑Compute gradient on each neuron (we compute m gradients for each weight)
    ❑For 0→m
        ❑For each weight of the neuron
            ❑Update the weight with the accumulative gradients $\theta' := \theta - \eta \sum_0^m \Delta f_m$
```

❑That is, we update the weights at the end of each batch of inputs with the cumulative computed gradients

Example: multilayer perceptron to perform the XOR operation. Online learning

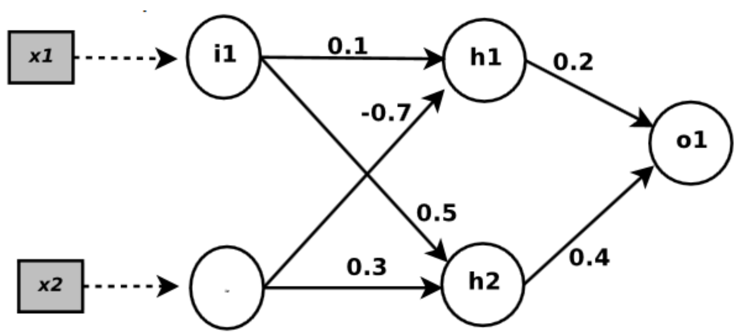
- Network description
  - 1 hidden layer
  - 2 neurons in input layer (i_1, i_2)
  - 2 neurons in hidden layer (h_1, h_2)
  - 1 neuron in output layer (o_1)
  - Initial network with random weights
  - Learning rate η= 0.25

d	x_1	x_2	h
d1	0	1	1
d2	1	0	1
d3	1	1	0
d4	0	0	0

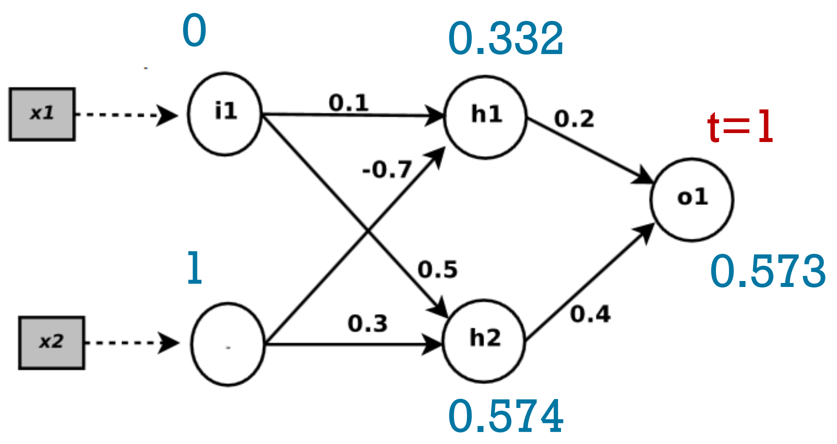


$$W1 = \begin{pmatrix} 0{,}1 & 0{,}5 \\ -0{,}7 & 0{,}3 \end{pmatrix} \qquad W2 = \begin{pmatrix} 0{,}2 \\ 0{,}4 \end{pmatrix}$$

Forward propagation of the first input d1

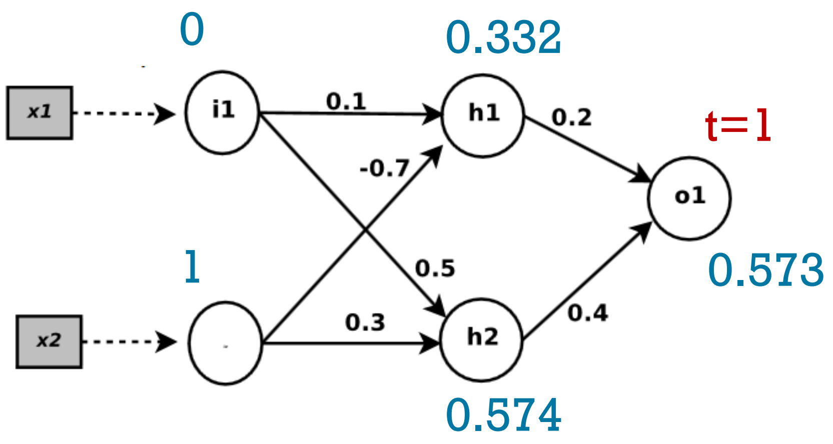


- Inputs
  - x_1=0, x_2= 1
  - Expected output: t_1 =1
- Hidden layer
  - Neuron h_1:
    - Input: $0{,}1*0 + (-0{,}7)*1 = -0{,}7$
    - Output: $\frac{1}{1+e^{0.7}} = 0{,}332$
  - Neuron h_2:
    - Input: $0{,}5*0 + 0{,}3*1 = 0{,}3$
    - Output: $\frac{1}{1+e^{-0.3}} = 0{,}574$
- Output layer:
  - Neuron o_1:
    - Input: $0{,}2*0{,}332 + 0{,}4*0{,}574 = 0{,}296$
    - Output: $\frac{1}{1+e^{-0.296}} = 0{,}573$

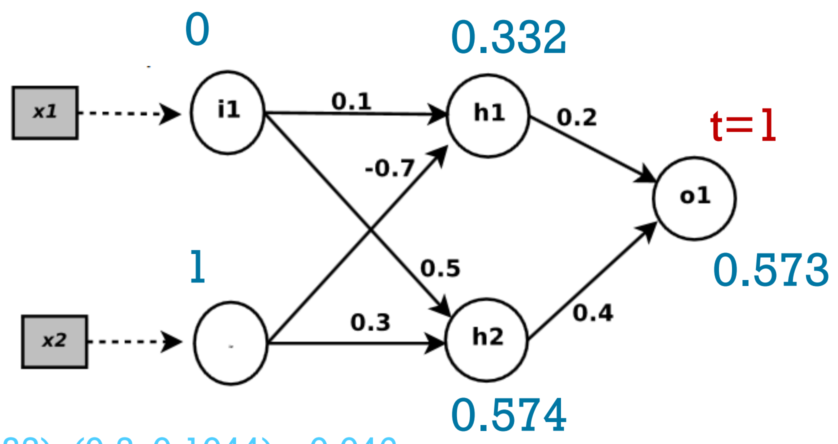


Weight adjustment by backpropagation of the error

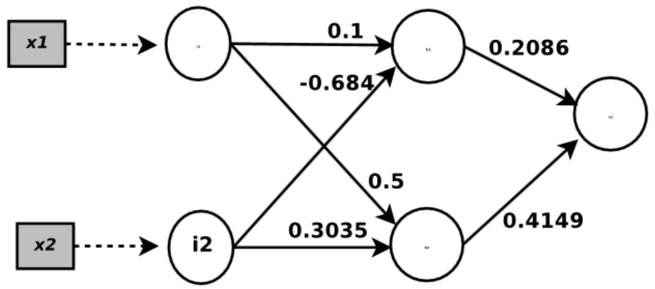
- Output layer weights
  - Neuron o_1
    - Real error obtained in neuron
      - Error :$t_1 - z_1 = 1 - 0{,}573 = 0{,}427$
    - new weights for neuron
      - **W*** = w* - η* Δf
      - $\Delta f = \delta * \text{input} = \delta * h$
      - $\delta = \sigma'(z) * \text{Error}'(z) = (z * (1\text{-}z)) * (t1 - z1)$
        - $(0{,}573 * (1 - 0{,}573)) * 0{,}427 = 0{,}1044$
      - $W^1{}_0 = W^1{}_0 + \eta * h1 * \delta = 0{,}2 + 0{,}25 * 0{,}332 * 0{,}1044 = 0{,}2086$
      - $W^1{}_1 = W^1{}_1 + \eta * h2 * \delta = 0{,}4 + 0{,}25 * 0{,}574 * 0{,}1044 = 0{,}4149$



Weight adjustment by backpropagation of the error



- Hidden Layer Weights
  - δ =0,1044
    - **W*** = w* - η* Δf
    - Δf = $\delta_{prev} * w_{prev}$ * σ'(z)* input = $\delta 1$ * input
- Neuron h1
  - Estimated error
