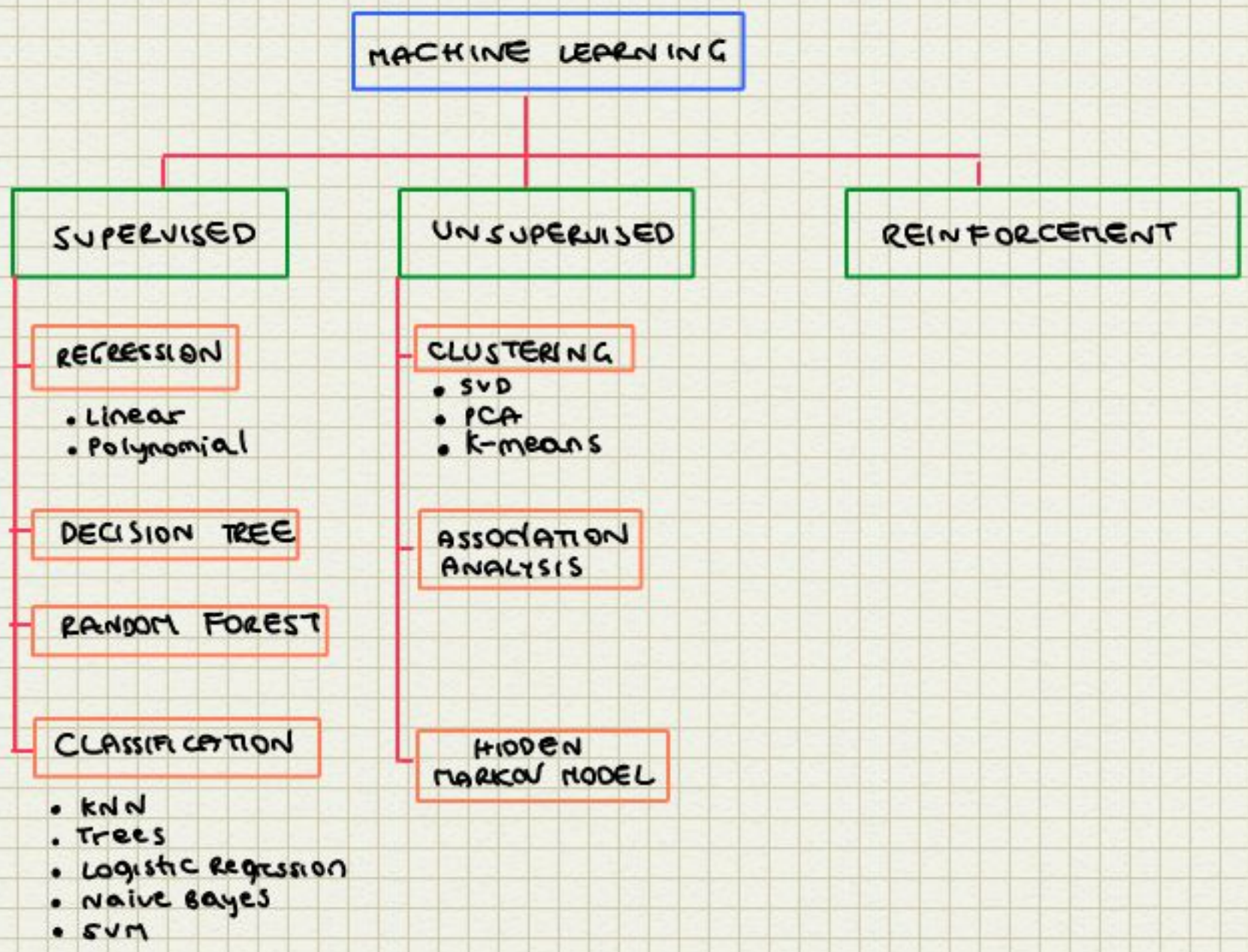
Machine Learning notes

Machine learning (University of Oxford)

MACHINE LEARNING

INTRODUCTION

MACHINE LEARNING is defined as a set of methods that can detect automatically patterns in data, and then use the uncovered patterns to predict future data or perform other kinds of decision making under uncertainty (such as planning how to collect more data).



Machine Learning is useful in many tasks:

- Classification
- Recognized patterns
- Predicting events
- Finding optimal strategies
- Information retrieval
- Recommender systems
- Vision: Object/Face Recognition, Image segmentation

CHAPTER 1: SUPERVISED LEARNING

1.1 DESIGN LEARNING SYSTEM

Any computer program improves its performance at some tasks through experience.

DEFINITION: A computer program learn from EXPERIENCE E with respect to some class of TASKS T and PERFORMANCE MEASURE P if its performance at tasks in T, as measured by P, improves with experience E

Examples:

A checkers learning problem:
- TASK T: playing checkers games
- performance measure P: percent of games won against opponents
- Training experience E: playing pratice games against itself

A handwriting recognition learning problem:
- TASK T: recognizing and classifying handwritten words within images
- performance measure P: percent of words correctly classified
- Training Experience E: a database of handwritten words with given classifications

A human driving learning problem:
- TASK T: driving on public four-lane highways using vision sensors
- performance measure P: average distance traveled before an error (as judged by human overseer)
- Training experience E: a sequence of images and steering comands recorded while observing a human driver

LEARNING ALGORITHM is able to learn from data

Example: a database system improves its performance at answering database queries based on the experience gained from database updates.

The 3 main ingredients are:

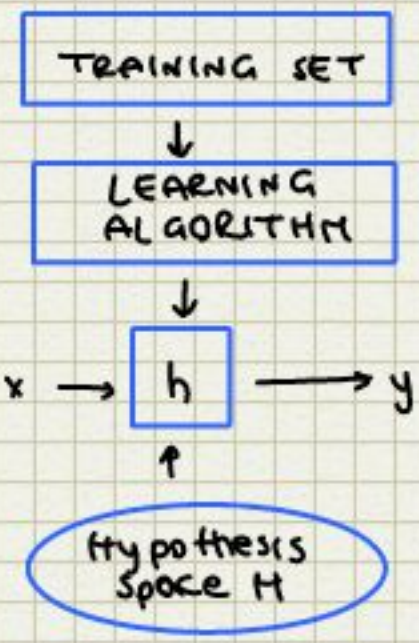
1) TASK is defined by the problem we want to tackle and the desired output

2) MEASURE PERFORMANCE: how good is the function/model returned by the learning algorithm

3) EXPERIENCE is provided by the available data

SUPERVISED LEARNING

The aim of SL is to build a model that is able to predict the target variable. If the target variables consists of categories, we call the learning task CLASSIFICATION. Alternatively, if the target is a continuously varying variable (price of a house), it's a REGRESSION task.



- TRAINING DATA is the subset of all possible data that can be derived by a particular experience of phenomenon
- HYPOTHESIS SPACE H is the set of functions which can be implemented by the ML system.
  We assume that the function to be learned f may be represented/approximated by the hypothesis $h \in H$
- LEARNING ALGORITHM can be seen as a search algorithm to H

1.2 LINEAR REGRESSION

1.3 GRADIENT DESCENT

We consider a simple LINEAR REGRESSION MODEL

HYPOTHESIS $h_\theta(x) = \theta_0 + \theta_1 x$ PARAMETERS: θ_0, θ_1

COST FUNCTION: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

I want to iterate the process to change the values of θ_0 and θ_1 in order to move to a point closed to the minimum of $J(\theta_0, \theta_1)$.

Then we want to apply and implement some algorithm for automatically finding the values of θ_0 and θ_1.

In other words, we need an efficient algorithm that minimizes the cost function J.

GRADIENT DESCENT is a very general algorithm to solve this optimization/minimization problem, that corresponds to the minimization of $J(\theta_0, \theta_1)$

Objective: $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

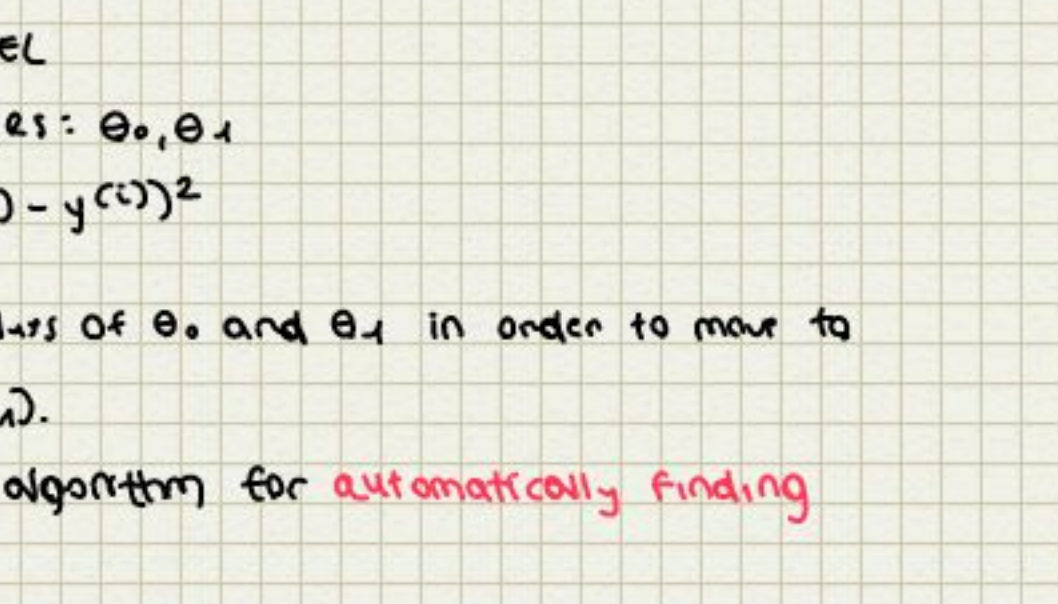
Once we defined a cost function $J(\theta_0, \theta_1)$, we want to minimize it over the parameters. In order to do this:

1. start with some configuration of parameters θ_i's
2. Try to adjust the configuration of parameters, so we keep changing θ_i's parameters in order to reduce the cost function J until we end up at a minimum.

Here we have a 3D plot, where there are the parameters θ_0 and θ_1, and the cost function $J(\theta_0, \theta_1)$.

It's a reasonable shape for an optimization problem using regression model.

The idea of GRADIENT DESCENT algorithm can be visualized starting by a random position, from which we want to reach the minimum. In this problem we can have 2 possible optimum.



We try to figure out a sequence of steps in order to reach the final goal of one of the minimums.

Given a specific cost function $J(\theta_0, \dots, \theta_n)$, we take a small step in direction of the negative gradient (that allows to go from the very top to bottom mountain), so that:

UPDATE RULE: $\theta_{k+1} = \theta_k - \eta \nabla J(\theta_k)$

- Starting from this particular configuration of parameters at time step K, θ_K, the update rule (consist in the value of θ in the next step) is given by the previous assignment minus the result of a parameter $\eta > 0$, known as LEARNING RATE, times the output of the gradient of the cost function on parameter θ_K.
- After each update, the gradient is re-evaluated for the new weight vector θ_{K+1} and the process is repeated several times

NOTE: the cost function is computed on the entire training set in order to evaluate ∇J (batch method)

A simple implementation is

repeat until convergence {

$$\theta_j = \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \text{(from } j=0 \text{ and } j=1\text{)}$$

}

The parameter θ_j is going to be updated according to the previous rule.
At each step, by looking all parameters in the training set, we update the values of parameters θ by computing the partial derivative with respect to each of parameters θ_0 and θ_1.

Those parameters should be simultaneously updated.

To do this, we need to use 2 variables tmp0 and tmp1. Then after I can update according to θ_0 and θ_1

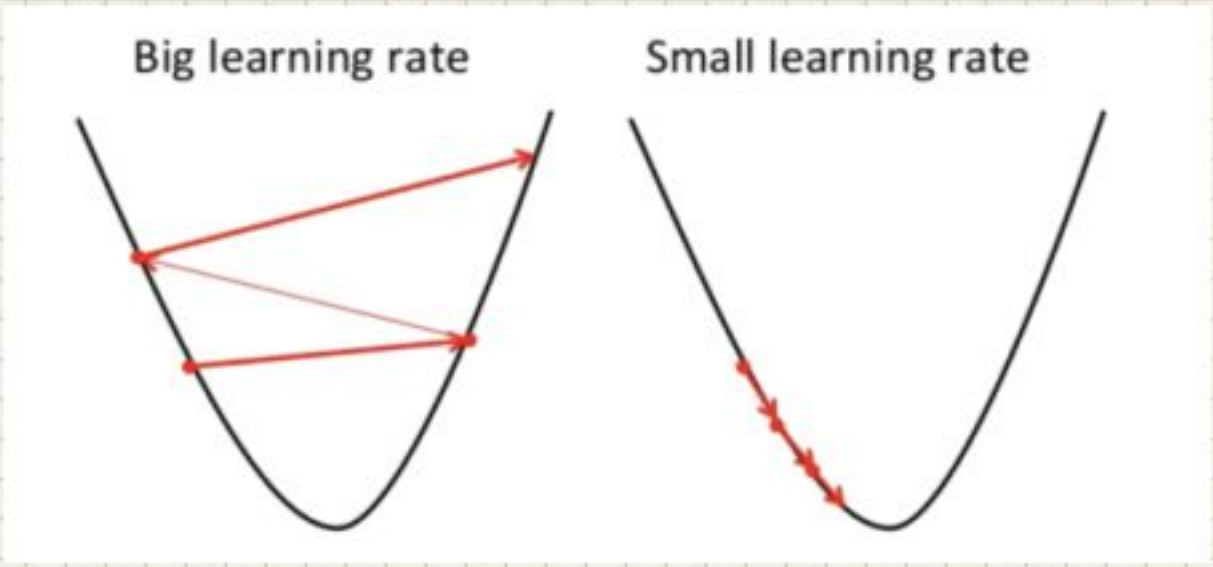
$$tmp0 := \theta_0 - \eta \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

$$tmp1 := \theta_1 - \eta \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

$$\theta_0 := tmp0; \quad \theta_1 := tmp1;$$

We need a proper learning rate η before gradient descent starts

An important parameter in GRADIENT DESCENT is LEARNING RATE, which determine the size of each step.
When learning rate is too big, gradient descent may jump across the valley and end up on the other side. This will lead to the cost function to diverge. If it's too small, it will take long time to converge.



