Introduction to machine learning

by Quentin de Laroussilhe - http://underflow.fr - @Underflow404

Machine learning

A machine learning algorithm is an algorithm learning to accomplish a task by observing data.

- Used on complex tasks where it’s hard to develop algorithms with handcrafted-rules
- Exploits patterns in observed data and extract rules automatically

Fields of application

- Computer vision
- Speech recognition
- Financial analysis
- Search engines
- Ads-targeting
- Content suggestion
- Self-driving cars
- Assistants
- etc...

Example : object detection

Big variation in **visual features :**

- Shape
- Background
- Size / position

Classifying an object in a picture is not an easy task.

Example : object detection

- Learn from annotated corpus of examples (a dataset) to **classify** unknown images among different object types
- Observe images to **learn patterns**
- Lot of data available (i.e: ImageNet dataset)
- Very good error rates (< 5% with deep-CNN)

General concepts

Types of ML algorithms

Supervised

Learn a function by observing examples containing the input and the expected output.

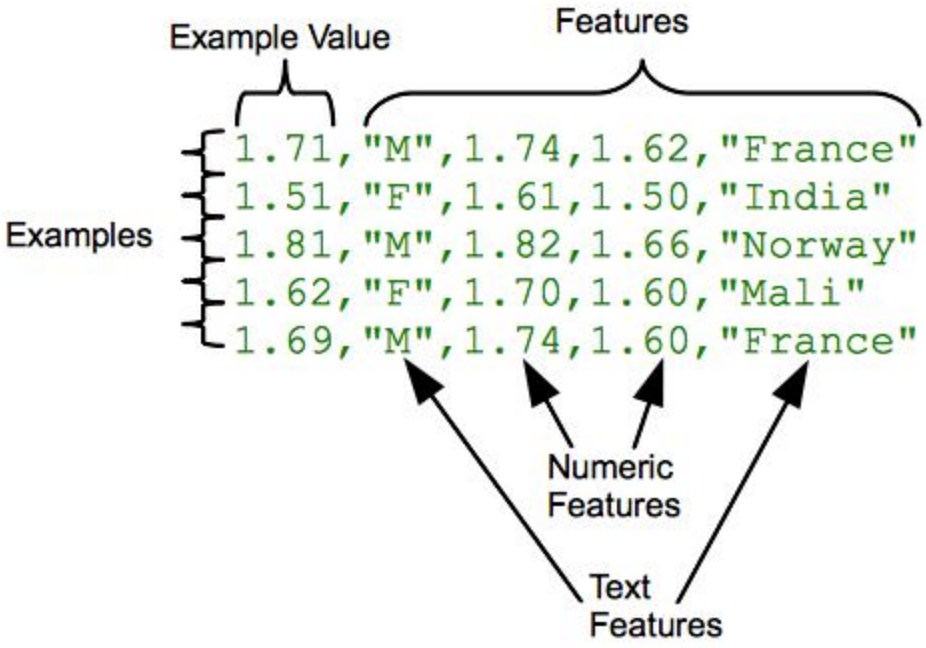
- Classification
- Regression

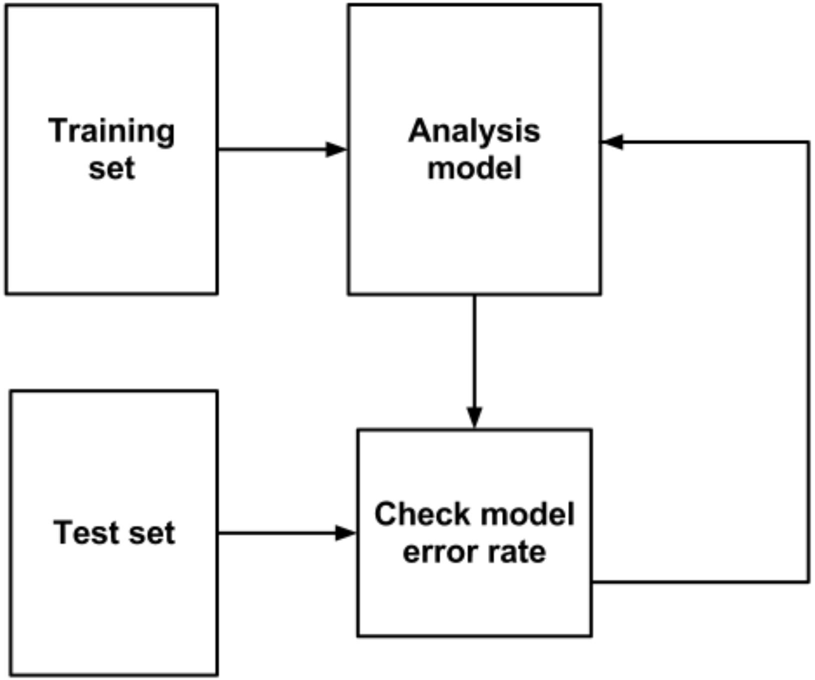
Unsupervised

Find underlining relations in data by observing the raw data only (without the expected output).