    - $\delta 1 : h1*(1-h1)*(W^1{}_0*0{,}1044) = = 0{,}332*(1-0{,}332)*(0{,}2*0{,}1044) = 0{,}046$
  - new weights for neuron
    - $W^0{}_0 = W^0{}_0 + \eta * i1 * \delta 1 = 0{,}1 + 0{,}25*0*0{,}046 = 0{,}1$
    - $W^0{}_1 = W^0{}_1 + \eta * i2 * \delta 1 = -0{,}7 + 0{,}25*1*0{,}046 = -0{,}684$
- Neuron h2
  - Estimated error
    - $\delta 1 : h2*(1-h2)*(W^1{}_1*0{,}1044) = 0{,}574*(1-0{,}574)*(0{,}4*0{,}1044) = 0{,}0102$
  - new weights for neuron
    - $W^0{}_2 = W^0{}_2 + \eta * i1 * \delta 1 = 0{,}5 + 0{,}25*0*0{,}0102 = 0{,}5$
    - $W^0{}_3 = W^0{}_3 + \eta * i2 * \delta 1\ 0{,}3 + 0{,}25*1*0{,}0102 = 0{,}3025$
- New network
  - The output for the same input with these new Weights = 0,576 → less error, we have improved



Hyperparameters in neural networks

So far..

- Until now, apart from having to choose the number of layers, the number of neurons per layer, the activation functions in each layer / neuron ... we have found 2 hyperparameters:
  - the learning rate (η), which we said can be estimated with
    - SGD
    - Momentum
    - NAG
    - Adagrad
    - Adaddelta
    - Rmsprop
  - The number of batches and their size when training
- Another hyperparameter widely used in neural networks and machine learning is the desired level of complexity, i.e regularization

Regularization

- In machine learning, learning consists of finding the coefficients that minimize a cost function.
- Regularization consists of adding a penalty to the cost function. This penalty produces simpler models that generalize better. Models that are excessively complex tend to over-fit. That is, to find a solution that works very well for training data but very bad for new data. We are interested in models that, in addition to learning well, also perform well with new data.
- The most used regularizations are:
  - Lasso (also known as L1)
  - Ridge (also known as L2)
  - ElasticNet that combines both Lasso and Ridge.
- How does regularization work?
  - When we studied the gradient descent, we used the mean square error as the cost function F.
    - F = MSE
  - When we use regularization, we add a term that penalizes the complexity of the model. In the case of the MSE, we have:
    - F = MSE + $\alpha \cdot$C

  - C is the measure of complexity of the model. Depending on how we measure complexity, we will have different types of regularization. The hyperparameter α indicates how important it is to us that the model is simple in relation to how important its performance is.