1.4 LINEAR CLASSIFICATION AND LOGISTIC REGRESSION

The goal in CLASSIFICATION is to termine to which discrete category a specific example belongs to.
Some examples can be:
- Email : spam vs not spam
- Online transactions : fraudulent vs not fraudulent
- Tumor : malignant vs benign

GOAL : define a model, that is able to classify training examples and then test examples, following a certain procedure.

The outputs are a set of discrete values. They are usually called labels or classes.

Example: yes/no , 1/2/3/.../9 , cat/dog/person/...

BINARY OUTPUT — SET OF FINITE VALUES — CATEGORIES

Moreover we are interested in finding a good hypothesis function, that is a good approximation of our ideal target function f, which maps our features X to our output Y. $h \sim f : X \to Y$ where Y is categorical (while in regression $Y = R$)

There are usually 2 types of classification:

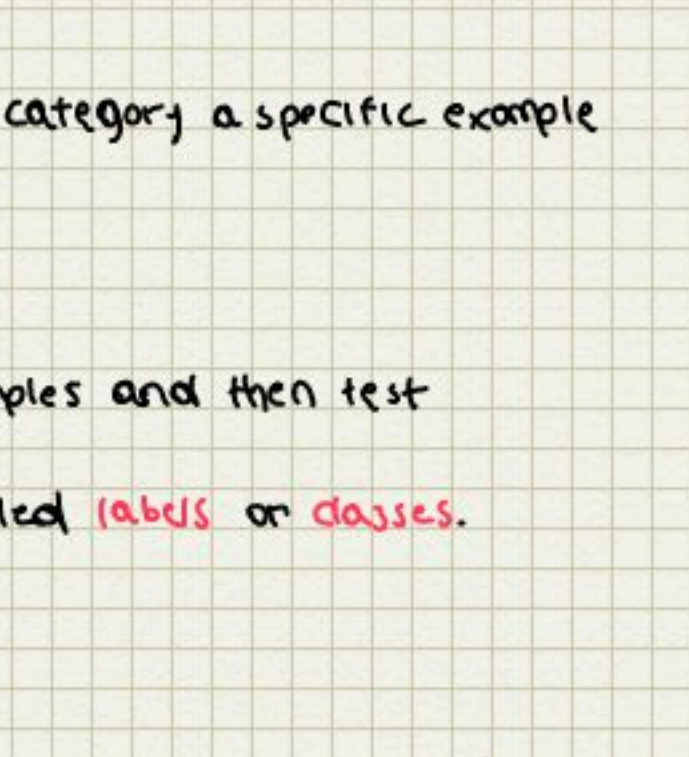
1. BINARY : output is a set of two possible labels
2. MULTI-CLASS : output is a set of multiple possible labels

1.4.1 BINARY PROBLEMS

We design a binary classification model using linear regression.

- Example: Try to develop a spam filter

Our classes are spam vs non-spam
The red points are samples belonging to the 2 different classes.



Our hyphotesis function would be represented by the linear regression model.
It's a reasonable output after we applied Gradient Descent in order to set and train our parameter θ.

Once we have a linear model, we need to add to the linear regression system a threshold classifier.
By setting a threshold such as 0.5, a value between 0 and 1, our decision rule would just select a right threshold and would be :

$$y = \begin{cases} 1 & \text{if } h_\theta(x) \geq 0.5 \quad \text{POSITIVE CLASS} \\ 0 & \text{if } h_\theta(x) < 0.5 \quad \text{NEGATIVE CLASS} \end{cases}$$

If we add a new sample, it's not the output expected. If we train a new linear regression model, the new straight line will be more similar to the line in purple.
By setting the threshold, we have a problem: in the previous example two points were classified as positive class, now they are misclasified.

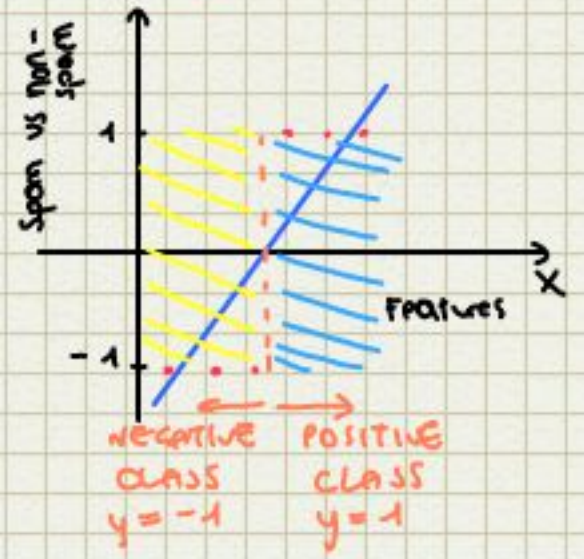
- We can try to use a different notation in order to develop the decision rule:

$y = \text{sign}(h_\theta(x))$

Our 2 classes are defined as POSITIVE and NEGATIVE

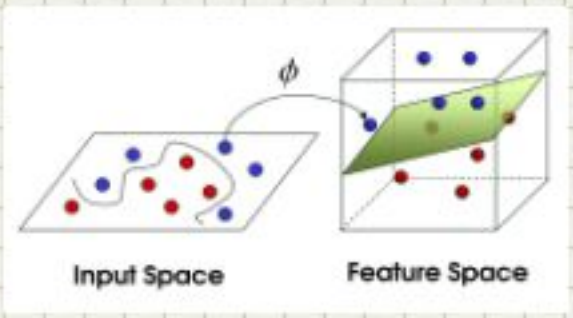
$$y = \begin{cases} 1 & \text{if } h_\theta(x) \geq 0 \\ -1 & \text{if } h_\theta(x) < 0 \end{cases}$$

This decision rule is based in linear regression.
So this specifies a linear classifier:



We are defining a linear boundary (threshold), which separate the space into 2 areas.
This case corresponds to 1D threshold.

- Example 2: In 2D space there is a line separating 2 half-spaces belonging to 2 different classes (red and blue points)



- Example 3: In 3D this decision boundary is a plane in the space.

LEARNING LINEAR CLASSIFIERS

LEARNING = we need to estimate a good decision boundary:

- find the right direction θ and location θ_0 of the boundary
- we need to define a criteria, that tell us how to select the parameters

There are 3 types of Loss (cost) functions:

- Zero|One: $J_{01}(\theta) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\{0 \text{ if } h_\theta(x^{(i)}) = y^{(i)}, 1 \text{ otherwise}\}}_{\text{cost}(h_\theta(x^{(i)}), y^{(i)})}$
- Absolute: $J_{abs}(\theta) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{|h_\theta(x^{(i)}) - y^{(i)}|}_{\text{cost}(h_\theta(x^{(i)}), y^{(i)})}$
- Squared: $J_{sqr}(\theta) = \frac{1}{2m}\sum_{i=1}^{m} \underbrace{(h_\theta(x^{(i)}) - y^{(i)})^2}_{\text{cost}(h_\theta(x^{(i)}), y^{(i)})}$

These loss functions can be a good choice. They depend on algorithm we apply in order to optimize the parameter.
If the choice is apply gradient descent, we need a loss function where we can compute derivative.
Then in this case Loss squared function is a good choice.

1.4.2 LOGISTIC REGRESSION

Often applying linear regression to classification tasks is not a great idea.

A better approach is LOGISTIC REGRESSION, that is one of the most popular classification algorithm.

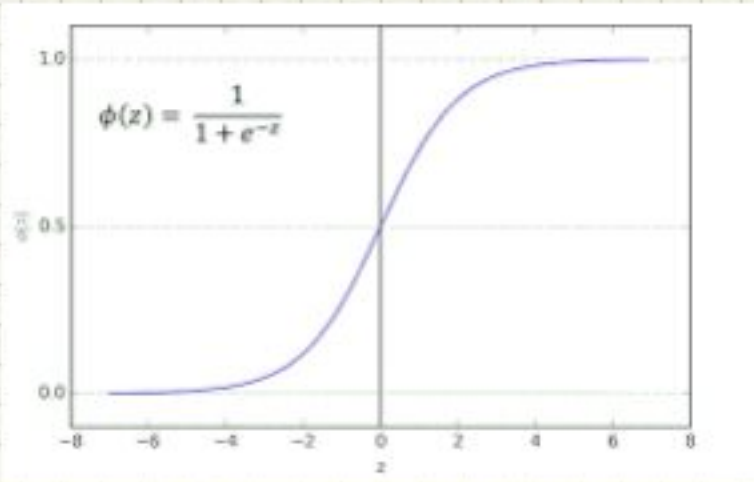
It has a property: the output of hypothesis function is always between 0 and 1

$0 \leq h_\theta(x) \leq 1$

$h_\theta(x) = g(\theta^T x)$

we define a new function g, that depends on linear regression model

where $g(z) = \frac{1}{1+e^{-z}} = \sigma(z)$ SIGMOID or LOGISTIC FUNCTION



The boundary is between 0 and 1 and this is the shape of SIGMOID FUNCTION.

we have a non-linear function, that goes to 1 at $+\infty$ and goes to 0 at $-\infty$.

The logistic regression threshold is set at $g(z) = 0,5$

$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$

we can assign a probabilistic interpretation at the output of the hypothesis function $h_\theta(x)$.

$h_\theta(x)$ can be interpreted as estimated probability of having $y=1$ on input x.

$h_\theta(x) = P(y=1 | x; \theta)$

Example: Discriminate between malignant vs benign tumors.

The features, that represent the problem

$X = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ \text{tumor\_size} \end{pmatrix}$ $h_\theta(x) = 0.7$ → Tell patient that 70% chance of tumor being malignant

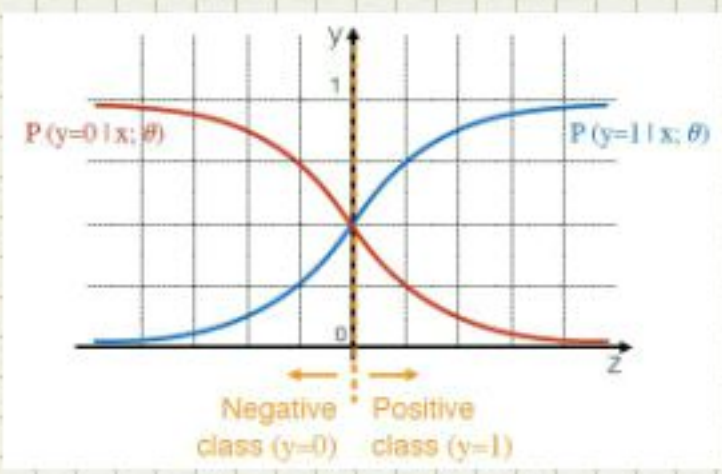
MARGINALIZATION PROPERTY of probability is $P(y=1 | x; \theta) + P(y=0 | x; \theta) = 1$

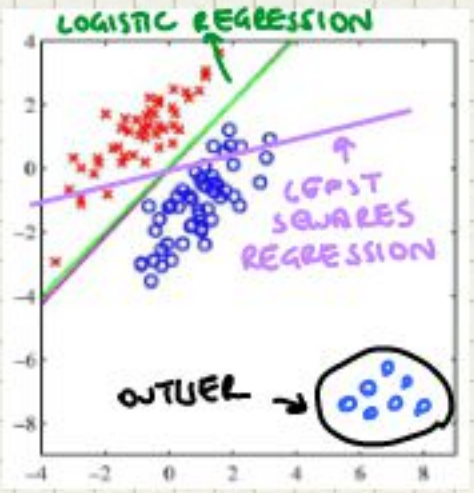
Then I can obtain $P(y=0 | y; \theta) = 1 - P(y=1 | x; \theta) = 1 - \frac{1}{1+e^{-\theta^T x}} = \frac{e^{-\theta^T x}}{1+e^{-\theta^T x}}$

The decision boundary for logistic regression is:

$$y = \begin{cases} 1 & \text{if } h_\theta(x) \geq 0 \\ 0 & \text{if } h_\theta(x) < 0 \end{cases}$$

we use a LINEAR decision boundary, that corresponds to 0.95, that separates the 2 half-spaces.





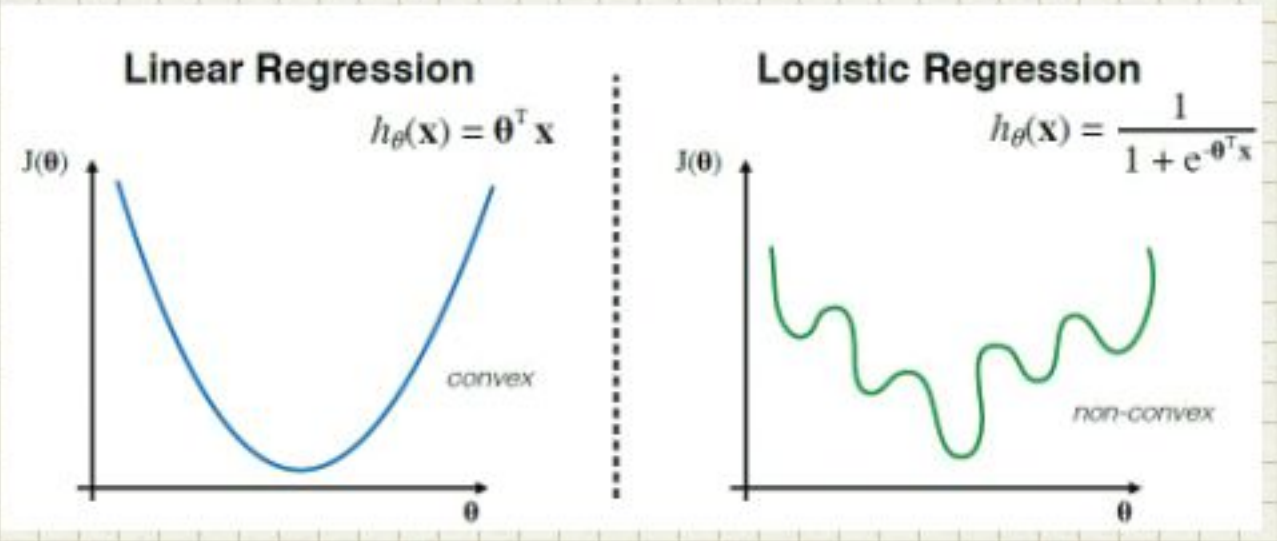
The red dots and the blue dots are slightly different.
There are 2 clusters of blue dots.
Moreover there are outliers far from the boundary.