- Clustering
- Dimensionality reduction

Training set





Classification vs Regression

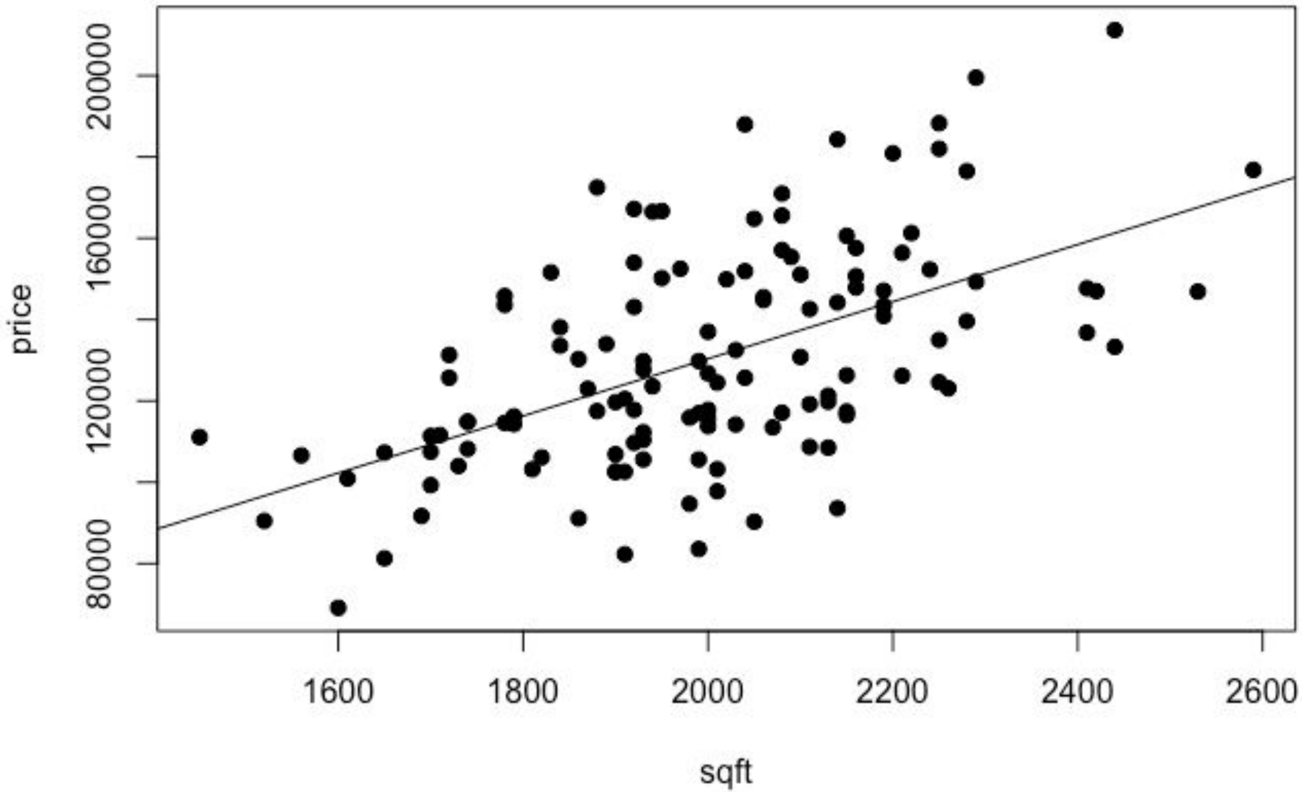
Regression
Learn a function mapping an input element to a real value.

i.e: *Predict the temperature of tomorrow given some meteo signals*

Classification
Learn a function mapping an input element to a class (within a finite set of possible classes).

i.e: Predict the weather of tomorrow: {sunny, cloudy, rainy} given some meteo signals

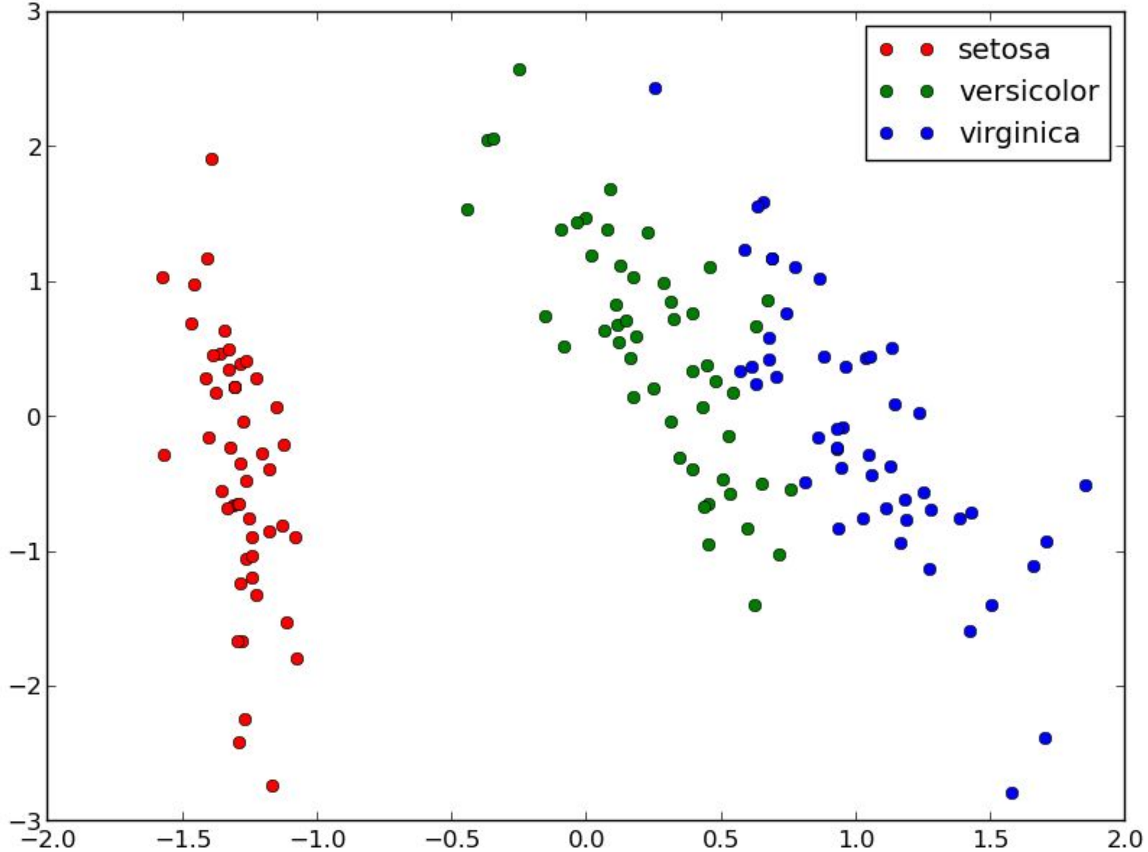
Regression



price
200000
160000
120000
80000
1600
1800
2000
2200
2400
2600
sqft

Classification

Illustration: scikit-learn documentation



Clustering

A clustering algorithm separate different observed data points in similar groups (clusters). We do not know the labels during training.