Lasso regularization (L1)

- In Lasso regularization, also called L1, the complexity C is measured as the mean of the absolute value of the model's coefficients.
- This can be applied to linear regressions, polynomials, logistic regression, neural networks, support vector machines, etc.
- Mathematically it would be:

$$C = \frac{1}{n}\sum_{j=1}^{N}\left|w_j\right|$$

- When is Lasso (L1) effective?
  - Lasso will help us when we suspect that several of the input attributes (features) are irrelevant. By using Lasso, we are encouraging the solution to be less dense. That is, we favour that some of the coefficients end up being 0. This can be useful to discover which of the input attributes are relevant and, in general, to obtain a model that generalizes better. Lasso can help us, in this sense, to make the selection of input attributes. Lasso works best when attributes are not highly correlated with each other.
- Lasso is a training method by itself:
  - With MSE we will be minimizing the following cost function

$$F = \frac{1}{m}\sum_{i=1}^{M}(y_i - \overline{y_i})^2 + \alpha\,\frac{1}{n}\sum_{j=1}^{N}\left|w_j\right|$$

Ridge regularization (L2)

- In Ridge regularization, also called L2, the complexity C is measured as the mean of the square of the coefficients in the model.
- Mathematically it would be:

$$C = \frac{1}{2n}\sum_{j=1}^{N} {w_j}^2$$

- When is Ridge (L2) effective?
  - Ridge will help us when we suspect that several of the input attributes (features) are correlated with each other. Ridge makes the coefficients smaller. This decrease in the coefficients minimizes the effect of the correlation between the input attributes and makes the model more generalizable. Ridge works best when most of the attributes are relevant.

ElasticNet regularization (L1 and L2)

- ElasticNet combines the L1 and L2 regularizations. With the parameter r we can indicate the relative importance of Lasso and Ridge respectively.
- Mathematically:

$$C = r \cdot Lasso + (1 - r) \cdot Ridge$$

- When is ElasticNet effective?
  - We will use ElasticNet when we have a large number of attributes. Some of them will be irrelevant and others will be correlated with each other.

More on neural networks

So far our programming could look like this..