The outliers have a huge impact on the decision boundary for the linear classification model, but its influence isn't the same for logistic model.

- The loss function of Linear Regression is:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{cost}(h_\theta(x^{(i)}) - y^{(i)})$$

where $\text{cost}(h_\theta(x^{(i)}) - y^{(i)}) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$



If we try to apply the same cost function defined by the Regression model the shape of $J(\theta)$ is going to be a non-convex function with multiple local optimals for Logistic Regression.

- Then loss function of Logistic Regression is $J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{cost}(h_\theta(x^{(i)}) - y^{(i)})$

where $\text{cost}(h_\theta(x^{(i)}) - y^{(i)}) = \begin{cases} -\log(h_\theta(x^{(i)})) & \text{if } y^{(i)} = 1 \\ -\log(1 - h_\theta(x^{(i)})) & \text{if } y^{(i)} = 0 \end{cases}$

A simplified notation is

$$\text{Cost}(h_\theta(x^{(i)}), y^{(i)}) = -y^{(i)} \cdot \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \cdot \log(1 - h_\theta(x^{(i)}))$$

$$\Rightarrow J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \cdot \log(1 - h_\theta(x^{(i)}))$$

It's a convex function!

We can learn our parameters with GRADIENT DESCENT.

$$\min_\theta J(\theta)$$

repeat until convergence {

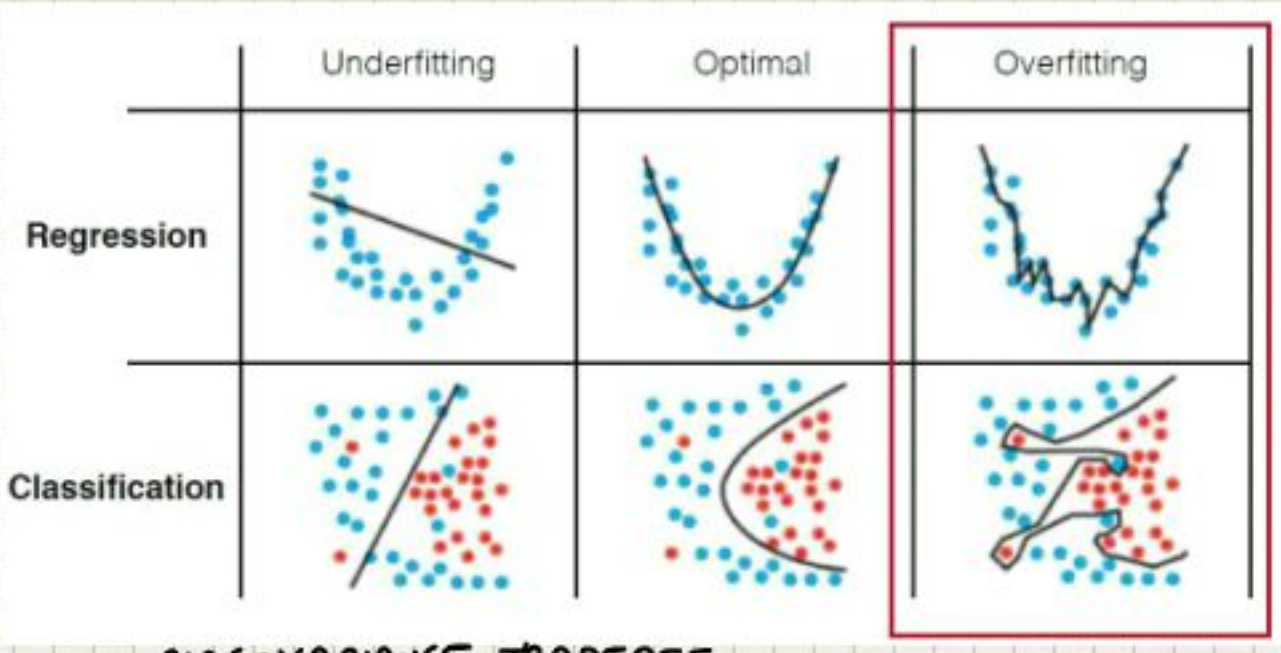
$$\theta_j = \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\theta) = \theta_j - \frac{\eta}{m}\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})\, x_j^{(i)}$$

(simultaneously update all θ_j)

}

1.5 REGULARIZATION & OVERFITTING

Regularization is one of the fondamental tools used to overcome one of the measure challenging in developing machine learning systems, that is overfitting.



BIAS-VARIANCE TRADEOFF

we have a UNDERFITTING PROBLEM if we fit the data with a simple regression model. There are high bias, provided by the fact that the underlying assumption is that a linear model would be a good approximation of this distribution and it's not actually the case.

On the other hand, we may have OVERFITTING PROBLEM if we try to fit high order polynomial. So we are memorizing our training sample, we mimic the particular distribution of training points for this example.

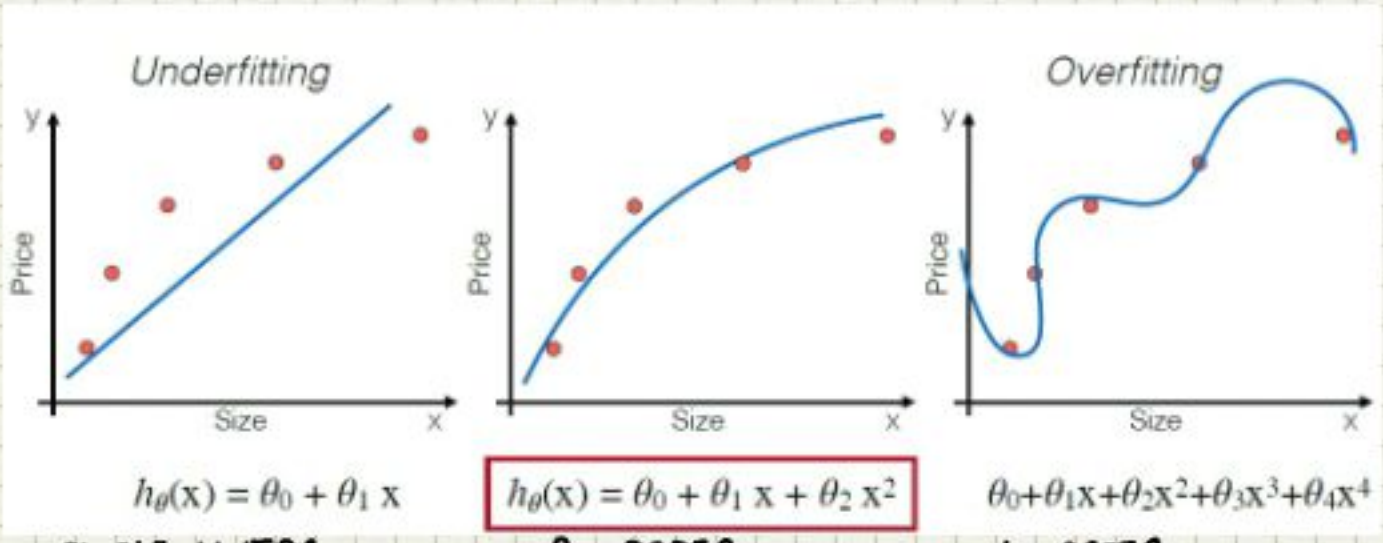
It's very bad because if you add new points, the OPTIMAL MODEL is going to be better than the overfitting regression, that is not able to generalize the new sample.

The same concepts are applied for classification.

In particular we focus on OVERFITTING PROBLEM.

We start with Linear Regression.

We assume to have 5 samples.



SINGLE LINEAR REGRESSION MODEL

2 ORDER POLYNOMIAL

4 ORDER POLYNOMIAL

- In underfitting case, we apply LR model and we learn the parameters with GRADIENT DESCENT. It's not the best solution
- In the optimal case, the hypothesis function is represented by the quadratic function. It's a better choice
- On the other hand, we may overfit if we use a high order polynomial (Ex: 4 order) we are fitting all the 5 training points with the 4 order polynomial. If we add a new point, we may have trouble.

How do I get the quadratic function starting from the simple linear model?

$$h_\theta(x) = \begin{pmatrix}\theta_0\\ \theta_1\\ \theta_2\end{pmatrix}^T \begin{pmatrix}x_0\\ x_1\\ x_2\end{pmatrix} = \begin{pmatrix}\theta_0\\ \theta_1\\ \theta_2\end{pmatrix}^T \begin{pmatrix}1\\ x\\ x^2\end{pmatrix}$$

x_1: size of house

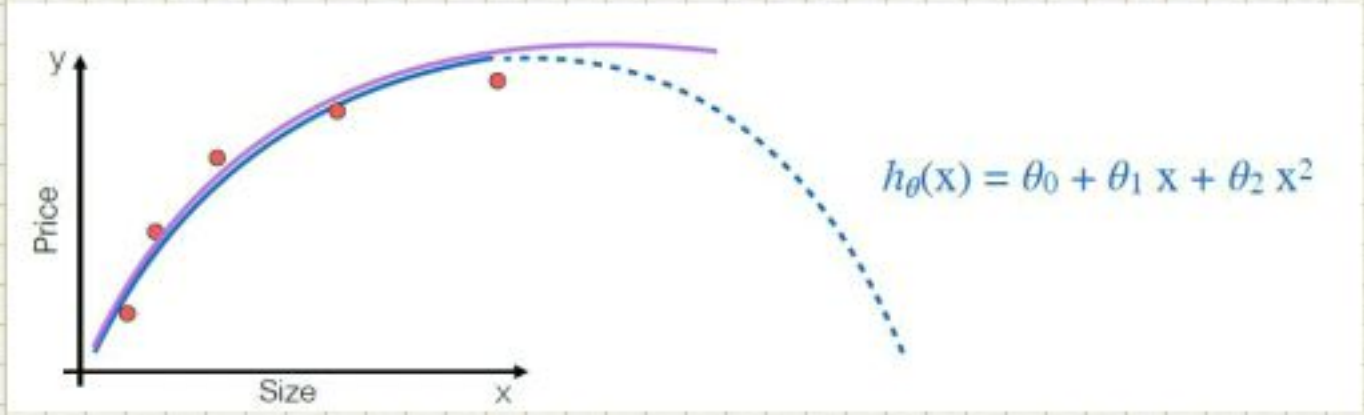
x_2: (size of house)^2

A simple way to plug NON-LINEAR FUNCTION in the linear model is applying simple FEATURE ENGINEERING. Starting from the original features, we introduce new features corresponding to x_2, obtained by computing the square of the size of the house.

In order to make it work, FEATURE SCALING / NORMALIZE FEATURE is important to have feature, that are comparable among them.

The same trick can be applied for any non linear function.

$$h_\theta(x) = \begin{pmatrix}\theta_0\\ \theta_1\\ \theta_2\end{pmatrix}^T \begin{pmatrix}1\\ x\\ \sqrt{x}\end{pmatrix}$$



Now we know how to plugin and obtain non-linear functions.

In real scenario with many features, the overfitting problem may happen. So the learned hypothesis function may fit the training set very well ($J(\theta) \approx 0$), but will fail to generalize to new samples (predict prices for new houses)

If we have many features and little training data, overfitting could be a problem.

To address this problem we have 2 options:

- Reduce the # of features through MODEL SELECTION.

We want to select among them the features that are more representative

- REGULARIZATION

The idea is to keep all the features, but reduce the influence of parameters θ_j.

When we learn the parameters, we would like to keep all them, but assign a difference weight / influence to them.

It works well when we have many features, each of which contributes a bit to predict y.

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \qquad h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cancel{\theta_3 x^3} + \cancel{\theta_4 x^4}$$

Example: focus on quadratic function and high order polynomial.

The idea of regularization is start from high order polynomial model to penalize the parameters assigned to the features.

I want a procedure, that gives very small values for θ_3 and θ_4 (close to 0)

The way the parameters are learned is through GRADIENT DESCENT

$$\min_\theta J(\theta) = \min_\theta \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + 1000\,\theta_3^2 + 1000\,\theta_4^2$$

we add additional terms, where the weights are the square of parameters.

The way to introduce a small cost to the original formulation is forcing θ_3 and θ_4 to be very small and multiplying them by huge numbers.

1.5.1 Regularized Linear Regression

HYPOTHESIS REPRESENTATION $h_\theta(x) = \theta^T x$ PARAMETERS $\theta = \begin{pmatrix}\theta_0\\ \vdots \\ \theta_n\end{pmatrix}$

OBJECTIVE $$\min_\theta J(\theta) = \min_\theta \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n}\theta_j^2\right]$$

EXTRA REGULARIZATION TERM will shrink (reduce) every single parameter

λ is the REGULARIZATION PARAMETER we can optimize in order to find the right assigment for all parameters θ.

We optimize all parameters $\theta_1, \dots, \theta_n$. In this way we are penalizing the higher order and more complicated hypothesis.

At the same time we want to find the parameters θ, that provide simpler solutions.