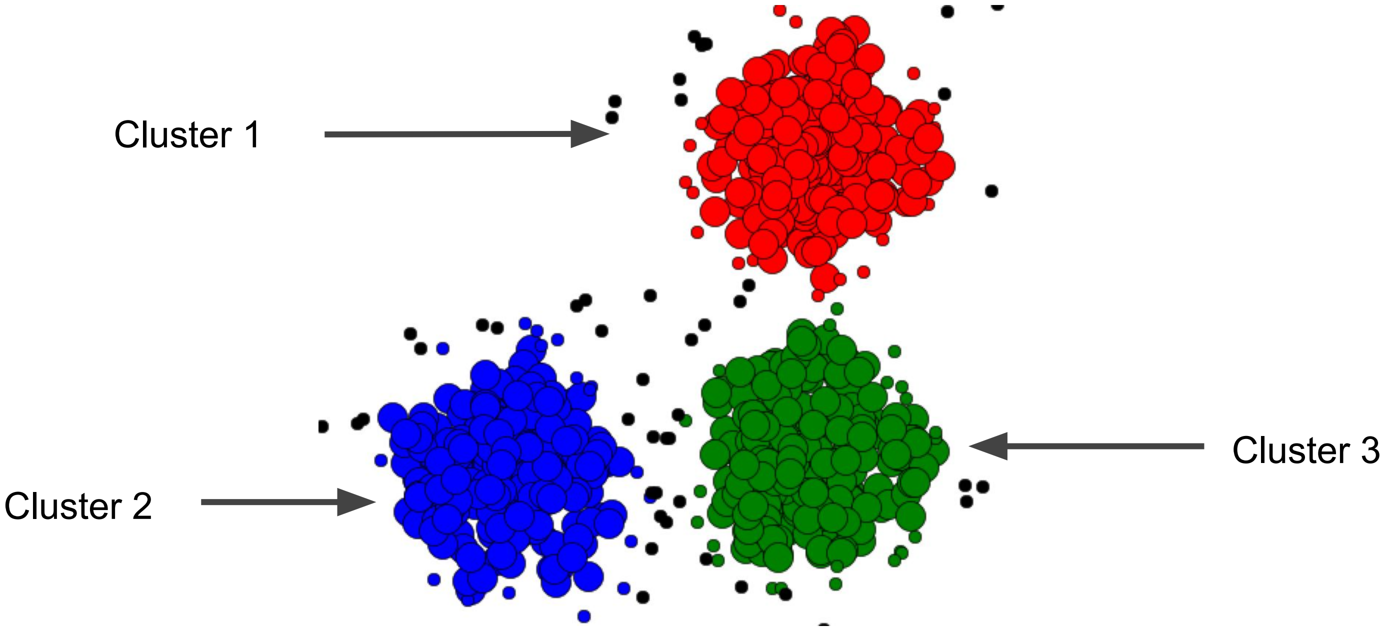


Illustration: Rana Forsati

Reinforcement learning

Learn the optimal behavior for an agent in an environment to maximize a given goal.

Examples:

- Drive a car on a road and minimize the collision risk
- Play video-games
- Choose the position of ads on a website to maximize the number of clicks

Feature extraction

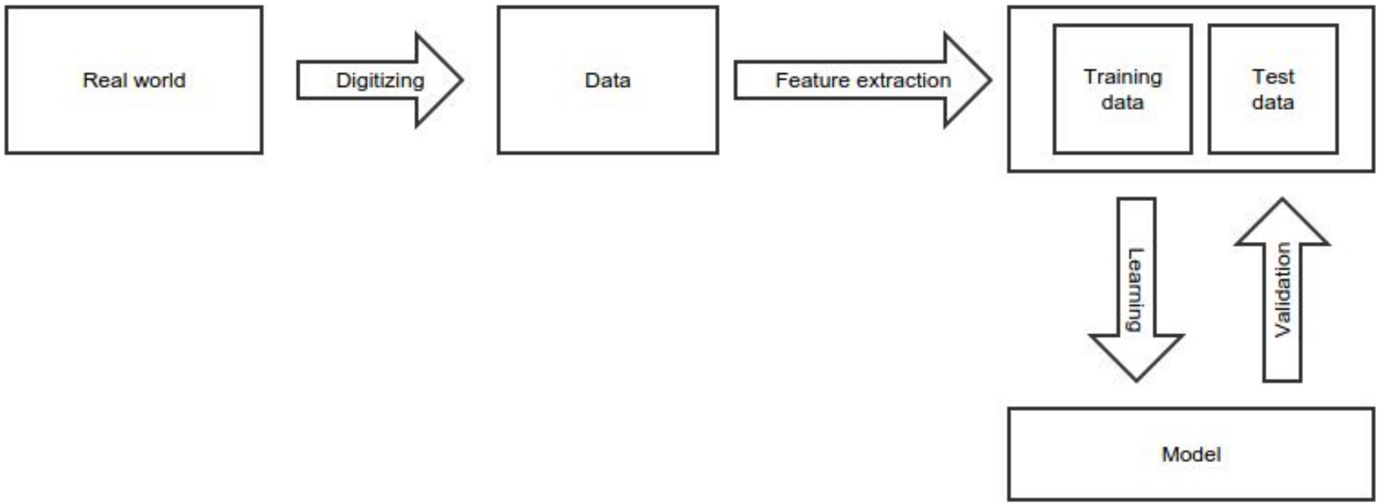
The first step in a machine learning process is to extract useful values from the data (called features).

The goal is to extract the information useful for the task we want to learn.

Examples:

- Stock market time-serie → [opening price, closing price, lowest, highest]
- Image → Image with edges filtered
- Document → bag-of-word