```python
import numpy as np
from keras.layers import Dense
from keras.models import Sequential

n_cols = predictors.shape[1]
model = Sequential()
model.add(Dense(100, activation='relu', input_shape=(n_cols,)))
model.add(Dense(100, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(predictors, target, validation_split=0.3)
```

Add layers

Sequential = each layer has weights/connections that belong only to the one layer after it in the network. There are more complex models.

Dense= all the nodes in the previous layer connect to all the nodes in the current layer.
100 = n° of nodes per layer;
input_shape = (shape of input, empty= undefined n° of rows/datapoints)

Validation_split= for cross-validation

Optimizer = controls the learning rate; adam = good algorithm to tune the learning rate, it adjust the learning rate as it does the gradient descent.
Loss = function, for regression we can choose MSE

Scaling data prior can help

```python
from keras.models import load_model
model.save('model_file.h5')
my_model = load_model('my_model.h5')
predictions = my_model.predict(data_to_predict_with)
my_model.summary()
```

For classification models

- Changes
  - The loss function
    - The most common log loss (in python = `'categorical_crossentropy'`)
      - The lower the better
        - As it is hard to interpret we can add `'metrics = ['accuracy']'` in our compiler step to ease the diagnosis
  - Modify the last layer
    - Instead of one node..
    - It has a separate node for each possible outcome
    - Change the activation function in this last layer to **softmax**
      - Softmax ensures that predictions sum 1, so that they can be interpreted as **probabilities**
        - When computing accuracy with the model's .evaluate() method, your model takes the class with the highest probability as the prediction. np.argmax()
  - Have the output data transformed to one hot encoding

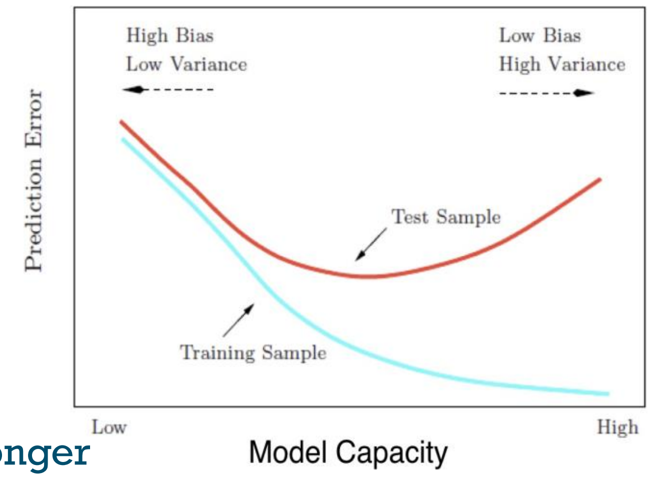
Early stopping

- By default we normally have 10 epochs
- We can add more (more training) but control that things are improving…if not include and early stopping
- With early stopping we indicate the number of epochs we allow not to have improvements before we stop the training