We can learn our parameters with GRADIENT DESCENT: $\min_\theta J(\theta)$

* repeat until convergence { (simultaneously update all θ_j)

$$\theta_0 := \theta_0 - \eta \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})\,x_0^{(i)}$$

$$\theta_j := \theta_j - \eta\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})\,x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] \quad (j = 1, \dots, n)\ \}$$

$$\theta_j := \theta_j\left(1 - \eta\frac{\lambda}{m}\right) - \eta\cdot\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})\,x_j^{(i)}$$

$\left(1 - \eta\frac{\lambda}{m}\right) < 1$ ⇐ it shrink the parameters θ_j

the second term is the same term before regularization

1.5.2 Regularized Logistic Regression

Our cost function was:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\cdot \log h_\theta(x^{(i)}) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))\right]$$

We can regularize this equation by adding a term

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\cdot \log h_\theta(x^{(i)}) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j$$

In the same way we can learn our parameters with GRADIENT DESCENT

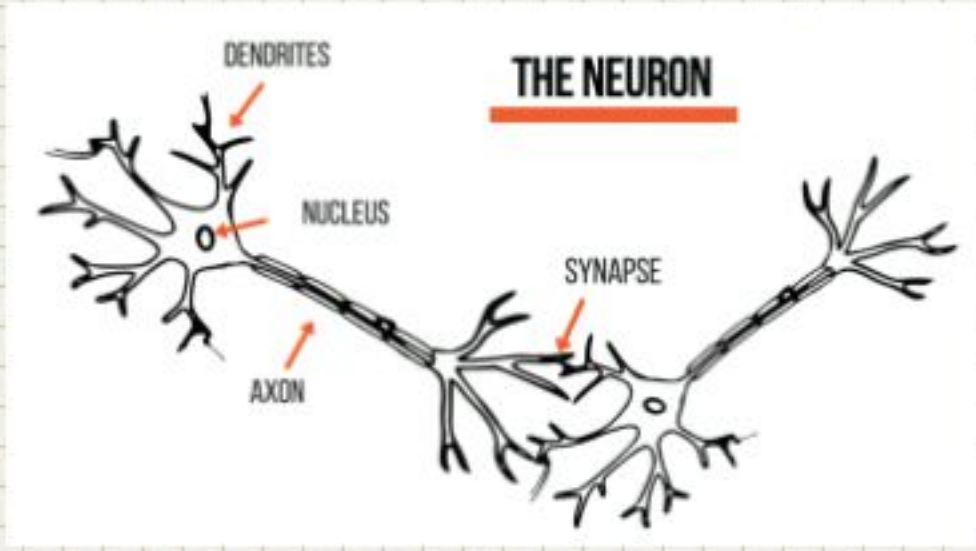
$\min_\theta J(\theta)$ *

1.6 MODEL SELECTION & EVALUATION

17 DIAGNOSING ML & KNN

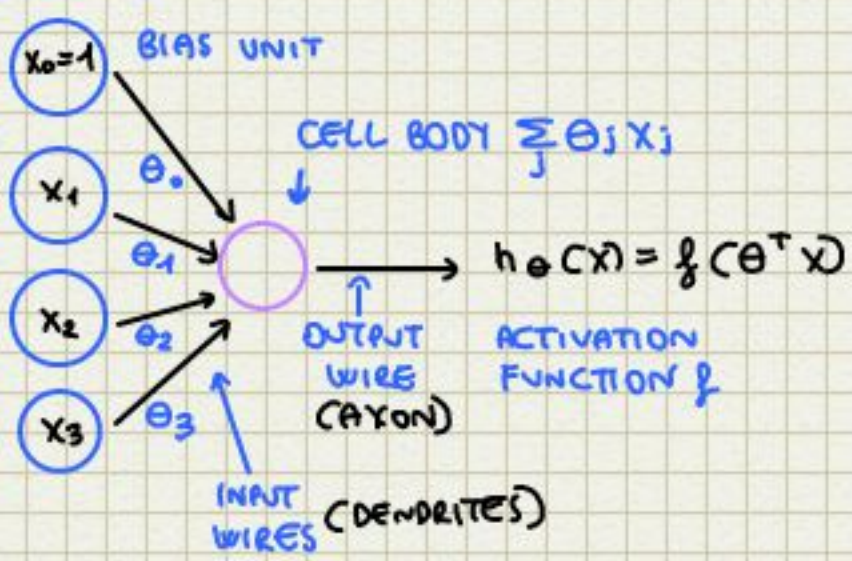
CHAPTER 2: NEURAL NETWORKS

NEURAL NETWORKS, commonly known as ARTIFICIAL NEURAL NETWORKS are computing systems, that are inspired by the functioning and the structure of the human brain. Before understanding what is NN, we have to first understand how neurons communicate with each other in a brain.



The HUMAN BRAIN is composed of billions of cells called NEURONS.
Information travel in electric signals inside neurons. Any information that needs to be communicated to any other part of the brain is gathered by the DENDRITES in a neuron, which is then processed in the neuron cell body and is passed to other neurons through the AXON.

SO now let's try to understand what is NN



$$x = \begin{pmatrix} x_0 \\ x_n \end{pmatrix}$$ input

$$\theta = \begin{pmatrix} \theta_0 \\ \theta_n \end{pmatrix}$$ parameters (weights)

As the human brain, the ARTIFICIAL NEURAL NETWORKS are large networks, where multiple simple units (/artificial neurons) are connected together, following the same principle. Above we defined a simple model for an artificial neuron, that is the COMPUTATIONAL UNIT.

In the neural network representation of this model there are n inputs, a single artificial neuron and a single output.

$\theta_0, \theta_1, \theta_2, \theta_3$ are represented as arrows and are the WEIGHTS

The artificial neuron is doing the linear combination, calculated by multiplying the weights to their respective inputs and summing them up.

With the input $x_0 = 1$, we are working on a LINEAR CLASSIFICATION MODEL. (assigned by default)

After the linear combination of inputs, the artificial neuron is computing a

NON-LINEAR function f, called ACTIVATION FUNCTION, applied to the $\Sigma x_i w_i$.

The most used non-linear activation functions are:

- Perceptron is the first neural network model introduced, in which:

$$h_\theta(x) = \begin{cases} 1 & \text{if } \theta^T x > 0 \\ 0 & \text{otherwise} \end{cases}$$

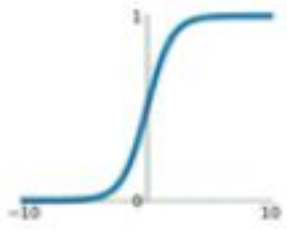
- Logistic unit, refered also as sigmoid function, is the activation function of the Logistic Regression model:

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

Activation Functions

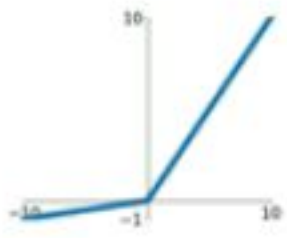
Sigmoid

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



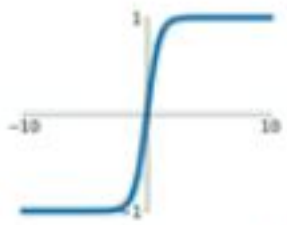
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$



Maxout

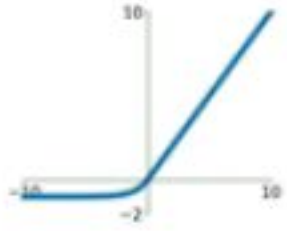
$\max(w_1^T x + b_1, w_2^T x + b_2)$

→ **ReLU**

$\max(0, x)$

→ **ELU**

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

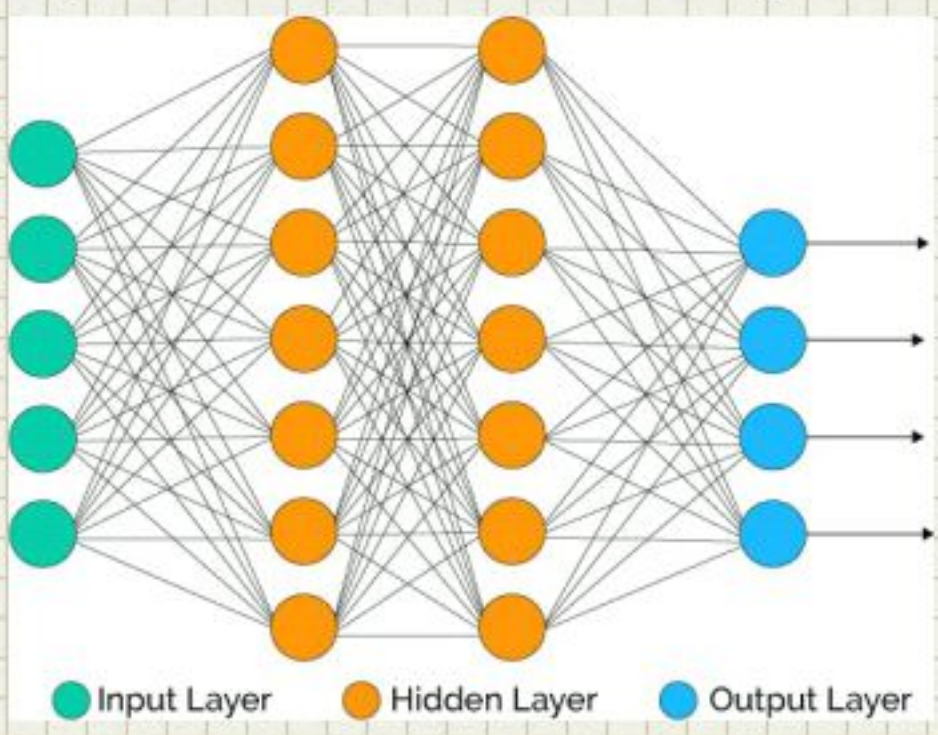


In general an ARTIFICIAL NEURAL NETWORK is a group of different neurons strong together. We can connect multiple istances (computational units) together.

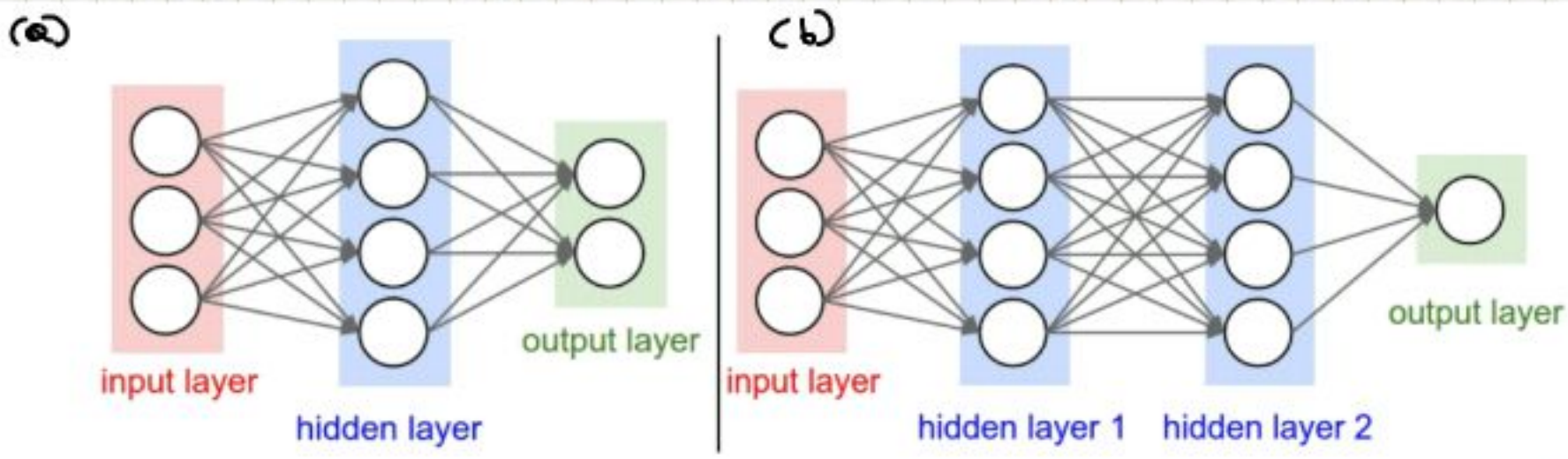
Neurons are connected together in layers.

There are three types of layers:

- INPUT LAYER is composed by INPUTS
- HIDDEN LAYER is the intermediate layer between the input layer and the output layer. It takes in a set of weighted inputs and produces output through an activation function. There can be one or more hidden layers depending on the complexity of the problem
- OUTPUT LAYER is the last layer of ANN architecture that produces output for a particular problem.



These 3 layers are referred as FULLY CONNECTED LAYERS, are specialized layers designed for specific tasks.



Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. **Right:** A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

(a) has 4+2 neurons. How many learnable parameters?