Modelisation process

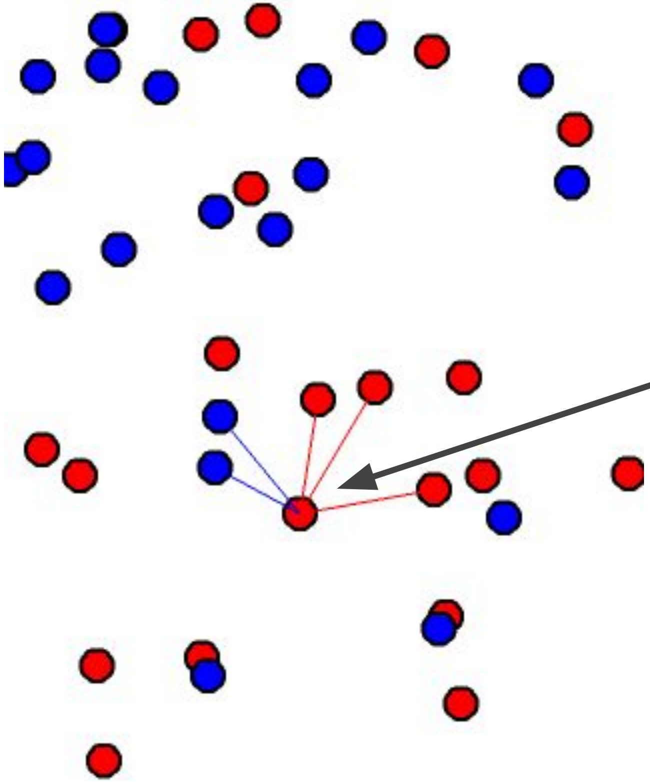


k nearest neighbors

k-nearest neighbors

- Classification and regression model
- Supervised learning: we have annotated examples
- We classify a new example based on the labels of his “nearest neighbors”
- k is the number of neighbors taken in consideration

k-nearest neighbors



To classify a point:

We look the k-nearest neighbors (here k=5) and we do a **majority vote.**

This point has 3 red neighbors and 2 blue neighbors, it will be classified as red.

k-nearest neighbors

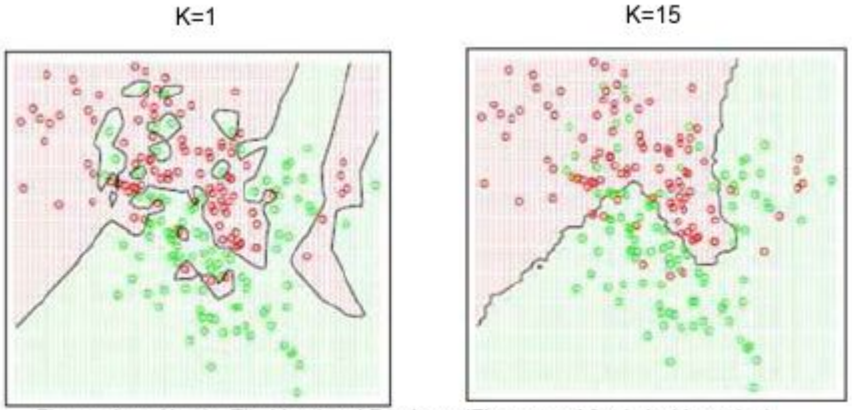
- N data points
- Require a **distance function** between points $d(x_1, x_2)$
- Regression (average the value of the k-nearest neighbors)

$$f : X \rightarrow \frac{1}{N} \sum_{i=0}^{k-1} Y_i$$

- Classification (majority vote of the k-nearest neighbors)

k-nearest neighbors : effect of k

- k is the number of neighbors taken in consideration
- If k = 1
  - The accuracy on the training set is 100%
  - It might not generalize on new data
- If k > 1
  - The accuracy on the training set might not be 100%
  - It might generalize better on unseen data



Figures from Hastie, Tibshirani and Friedman (Elements of Statistical Learning)

k-nearest neighbors : weighted version

In the case of unbalanced repartition between classes we can give weights to the examples.

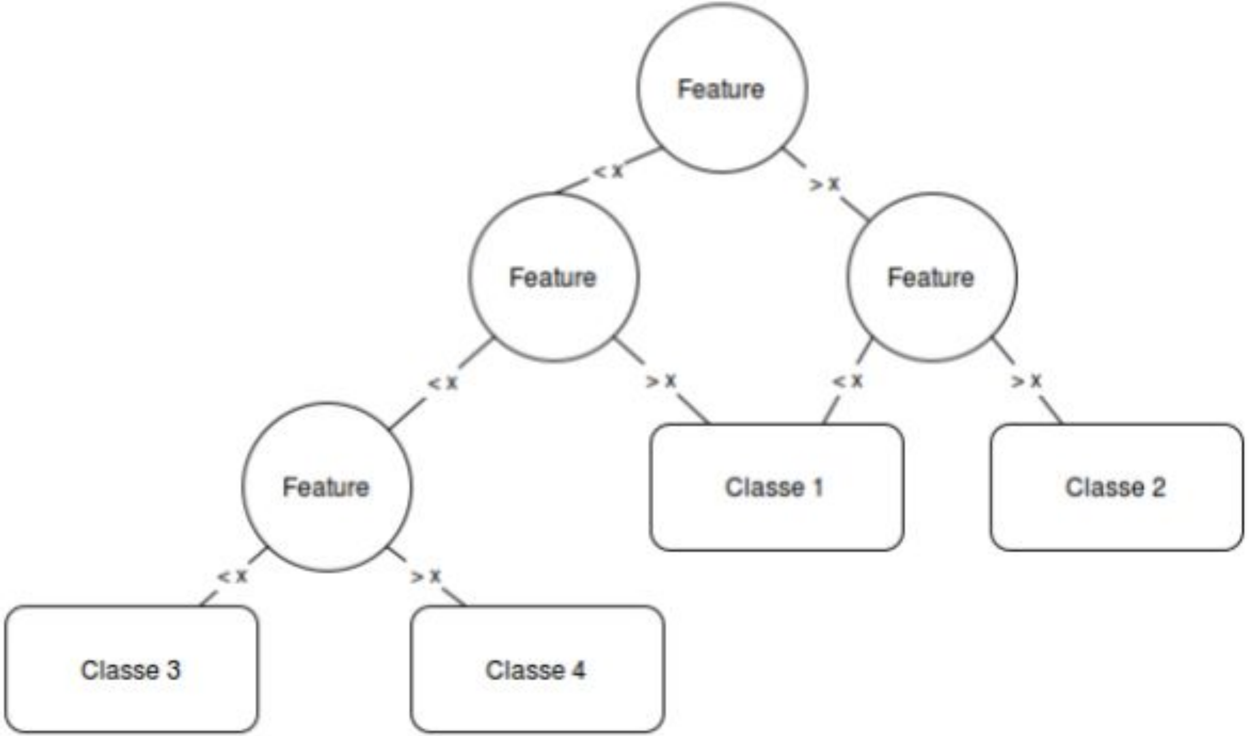
- The weight of a under represented class will be set high.
- The weight of a over represented class will be set low.