```
from keras.callbacks import EarlyStopping

early_stopping_monitor = EarlyStopping(patience=2)

model.fit(predictors, target, validation_split=0.3, nb_epoch=20,callbacks = [early_stopping_monitor])
```

How to choose an architecture?

- Model capacity
  - The more complex more capacity, but more overfitting
  - Workflow for optimizing model capacity
    - Start with a small network
    - Gradually increase capacity
    - Keep increasing capacity until validation score is no longer improving
    - Example



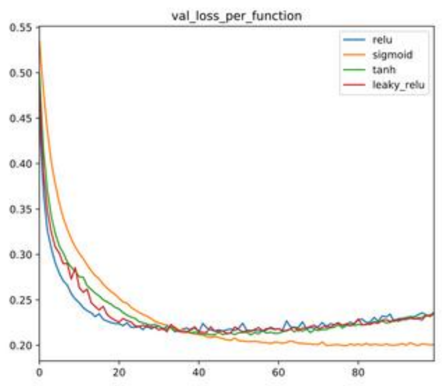
Hidden Layers	Nodes Per Layer	Mean Squared Error	Next Step
1	100	5.4	Increase Capacity
1	250	4.8	Increase Capacity
2	250	4.4	Increase Capacity
3	250	4.5	Decrease Capacity
3	200	4.3	Done

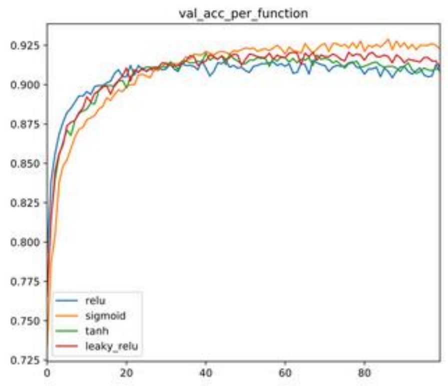
You can study different outputs for different activation functions

```python
# Activation functions to try
activations = ['relu', 'leaky_relu', 'sigmoid', 'tanh']


# Loop over the activation functions
activation_results = {}


for act in activations:
  # Get a new model with the current activation
  model = get_model(act_function=act)
  # Fit the model
  history = model.fit(X_train, y_train, validation_data=(X_test,y_test), epochs=100, verbose=0)
  activation_results[act] = history
```

- For every history callback of each activation function in activation_results:
  - The history.history['val_loss'] can be extracted.
  - The history.history['val_acc'] can be extracted.
  - And saved in two dictionaries: val_loss_per_function and val_acc_per_function

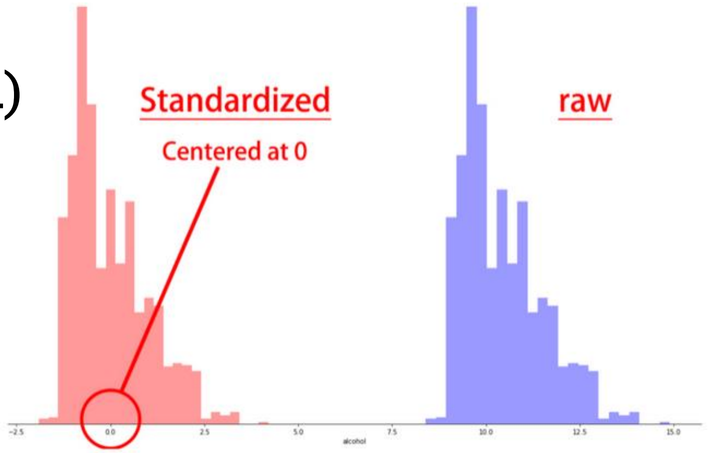
```python
# Create a dataframe from val_loss_per_function

val_loss= pd.DataFrame(val_loss_per_function)
# Call plot on the dataframe
val_loss.plot()

plt.show()
# Create a dataframe from val_acc_per_function
val_acc = pd.DataFrame(val_acc_per_function)