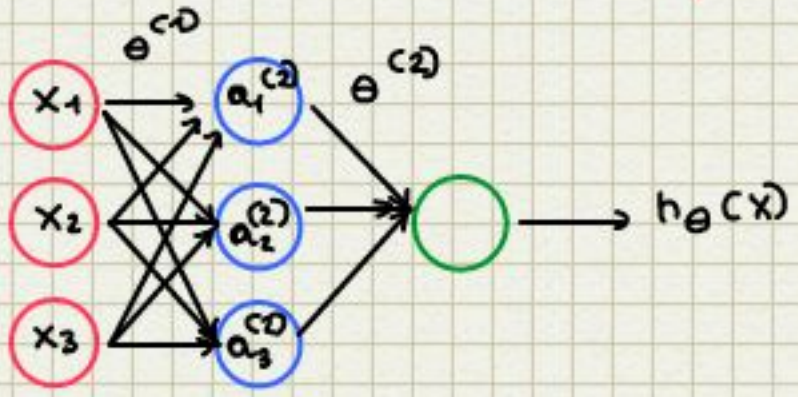
#parameters = 3x4 + 4x2 + 6 (biases) = 20 + 6 = 26

for the additional term $x_0 = 1$

(b) has 4+4+1 neurons

parameters = 3x4 + 4x4 + 4x1 + 9 (biases) = 32 + 9 = 41

NEURAL NETWORKS: model representation



- $a_i^{(j)}$ is the activation of unit i in layer j
- $\theta^{(j)}$ is the matrix of weights controlling function mapping from layer j to layer $j+1$

Neural networks working process can be broken down in 3 steps:

1) Initialization

The first step after designing a neural network is initialization:

- Initialize all weights $\theta^{(j)}$ to small random values
- set all the biases to 1

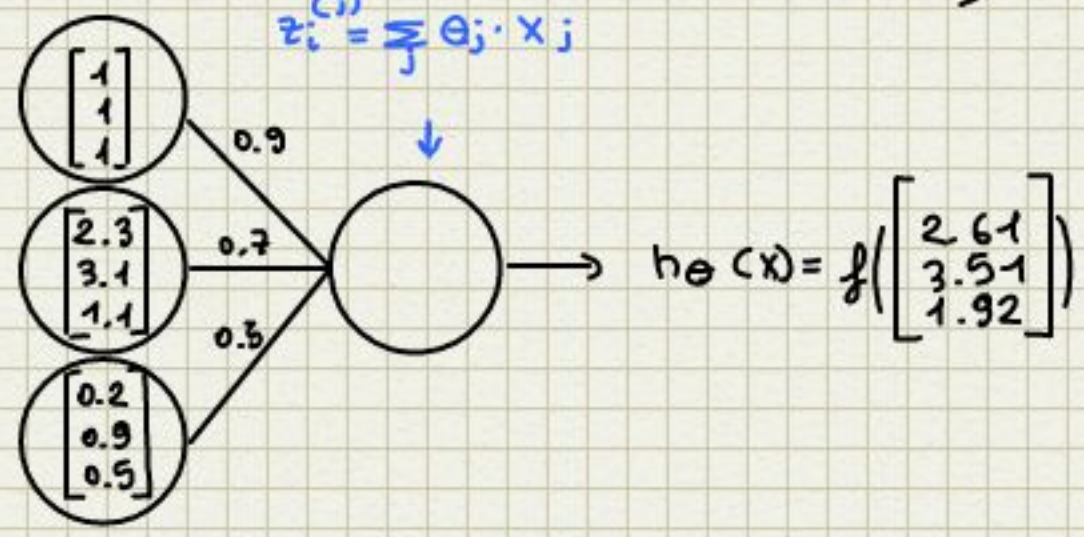
There are some common initialization techniques

2) Feedforward

Feedforward is the process NN use to turn the input into an output

- Input units are set by an exterior function, which causes their output links to be activated.
  We work forward through the network: from layer j to layer $j+1$ we pass the output of layer j, that is going to be the input for layer $j+1$
- Then each output is the weighted sum of the activation on the links feeding into a node.

feedforward →

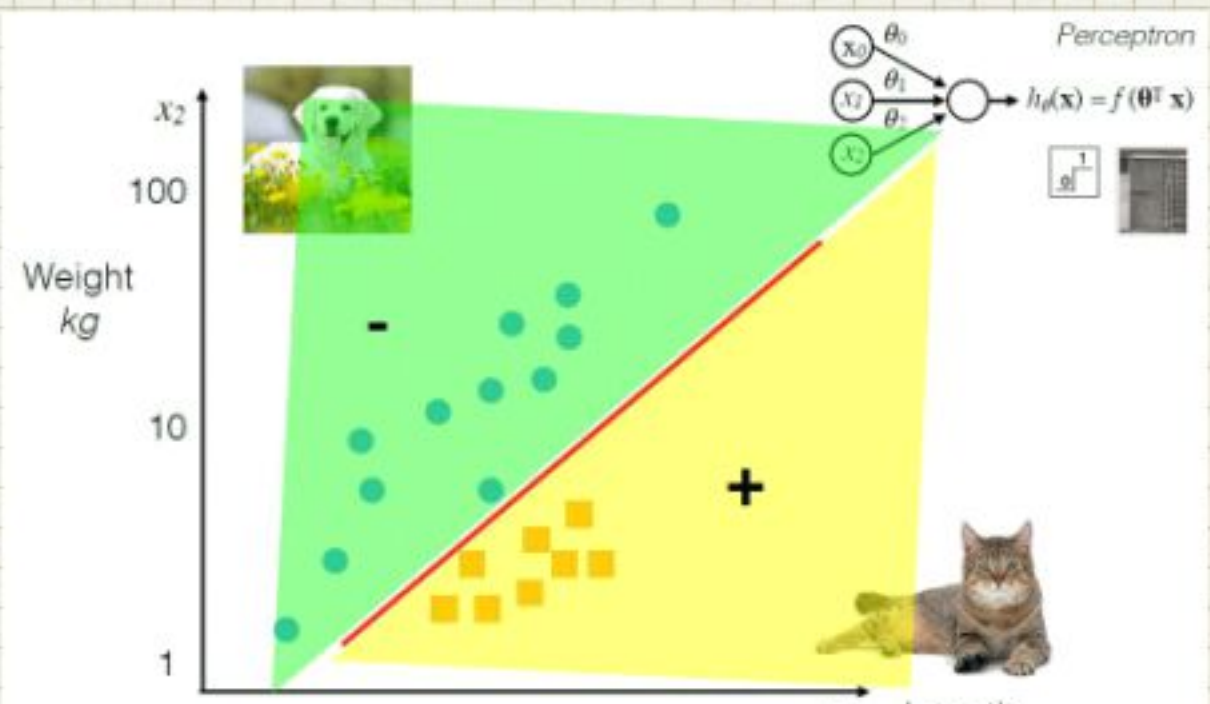


- The activation function transform this linear combination (Example: non-linear function sigmoid)

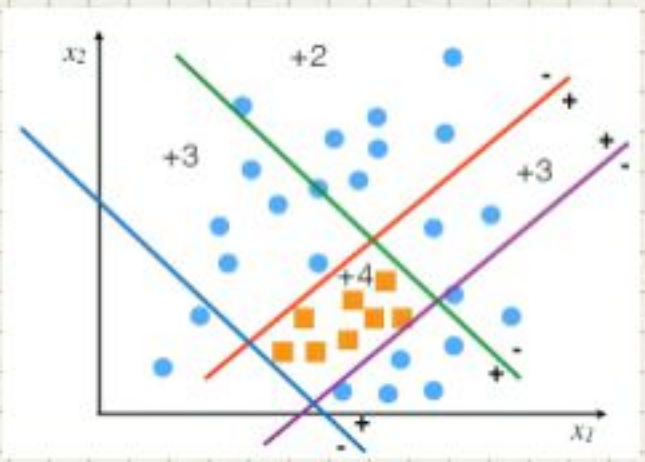
This function corresponds to the threshold of that node, applied to the input

$z_i^{(j)}$ is the linear combination of the input nodes

Example 1: Binary classifier to recognize cats and dogs, based on two simple features, weight and length.
In general cats are smaller in terms of weight and length with respect to dogs.
Moreover cats are less diversified.
So it should be easy to get a good approximation, that is able to separate these 2 cases.



Example 2: Binary classifier that recognize cats vs non-cats, based on two features, x_1 and x_2. We can observe that we don't have anymore a linearly separable task.
Now I can learn different units, that are specialized between different parts of feature space.
The red line is the linear classification model, that discriminates cats from dogs.



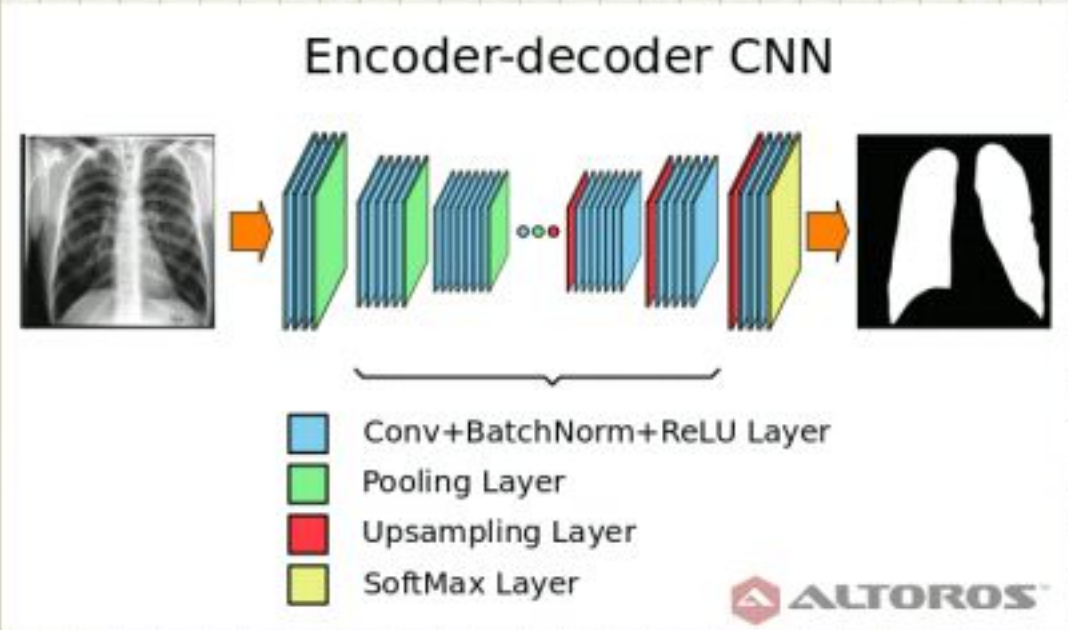
The purple line is able to separate large animals vs small animals.
The same for the green and blue lines.
If all these models are correctly separating the space in order to get in the right space the cats sample, we can sum the output of all these 4 lines.
In the end we get a classification of cats, even if it's not a linearly separable case.

In NN the output neuron computes the sum of agreements between all these hidden neurons (the colored lines)

DEEP NEURAL NETWORKS are networks that apply the feed-forward computation.

The main difference from other NN is that the designed architecture have a lot of intermediate hidden layers.

The applications of deep neural networks are visual recognition, visual classification on large datasets.



There are 2 strategies for multiclass classification problems:

- One-vs-one: we train $K(K-1)/2$ binary classifiers

  Each binary classifier is trained to discriminate between 2 classes. At prediction time we apply a voting scheme

- One-vs-all: we train a classifier per class.

  Each binary classifier is trained using its positive samples and samples belonging to all other classes as negative.

  This strategy requires each classifier to produce a real-valued confidence score for its decision

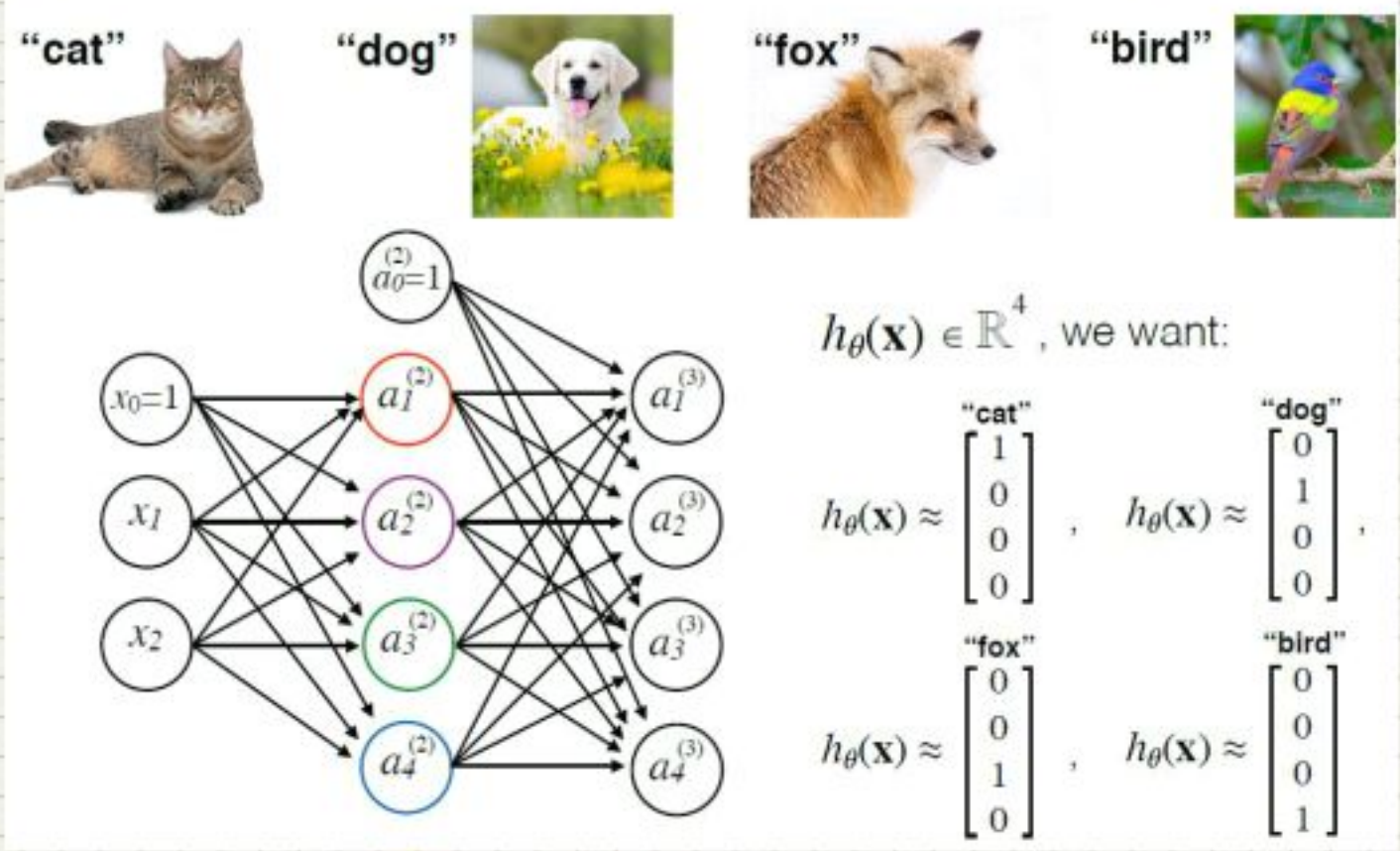
  At prediction time we select the classifier with the highest confidence score.