When we do the majority vote, we take the weight in consideration:

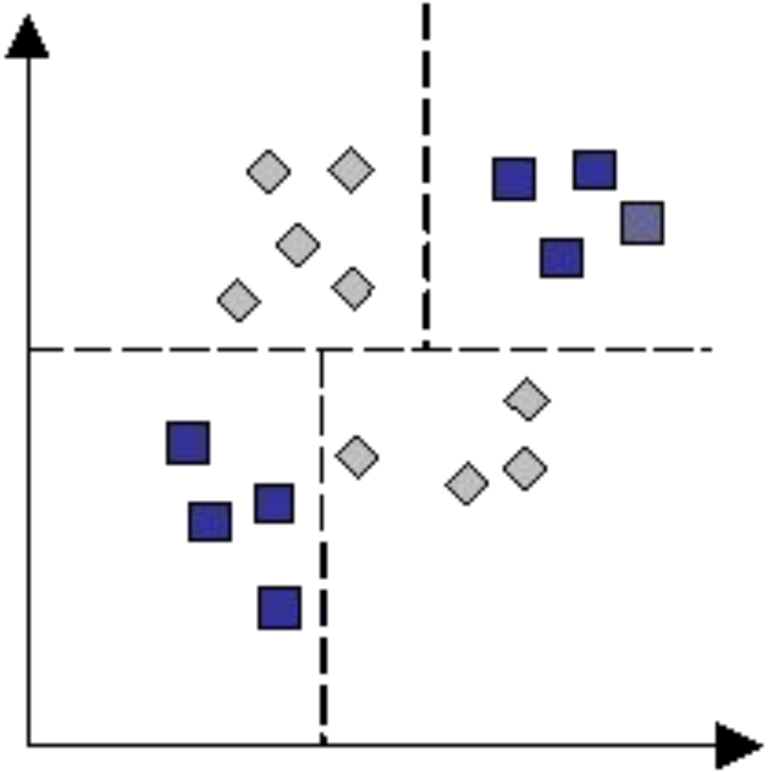
- For classification we do a weighted vote.
- For regression we do a weighted average.

Decision trees, random forests

Decision tree



Decision tree



- Decision trees partition the feature space by splitting the data
- Learning the decision tree consists in finding the order and the split criterion for each node

Decision tree

- The decision tree learning is parametrized by the method for choosing the splits and the maximum height
- If the maximum height is big enough, all the examples of the training data would be correctly classified: **overfitting**.

Decision tree : entropy metric

Entropy: $H(S) = -\sum_{x \in X} p(x) \log_2 p(x)$ $\qquad IG(A,S) = H(S) - \sum_{t \in T} p(t) H(t)$

- S: The datasets before the split
- X: Set of existing classes
- p(x): Proportion of elements in class x to the number of elements in S
- A: The split criterion
- T: The datasets created by the split

At each step we create the node by splitting with the criterion with the **highest information gain**.

Random forests

- When the depth of a decision tree is growing the error on validation data tends to increase a lot: high variance
- One way to exploit a lot of data is to train multiple decision trees and average them

Algorithm:

- Select N points in the training data and k features (usually sqrt(p))
- Learn a new decision tree
- Stop when we have enough trees

Clustering - k means

Clustering with k-means

- Clustering algorithm
- Require a **distance function** between points $d(x_1, x_2)$
- k is the number of cluster the algorithm will find

Clustering with k-means

Objective: Divide the dataset in k sets by *minimizing the within-cluster sum of squares* (sum of distances of each point of the cluster to the center of the cluster)

$$\underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2$$

Where S are the sets we are learning and $\boldsymbol{\mu}_i$ the mean of the set i.

Clustering with k-means

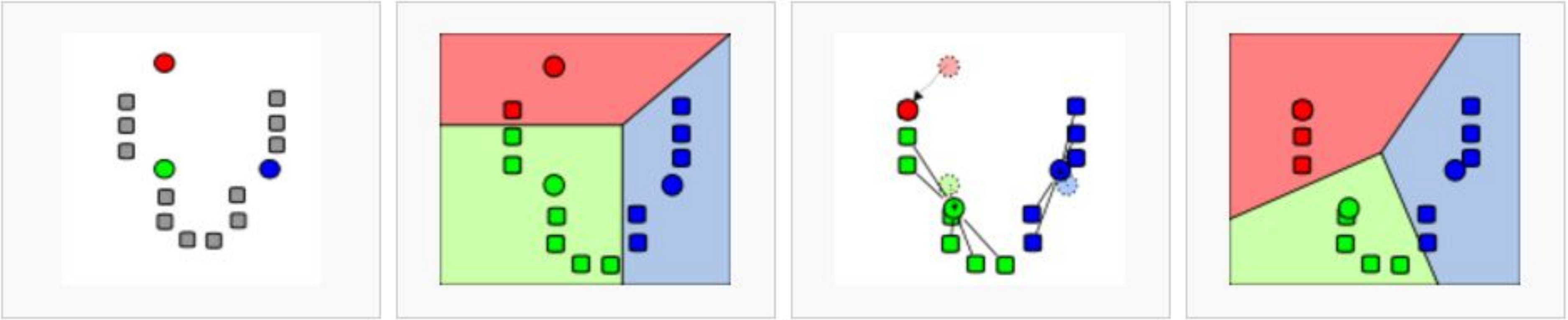


Illustration: Wikipedia, k-means

Gradient descent

Gradient descent

1. Define a model depending on W (the parameters of the model)
2. Define a loss function that quantify the error the model does on the training data
3. Compute the gradient of this loss
4. Adjust W to minimize the loss by following the direction of the computed gradient
5. Repeat until:
   - convergence
   - the model is good enough
   - your spent all your money