# Call plot on the dataframe
val_acc.plot()
plt.show()
```



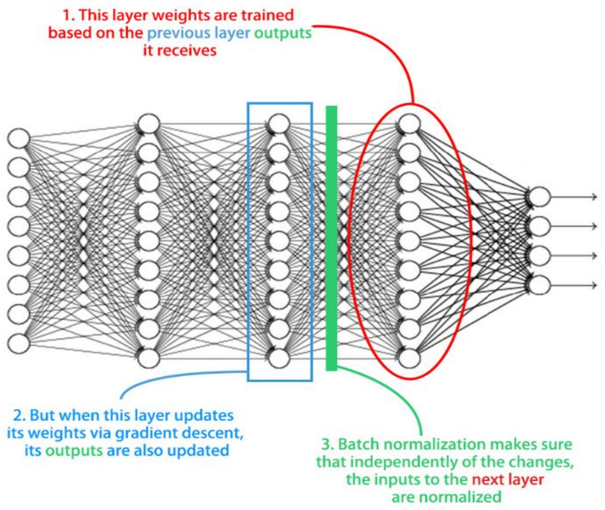


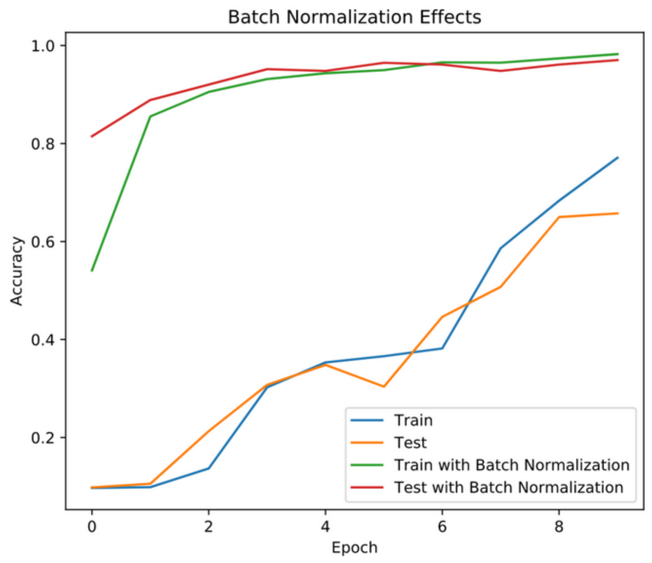
Batch normalization

- Standardization (a normalization approach)
  - $\frac{data - mean}{standard\ deviation}$
- Batch normalization advantages
  - Improves gradient flow
  - Allows higher learning rates
  - Reduces dependence on weight initializations
  - Acts as an unintended form of regularization
  - Limits internal covariate shift



```
from keras.models import BatchNormalization
model = Sequential()
model.add(Dense(100, activation='relu', input_shape=(n_cols,)))
 model.add(BatchNormalization())

model.add(Dense(1))
```


1. This layer weights are trained based on the previous layer outputs it receives
2. But when this layer updates its weights via gradient descent, its outputs are also updated
3. Batch normalization makes sure that independently of the changes, the inputs to the next layer are normalized



Neural networks summary

- Deep networks internally build representations of patterns in the data that are useful for making predictions and they find increasingly complex patterns as we go through successive hidden layers of the network
  - Modeller doesn't need to specify the interactions
    - When you train the model, the neural network gets weights that find the relevant patterns to make better predictions
- Partially replace the need for feature engineering or manually creating better predictive features
- Deep learning ~ Representation learning =Subsequent layers build increasingly sophisticated representations of raw data until we get to a stage where we can make predictions
- Advantages
  - Excellent predictors.
  - Adaptive (online learning)
- Disadvantages
  - Costly training.
  - Finding appropriate architecture can be difficult
    - Number of hidden layers / nodes in each hidden layer
    - Activation function
  - Determining the hyperparameters for the optimization
    - Type of optimization
      - In SGD: **learning rate**, size of mini-batches, momentum term, strength of regularization terms, …
  - Difficult interpretation.