Example 3: 4 classes:
- cat
- dog
- fox
- bird

There are 4 units in the last layer.

The encoding we can use to discriminate these 4 classes is ONE-HOT ENCODING.

Then each output unit is going to be responsible to recognize examples belonging to one of the 4 classes.



This allows us to extend the definition of Loss function, in particular the Logistic Regression cost function, because each unit of the final layer use sigmoid unit. $h_\theta(x) \in \mathbb{R}^K$, where $(h_\theta(x))_k$ = k^{th} output

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \cdot \log(h_\theta(x^{(i)})_k) + (1 - y_k^{(i)}) \cdot \log(1 - h_\theta(x^{(i)})_k)\right] +$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}(\theta_{ji}^{(l)})^2$$

L : number of layers in the network

u_l : number of units in layer l

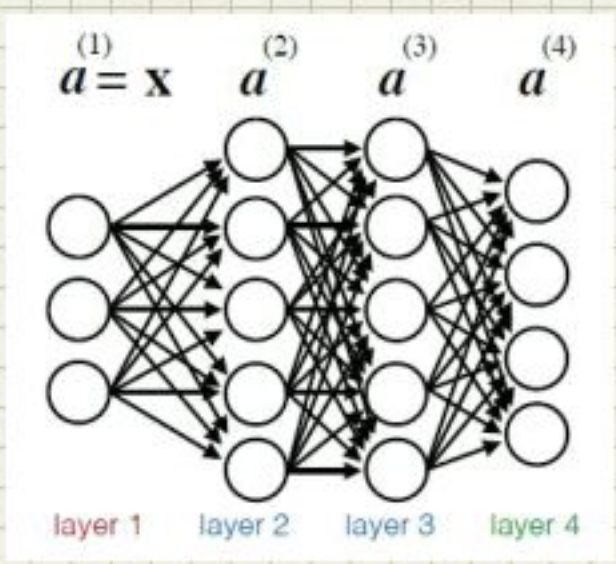
We can learn parameters with **Gradient Descent**.

We solve the optimization problem, trying to find the right choice of parameter θ, in order to minimize our cost function:

$$\min_{\theta} J(\theta)$$

In order to do that, we have to compute the partial derivative $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta)$ with respect to the v. θ related to layer l

$\theta_{ij}^{(l)} \in \mathbb{R}$ connects unit j in layer l to unit i in layer $j+1$



Example 4: 4-layer Neural network with 2 hidden layers.

Each unit of layer 1 is represented by $a^{(1)} = x$.
The same for the units in the other layers.

We use the forward propagation procedure: the unit in the next layer are obtained by propagating the values and applying the linear combination of inputs and then applying the non-linear f

Starting from the first layer through all the layers to the output layer.

At every iteration we add the BIAS UNIT $a_i^{(j)} = 1$

1) $a^1(x) = x$ 2) $z^{(2)} = \theta^{(1)} a^{(1)}$ $a^{(2)} = f(z^{(2)})$ add bias $a_0^{(2)} = 1$
3) $z^{(3)} = \theta^{(2)} a^{(2)}$ $a^{(3)} = f(z^{(3)})$
4) $z^{(4)} = \theta^{(3)} \cdot a^{(3)}$ $a^{(4)} = f(z^{(4)}) = h_\theta(x)$

3) **Backpropagation** apply the same procedure seen in forward propagation in order to approximate and to spread the error we can do by computing the activation in all different units.

By computing this Backpropagation step, we want to spread the weights responsible for the errors. $\delta^{(2)} \leftarrow \delta^{(3)} \leftarrow \delta^{(4)}$

1) Compute the errors we are doing in all the units $\delta_j^{(l)}$ (error of unit j in layer l)
From the output layer (last layer) we compute the error for all units, obtained computing the difference between the activation for each node and the corresponding original value y_j: $\delta_j^{(4)} = a_j^{(4)} - y_j$
2) Then we compute the error terms for the previous layers:
$\delta^{(3)} = (\theta^{(3)})^T \cdot \delta^{(4)}$ is obtained by computing the product between the parameters and the error coming from the next layer.
The error $\delta^{(3)}$ is processed by computing the corresponding **derivative** on the function f with respect to the linear combination of input nodes $z^{(3)}$
$\delta^{(3)} = (\theta^{(3)})^T \cdot \delta^{(4)} .* f'(z)$, $\delta^{(2)} = (\theta^{(2)})^T \delta^{(3)} .* f'(z)$.* element-wise multipl.

The derivative can be obtained by computing $g'(z^{(3)}) = a^{(3)} .* (1 - a^{(3)})$

Back propagation algorithm (BP)

1 Training examples $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$ and initialize all $\Theta^{(l)}$ randomly ($\neq 0$)

---

2 We set $\Delta_{ij}^{(l)} = 0$, that are accumulators used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

3 for $k=1$ to m // iterate over all training samples

4 set $a^{(1)} = x^{(k)}$ // set the input layer to $a^{(1)}$ unit

5 for $l=2$ to L compute $a^{(l)}$ // from layer 2 to layer L, we compute $a^{(l)} \rightarrow$ FP

6 compute $\delta^{(L)} = a^{(L)} - y^{(k)}$ // compute the error of output unit

7 compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$ // from layer $L-1$ compute all the errors in the opposite direction for all the intermediate layers $\rightarrow$ BP

8 compute gradients $\boxed{\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}}$ $\xrightarrow{\text{vectorization}}$ $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

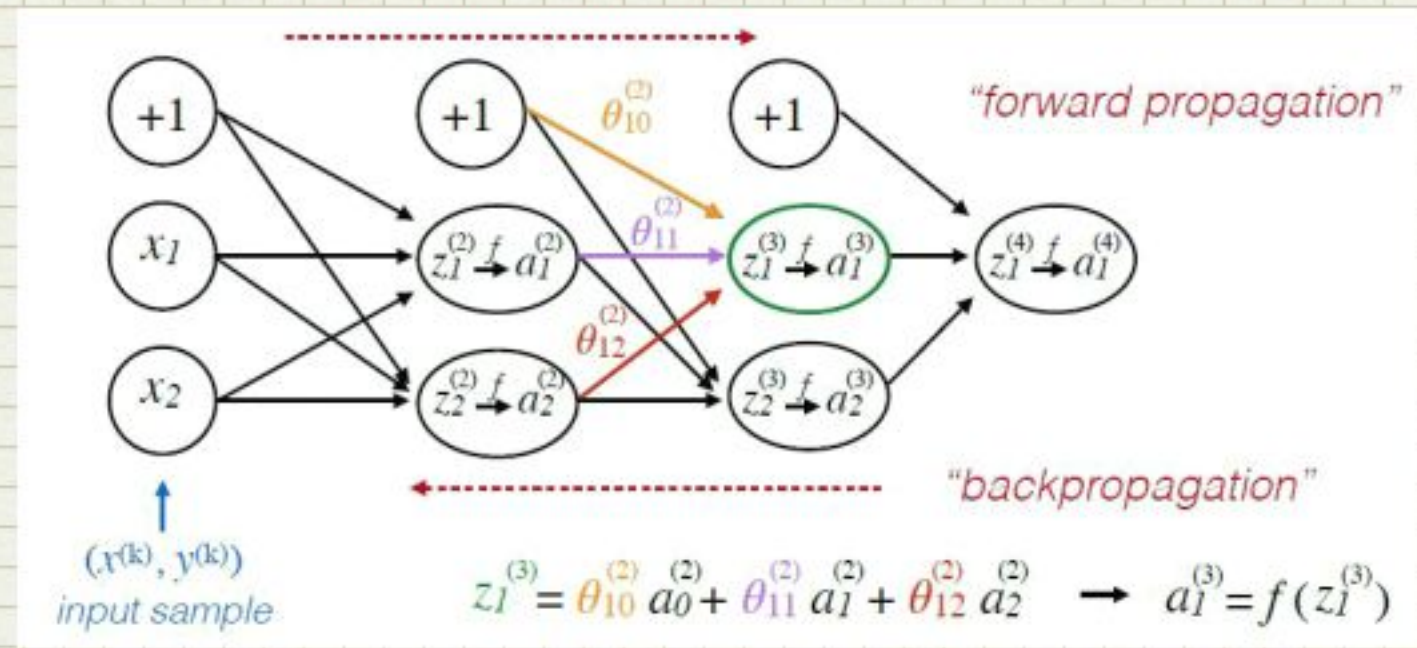
9 compute $D_{ij}^{(l)} = \begin{cases} \frac{1}{m} (\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}) & \text{if } j \neq 0 \\ \frac{1}{m} \Delta_{ij}^{(l)} & \text{if } j = 0 \end{cases}$

---

10 Update weights : $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \eta D_{ij}^{(l)}$

In order to compute BP, we compute $\delta_j^{(l)}$ error of cost for $a_j^{(l)}$ in layer l.

Example: $\delta^4 = a^{(4)} - y$



$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(k)$ (for $j \geq 0$)

$\text{cost}(k) = y^{(k)} \log h_\Theta(x^{(k)}) + (1 - y^{(k)}) \log (1 - h_\Theta(x^{(k)}))$

CHAPTER 3

3.1 DECISION TREES

A DECISION TREE is a structure in which internal nodes represent attributes (features) and leaf nodes represent class labels (CLASSIFICATION) or target values (REGRESSION), that can be real values.

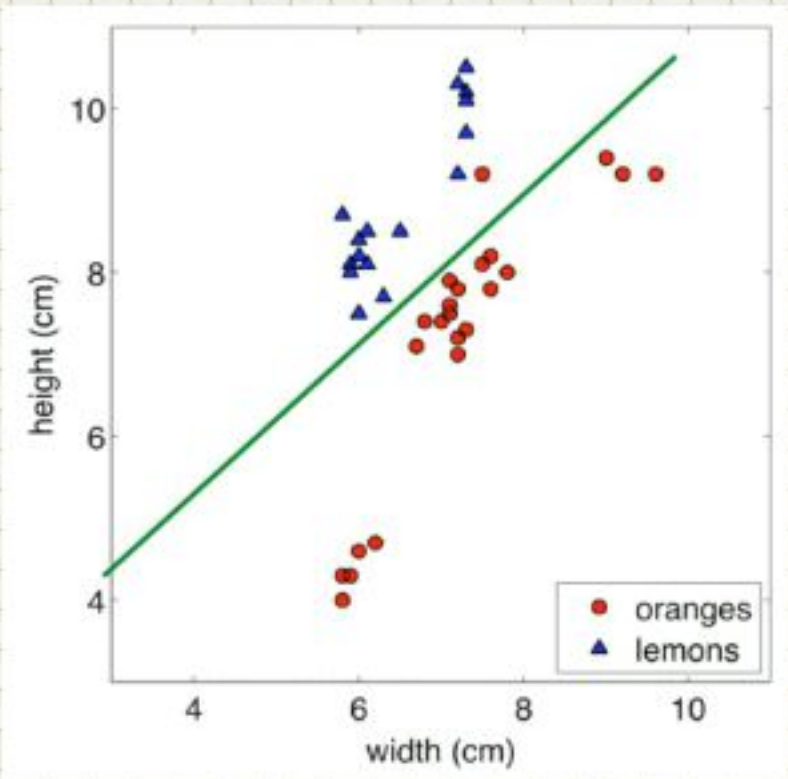
Decision trees are widely used in operations research to support decision analysis, but they are also used in AI/ML.

They are used because they are able to provide interpretable results.

Example: in medical domain it's not enough to have an effective solution based on NN or other classifiers, but there are some interpretable results, that can goranteed justify the final classification result the tree algorithm is providing.

A decision tree represents the hypothesis function $h(x)$ we want to approximate through machine learning algorithm.

We start with the intuition of decision tree:
We take a look at our classification task, where we want to solve this binary classification problem to discriminate between oranges and lemons.
In order to represent our example, we have 2 measurements, that correspond to our features.
A solution applied to solve this problem is a simple linear classifier.
We can construct a simple LINEAR DECISION BOUNDARY.



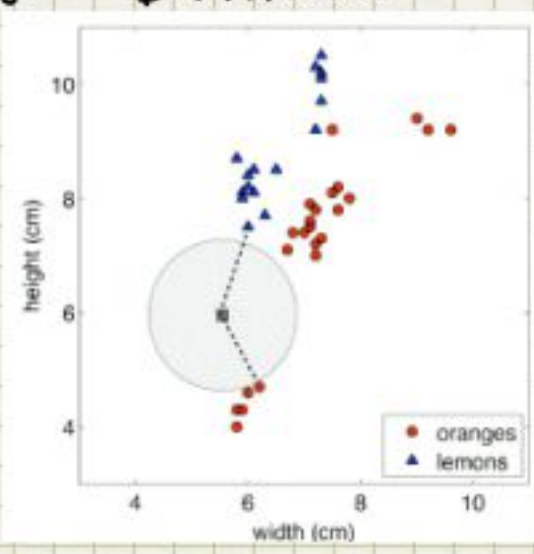
We can also use LOGISTIC REGRESSION or the NEURAL NETWORKS.

$$y = \text{sign}(h_\theta(x)) \quad h_\theta(x) = \theta^T x$$

All these approaches have something in common, computing by learning parameters θ an hypothesis function, that can approximate our ideal function f.