Linear regression

Linear regression : Introduction

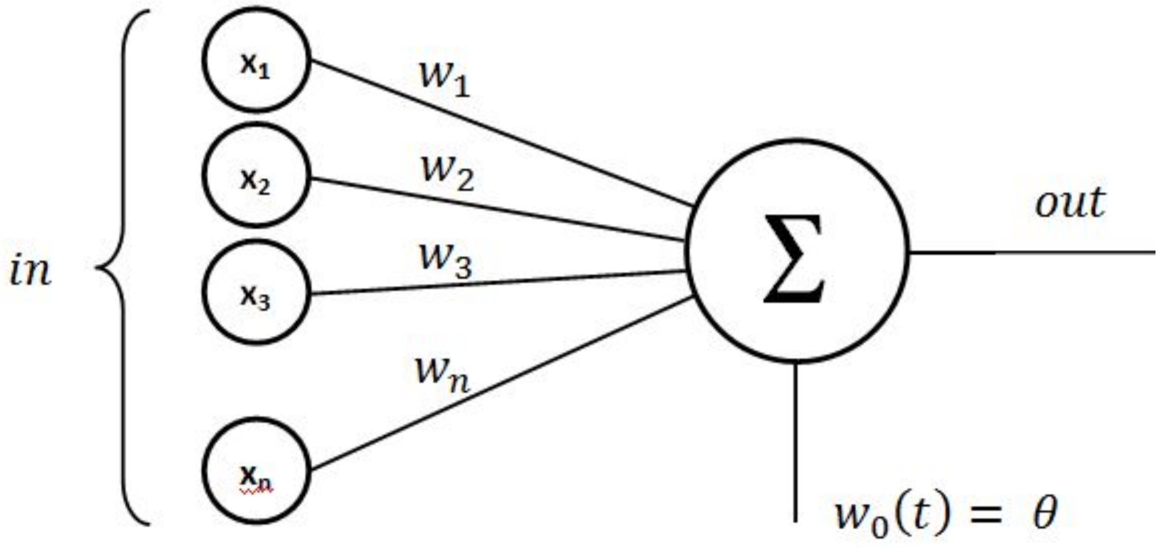
In supervised learning, we have examples of lots of input and the desired output value.

This is called the **dataset**:

- A matrix of **feature vectors** X. Each line is an vector containing an example*.
- A vector of **target values** Y. The kth component is the desired value for the kth example of X.

* : For simplification, the first element $x0 = 1$.

Linear model



Linear regression

Let say we extracted a real valued **vector of features** : X

We want to learn a **linear relation** between this features and an output value : Y

$$f : X \rightarrow Y \qquad X \in \mathbb{R}^k, Y \in \mathbb{R}$$

$$f : \sum_i W_i X_i \quad \text{(also)} \quad f : W.X^T$$

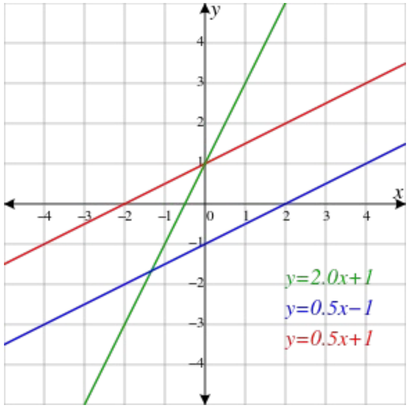
W is called weight vector, it is the **parameter of the model.**
Here we **learn the function f by adjusting W**.

Linear regression

With an input of dimension 1, the equation is:

$$f : X \rightarrow x_0.w_0 + w_1.x_1$$

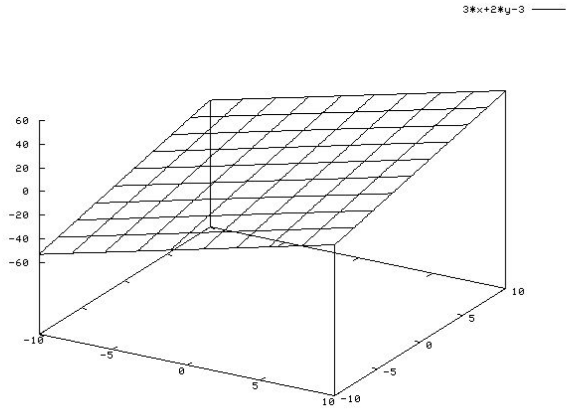
- x1 is the input value and w1 is the slope of the linear function
- We set x0 = 1 so f(0) = w0



The model can be viewed as a line, adjusted by W, mapping the input value to the output value.

It generalize in higher dimension by multiplying each coordinate of the input by its respective weight.

By adjusting W we can define any hyperplane mapping the input vector to the output value.

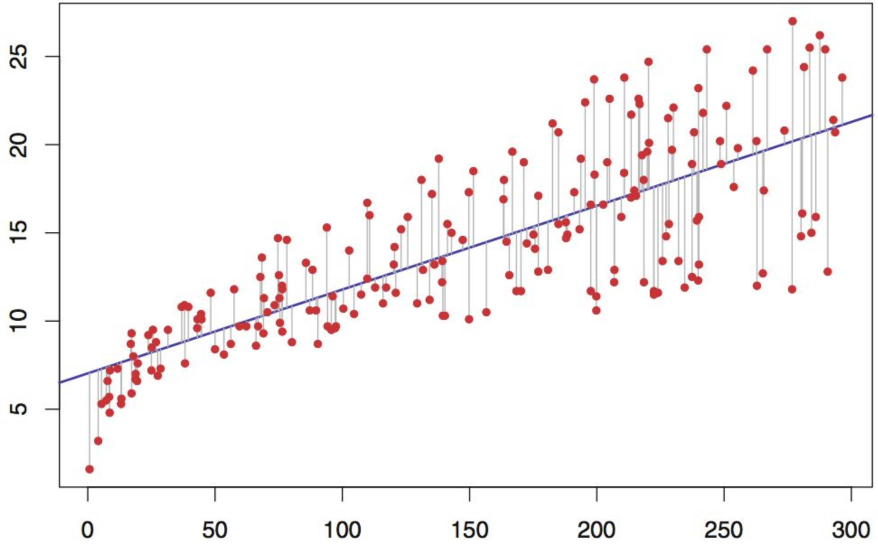


Linear regression : error function

We define a function E(W) which quantify the error we observe on the examples.

i.e: Euclidean distance between the target and f(X)

$$E(W) = \frac{1}{2}\sum_{i=0}^{n}(W.X_i^T - Y_i)^2$$



Linear regression : error function

We can define the error by using the **sum of squared distances** between expected targets and the values given by the model:

$$E(W) = \frac{1}{2}\sum_{i=0}^{n}(W.X_i^T - Y_i)^2$$

The **training error only depends of W**, the dataset is constant during the training.

Problem: Minimize the error by adjusting W (i.e: find the best W such as E is the lowest as possible).

Linear regression : gradient descent

$$E(W) = \frac{1}{2}\sum_{i=0}^{n}(W.X_i^T - Y_i)^2$$

The least squared loss of a linear model is a convex function ("bowl-shaped")

One simple way to find its minimum is by **following the slope of the error**.

$$W \leftarrow W + \alpha\frac{\delta E}{\delta W}$$

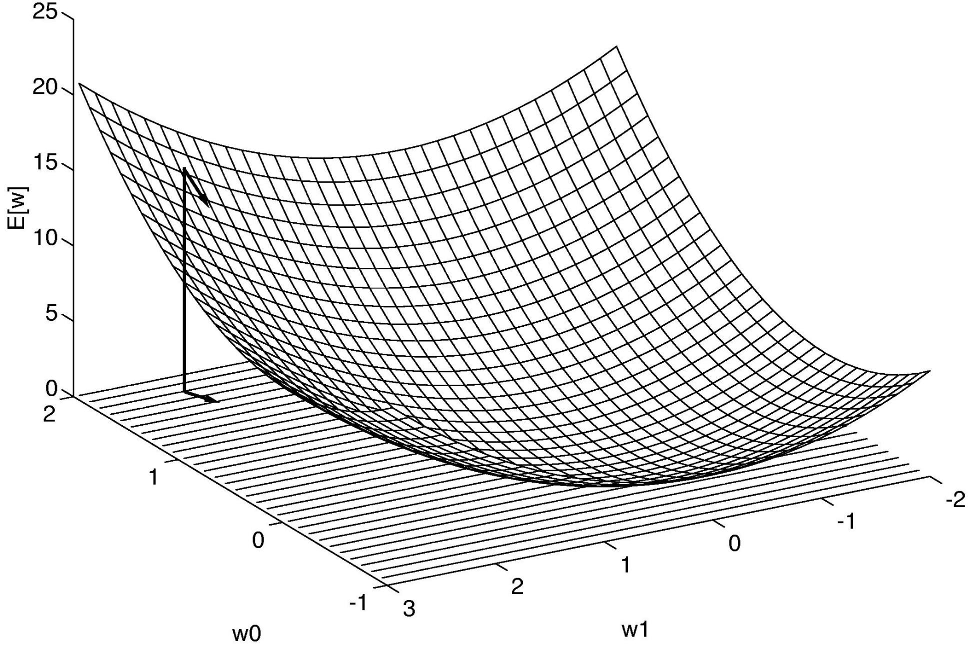


Illustration: Tom Mitchell, McGraw-Hill

```python
from theano import tensor, function, grad
import scipy
import numpy as np

def train_linear(examples, targets, learning_rate=0.01, steps=100):
    # Definition of the training data
    X = tensor.matrix("data")
    Y = tensor.matrix("targets")
    # Definition of the parameter vector of the model
    W = tensor.matrix("weights")

    examples = np.array(examples)
    targets = np.array(targets)

    # Definition of the model
    model = tensor.dot(X, W)
    # Definition of the error and of the loss function
    error = ((model - Y) ** 2).sum()
    grad_error = function([W, X, Y], grad(error, W))

    # Learning algorithm
    weights = scipy.random.standard_normal((examples.shape[1], 1))
    print("Initializing random weights: {0}".format(weights))
    for i in range(steps):
         weights -= learning_rate * grad_error(weights, examples, targets)

    print("Model trained {0} iterations, W={1}".format(steps, weights))
    return lambda x: np.dot(x, weights)
```

Perceptron

Perceptron : binary classifier

We saw how to learn a linear mapping between input values and targets.

Let’s see a close method for **classification**: the perceptron.

Learn an hyperplane to separate two class of data-points.