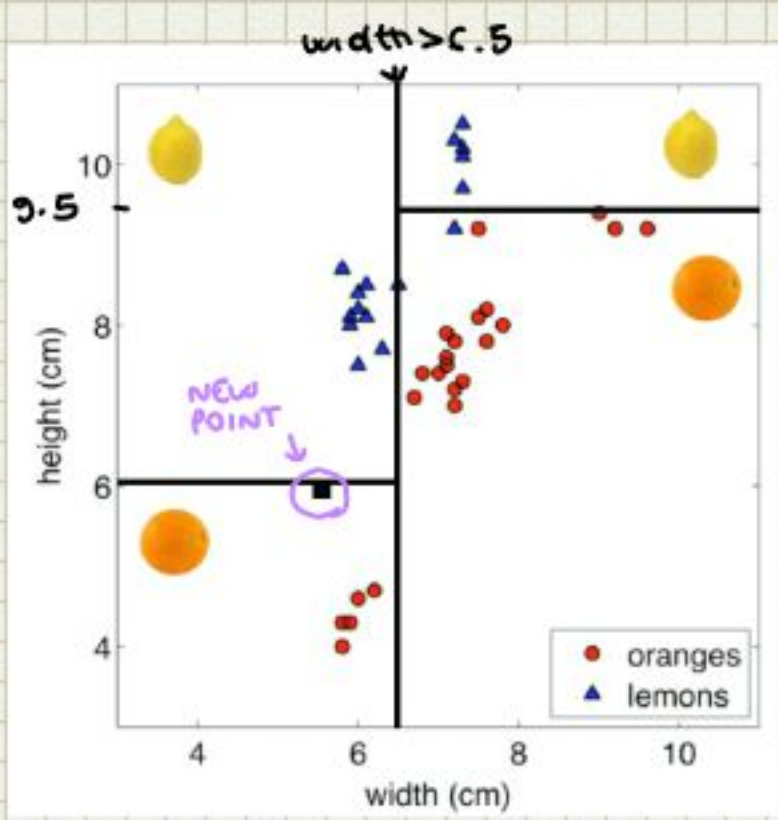
Another possible approach is NON-PARAMETRIC, in particular we focus on K-Nearest Neighbors.
The idea is to assign to new sample the class of the closest point in the feature space.
In this case the decision boundary between the 2 classes are provided by the istances in the training set.

DISTANCE



The general idea of DT is summarize by this different construction. We have to build the DT by following a specific procedure:

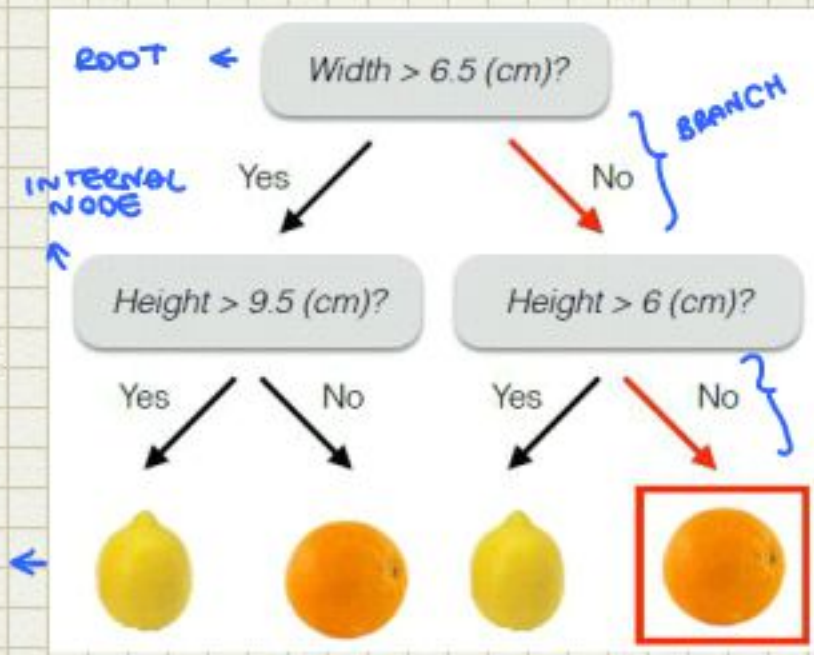
1) Pick on attribute (feature) and use it in order to design a test on data
2) Conditioned on a choice, pick another attribute. We iterate this process recursively
3) In the leaves (outcome of DT), we assign a class with majority vote
4) The some process is applied to all other branches.



Example1: classify samples belonging to oranges or lemons.

We test if the samples are bigger or smaller than a particular threshold.

- We ask if the sample is greater than 6.5 cm It's a binary test, so the answer is yes or no. In other words, we are saying that this sample is on left or in the right of this first boundary.
- The new test (in case of yes) consists in looking if the height of the sample is greater or not of 9.5 cm.
- This test can be represented in feature space by drawing a line.
By answering to a question, you have a decision path, where you can discriminate between one or the other class.
The same can be applied for the other question

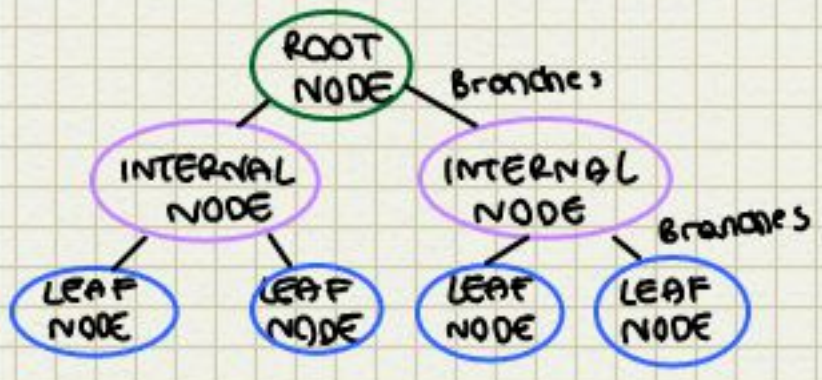


If we have a new point, we have to iterate, apply the decision tree to the particular istance you want to classify.

In that case, by following the path from the root to the leaf node and answering to the tests, we can get that the new sample is an Orange (answering no to both tests).

In DECISION TREE we can observe 3 ingredients:

1) INTERNAL NODES related to test attributes
2) BRANCHING is determined by attribute value
3) LEAF NODES are the outputs/class assignments.



Note: DT can also be seen as generative models of induction rules from empirical data.

Example 2: Build a decision tree to decide whether to wait for a table at a restaurant. This can be represented as a boolean function and our goal is to learn a definition for the goal predicate WillWait.
A boolean DT is equivalent to the assertion that the goal attribute is true if and only if the input attributes satisfy one of the paths leading to a leaf with value true.

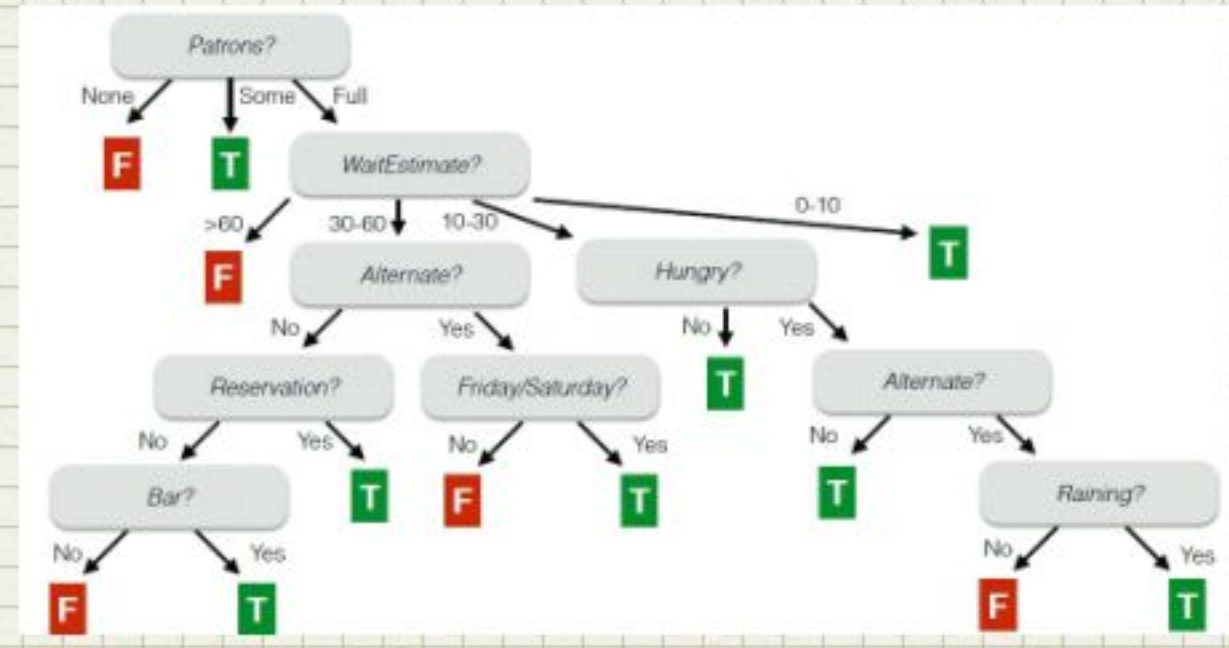
We have 10 attributes we consider as inputs. The rows are our samples.
n=12 training examples.

Each attribute corresponds to some event.

The target (Yes/No) is the answer to wait or not function.

Example	Input Attributes										Target (*WillWait*)
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	
$x^{(1)}$	Yes	No	No	Yes	Some	$$$	No	Yes	French	0–10	$y^{(1)}$
$x^{(2)}$	Yes	No	No	Yes	Full	$	No	No	Thai	30–60	$y^{(2)}$
$x^{(3)}$	No	Yes	No	No	Some	$	No	No	Burger	0–10	$y^{(3)}$
$x^{(4)}$	Yes	No	Yes	Yes	Full	$	Yes	No	Thai	10–30	$y^{(4)}$
$x^{(5)}$	Yes	No	Yes	No	Full	$$$	No	Yes	French	>60	$y^{(5)}$
$x^{(6)}$	No	Yes	No	Yes	Some	$$	Yes	Yes	Italian	0–10	$y^{(6)}$
$x^{(7)}$	No	Yes	No	No	None	$	Yes	No	Burger	0–10	$y^{(7)}$
$x^{(8)}$	No	No	No	Yes	Some	$$	Yes	Yes	Thai	0–10	$y^{(8)}$
$x^{(9)}$	No	Yes	Yes	No	Full	$	Yes	No	Burger	>60	$y^{(9)}$
$x^{(10)}$	Yes	Yes	Yes	Yes	Full	$$$	No	Yes	Italian	10–30	$y^{(10)}$
$x^{(11)}$	No	No	No	No	None	$	No	No	Thai	0–10	$y^{(11)}$
$x^{(12)}$	Yes	Yes	Yes	Yes	Full	$	No	No	Burger	30–60	$y^{(12)}$

PROBLEM of DT: you won't generalize to new examples.
We need some kind of generalization to ensure more compact DT.



3.1.1 ALGORITHM

A tree is built by splitting the training dataset into SUBSETS which constitute the successor children.

1) This decision is done by choosing an attribute on which to descent at each level
2) Then you condition on earlier (higher) scores
3) You restrict only one dimension at a time
4) You declare an output when you get to the bottom. It means that a subset at a node has all the same value of the target y.

This procedure is repeated on each derived subset in a recursive manner (RECURSIVE PARTITIONING).

Each path from the root to a leaf defines a region R_n of the input space.

Given the training set $\{(x^{(n1)}, t^{(n1)}), \dots, (x^{(nk)}, t^{(nk)})\}$, that fall into region R_n following the path.

CLASSIFICATION TREE

- Discrete outputs (class labels)
- leaf $y^{(k)}$ is set to the most common value in $\{(t^{(n1)}, \dots, t^{(nk)})\}$

REGRESSION TREE

- continuous output
- leaf $y^{(k)}$ is set to the mean value in $\{(t^{(n1)}, \dots, t^{(nk)})\}$

3.1.2 Learning Decision trees

Decision tree is an old algorithm, model. The main goal is construct or design the model in an effective way, in particular learn the simplest (smallest) decision tree in general starting from data, that is a NP complete problem.

The idea is to resort to a greedy heuristic (greedy = avida/golosa, heuristic = euristico) we start from an empty decision tree, then we split the starting set of examples by picking the next best attribute and then we apply this recursive partitioning approach.

The task is to decide what the best attribute is, using the information theory.

In general a decision tree represents a boolean function.

- Each path from root to leaf represents a conjuction of test on attributes.
- Different paths leading to the same classification represent a disjunction of conjunctions.

These 2 rules define a series of DNF (Disjunctive Normal Form), one from each class.

ID3 is one of the most popular algorithms for learning decision trees.

(S: training examples, A: attributes)

1) It starts by creating a root node r for the tree.
2) If all the examples in S belong to the same class y, it returns the root node r with label y.
3) If A is empty, it returns r labeled as the majority class in S.
   Otherwise, we select the optimal attribute $a \in A$, where A is the set of attributes.

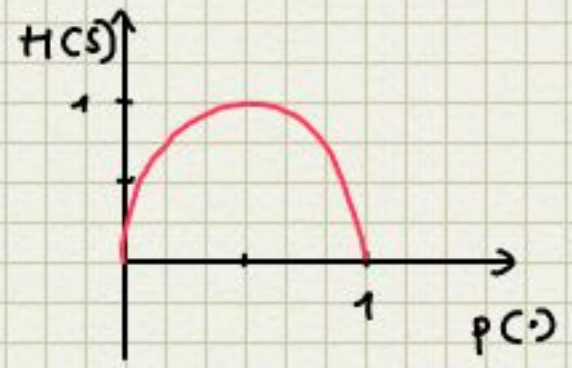
   Each iteration will pick one particular attribute and the goal is to select in a convenient way this sequence of attributes A in order to construct the tree.
4) After selecting the optimal attribute we can partition/split the set S into subsets for which the resulting entropy is minimized (or information gain is maximized).

   Then we make a decision tree node containing that attribute a.
5) After we can recurse on subsets using the remaining attributes.

The criteria used to select the optimal attributes are:

1) Entropy (H) is a measure of uncertainty (/randomness) in the dataset S.

   It can be used to measure which one of 2 sequences, S_1 and S_2, has more uncertainty or randomness.



$$H(S) = -\sum_{c \in C} p(c) \log_2 p(c)$$

C = set of classes in S

p(c) = proportion of elements in c to the elements in S.

- High entropy means that variable has a uniform like distribution the histogram is flat. Then values sampled from it are less predictable (having the same prob.)
- Low Entropy means that the distribution of variable has many peaks and valleys and histogram has many lows and highs. Then values sampled from it are more predictable

2) Information Gain measures the difference between entropy BEFORE split and the average entropy after the split of the dataset based on given attribute values.

Then IG measures how much uncertainty in S was reduced after splitting set S on attribute a

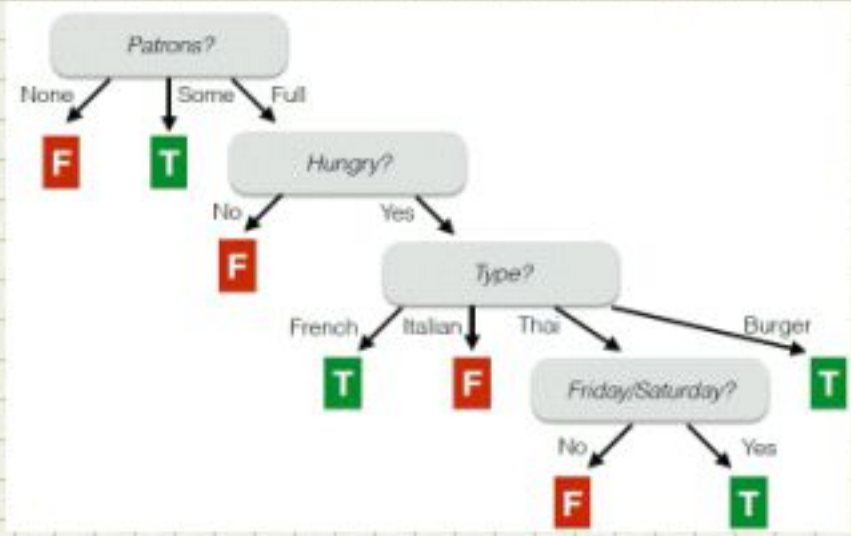
$$IG(S,a) = H(S) - \sum_{t \in T} p(t)\, H(t) = H(S) - H(S|a)$$

- $H(S)$: entropy of dataset S
- $p(t)$: proportion of elements in t to the elements in S
- $H(t)$: entropy of subset t

In ID3, infogain is calculated for each remaining attribute. The attribute with the largest IG is used to split the set on this iteration

The goals to make a good tree are:

- Not too small → need to handle distinctions in data
- Not too good → to reach computational efficiency and to avoid training examples (avoid overfitting problems)



- Occam's Razor → Find the simplest hypothesis (the smallest tree that fits the observations)
- Inductive Bias → small trees that place high information gain attributes close to the root are preferred

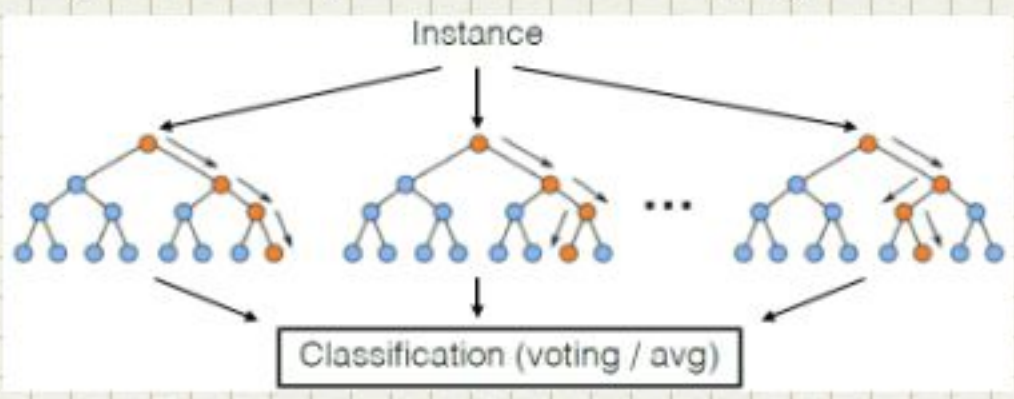
There are some problems:

1) There are exponentially less data at lower levels
2) A very big tree can overfit the data
3) Greedy algorithms don't usually yield the global optimum, but local
4) IG privileges attributes that assume a broad range of values.

Often you need to regularize the construction process to get small but highly informative trees.

3.1.3 Random Forests

RF is a classification algorithm consisting of many DTs. It uses bagging and feature randomness to combine trees. This way it reduces the variance of classifier (DT) and then making an ensemble out of it.



3.2 SUPPORT VECTOR MACHINE

SVM is a very popular and powerful algorithm:

- It's a SUPERVISED / PARAMETRIC algorithm usually used for classification, but can be also used for regression.
- It's a NON-PROBABILISTIC (binary) classifier, although there are methods such as Platt scaling to give a probabilistic interpretation of SVM output
- It's a LINEAR CLASSIFICATION model, but SVM can efficiently perform a non-linear classification using the kernel trick

In SVM the definition of cost function can be seen as approximation of the Logistic cost function

$$\text{cost}(\cdot) = -y^{(i)} \log\left(\frac{1}{1+e^{-\theta x^{(i)}}}\right) - (1-y^{(i)}) \cdot \log\left(1 - \frac{1}{1+e^{-\theta^T x^{(i)}}}\right)$$

→ Regularized Logistic Regression

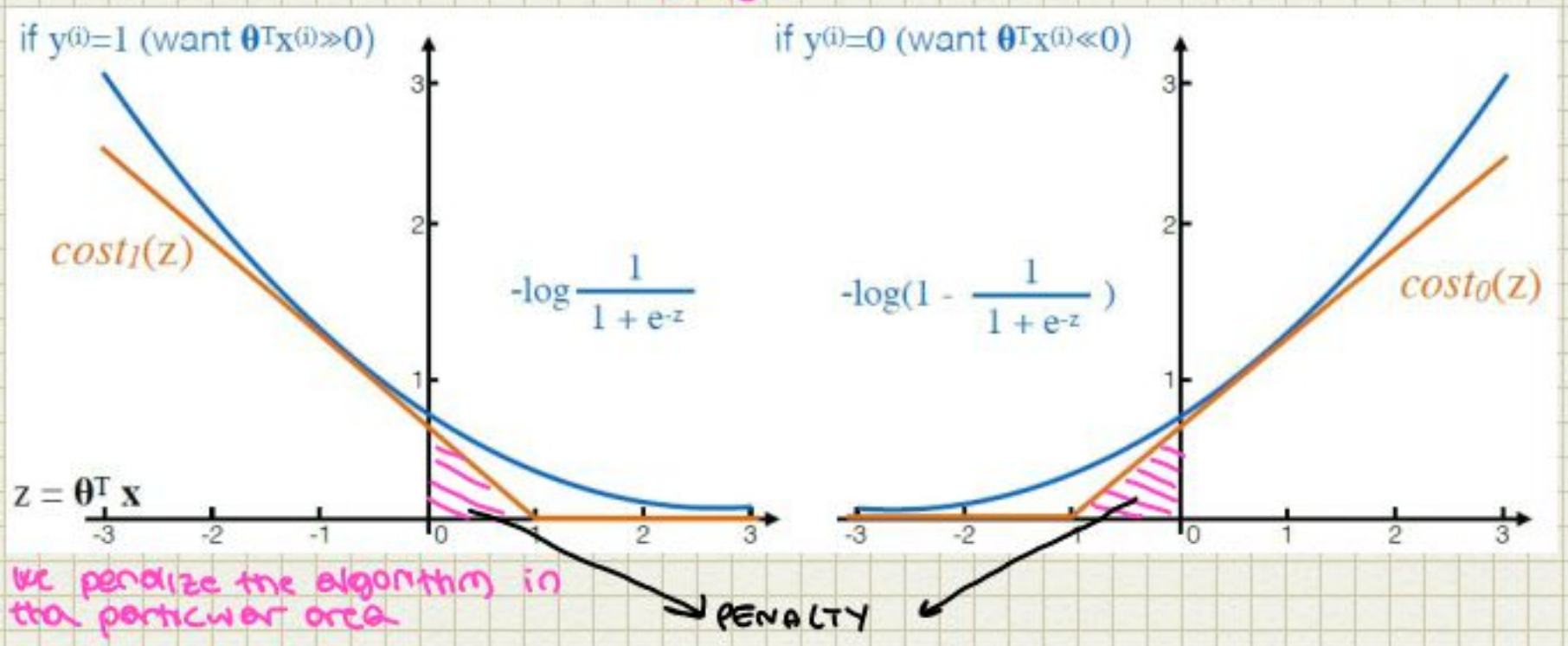
$$\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} -y^{(i)} \cdot \log(h_\theta(x^{(i)})) - (1-y^{(i)}) \cdot \log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2$$

→ support vector machine

$$\min_\theta \frac{1}{\cancel{m}}\underbrace{\left[\sum_{i=1}^{m} y^{(i)}\, \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)}) \cdot \text{cost}_0(\theta^T x^{(i)})\right]}_{A} + \underbrace{\frac{\lambda}{2\cancel{m}}\sum_{j=1}^{m}\theta_j^2}_{\lambda B}$$

$CA + B$, $C = \frac{1}{\lambda}$

$$\Rightarrow \min_\theta C \underbrace{\sum_{i=1}^{m}\left[y^{(i)} \cdot \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)}) \cdot \text{cost}_0(\theta^T x^{(i)})\right]}_{\approx 0} + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$



We penalize the algorithm in that particular area

We are forcing the algorithm to pick the right configuration parameters, in which we are optimizing θ in order to have:

- $\theta^T x^{(i)} \geq 1$, if $y^{(i)} = 1$
- $\theta^T x^{(i)} \leq -1$, if $y^{(i)} = 0$

$$J(\theta) = \sum_{i=1}^{m} \text{cost}(h_\theta(x^{(i)}), y^{(i)}), \quad \text{cost}(h_\theta(x^{(i)}), y^{(i)}) = \begin{cases} \max(0, 1-\theta^T x^{(i)}) & y^{(i)}=1 \\ \max(0, 1+\theta^T x^{(i)}) & y^{(i)}=0 \end{cases}$$

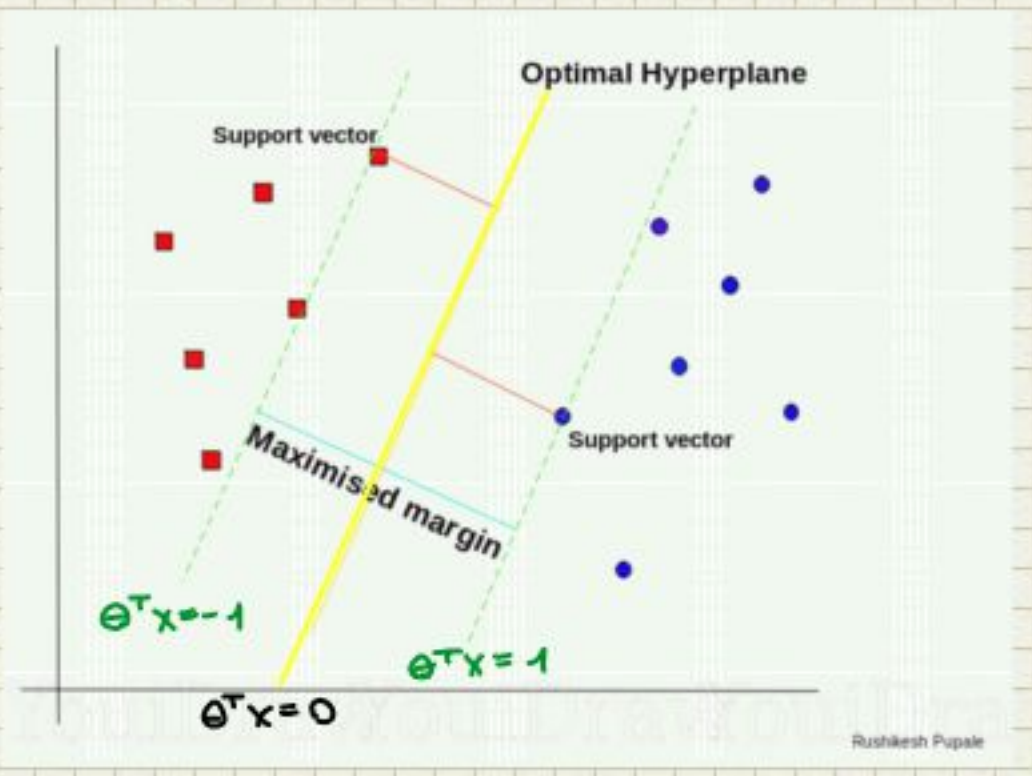
$$\min_\theta \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 \quad \text{s.t.} \quad \theta^T x \geq 1 \quad \text{if } y^{(i)}=1$$

$$\theta^T x \leq -1 \quad \text{if } y^{(i)}=0$$

SVM will pick a decision boundary in such a way that the separation between the 2 classes is as wide as possible.

The hyperplanes are obtained by following the constraints designed in the loss function.

These decision boundaries are boundaries obtained by optimizing the loss function in order to get a hyperplane s.t. we have a larger min distance from any of training examples.



Idea: instead of fitting all the points, we focus on boundary points