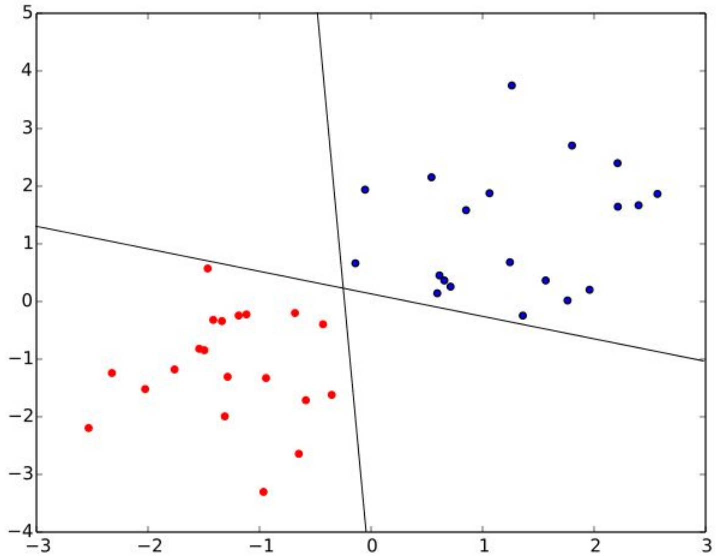


Illustration: Wikipedia, perceptron

Perceptron : binary classifier

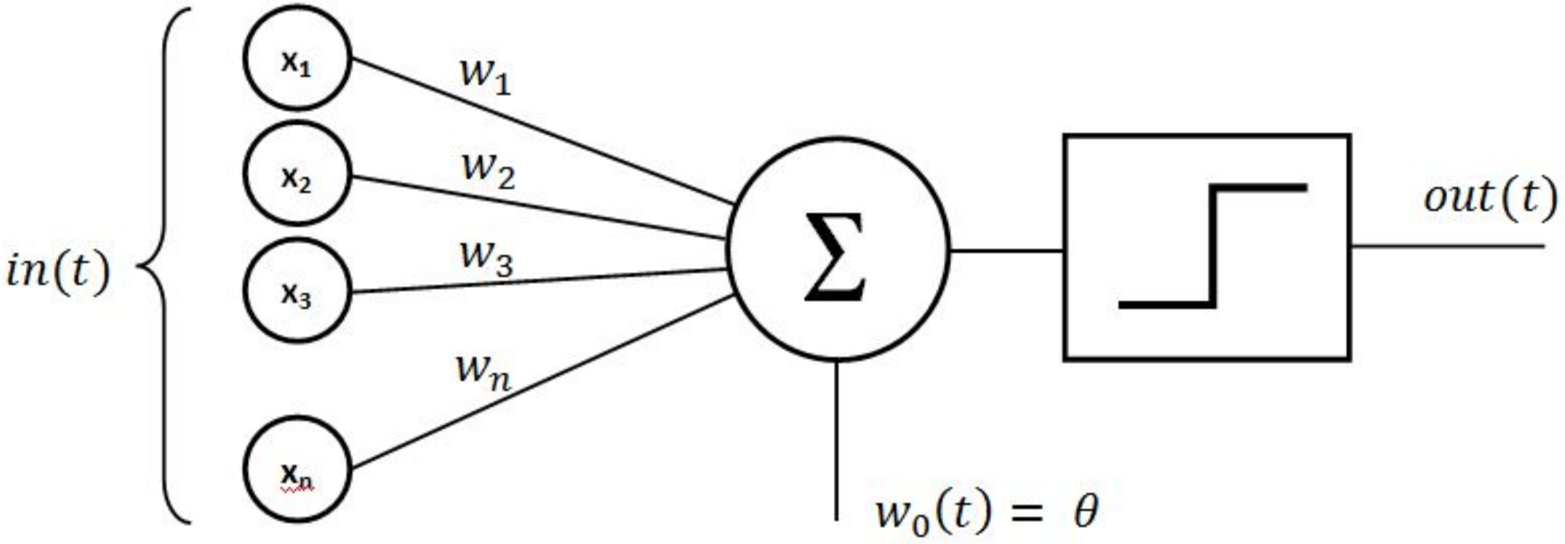


Illustration: Wikipedia, perceptron

Perceptron : binary classifier

We learn the parameters W of the function f:

$$f : X \rightarrow t(WX^T)$$

Where t is a transfer function

Example: The step function

- $t(x) = 1$ if $x > 0$
- $t(x) = -1$ otherwise

If the data point is above the hyperplane then it is a member of the class, otherwise it is not.

Perceptron : loss function

We can define the error of the perceptron by the number of elements it correctly classify… But this is a **hard problem** to optimize.

We use instead a **loss function** that we will minimize for each example.

One commonly used is:

$$L(x, y) = max(0, -xy)$$

The error function to minimize (by adjusting W) is:

$$E(W) = \sum_{i=0}^{n} L(WX_i^T, Y_i)$$

```python
# Train a linear classifier and returns the learned linear function
def train_perceptron(examples, targets, learning_rate=0.01, steps=100):
    # Definition of the training data
    X = tensor.matrix("data")
    Y = tensor.vector("targets")
    # Definition of the parameter vector of the model
    W = tensor.vector("weights")
    b = tensor.scalar("bias")

    examples = np.array(examples)
    targets = np.array(targets)

    # Definition of the model
    model = tensor.dot(X, W) + b
    # Definition of the error and of the loss function
    loss = -model * Y * ((tensor.sgn(-model * Y) + 1) / 2)
    error = loss.sum()
    grad_weights = function([W, b, X, Y], grad(error, W))
    grad_bias = function([W, b, X, Y], grad(error, b))

    # Learning algorithm
    weights = scipy.random.standard_normal(examples.shape[1])
    bias = 0.0
    print("Initializing random weights: {0}".format(weights))
    for i in range(steps):
        weights -= learning_rate * grad_weights(weights, bias, examples, targets)
        bias -= learning_rate * grad_bias(weights, bias, examples, targets)

    print("Model trained {0} iterations, W={1}, b={2}".format(steps, weights, bias))
    return lambda x: np.dot(x, weights) + bias
```

Principal component analysis

Principal component analysis

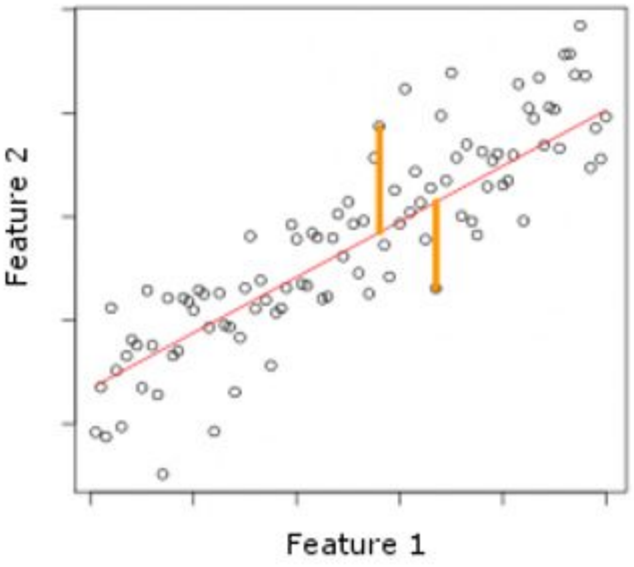
Motivation: Find a linear projection of the features of the dataset to reduce the dimensionality or attenuate the noise.

Method:

- Find new orthogonal components by maximizing the variance over each component
- It is also the set of orthogonal components which minimize the projection distance

Principal component analysis

Linear regression



PCA

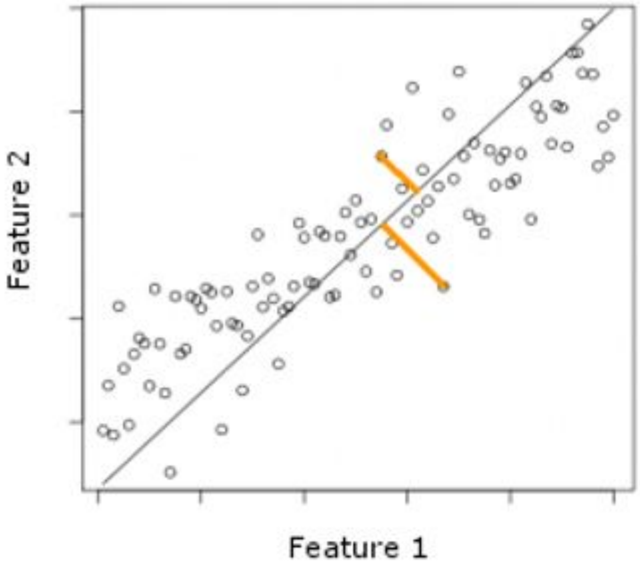
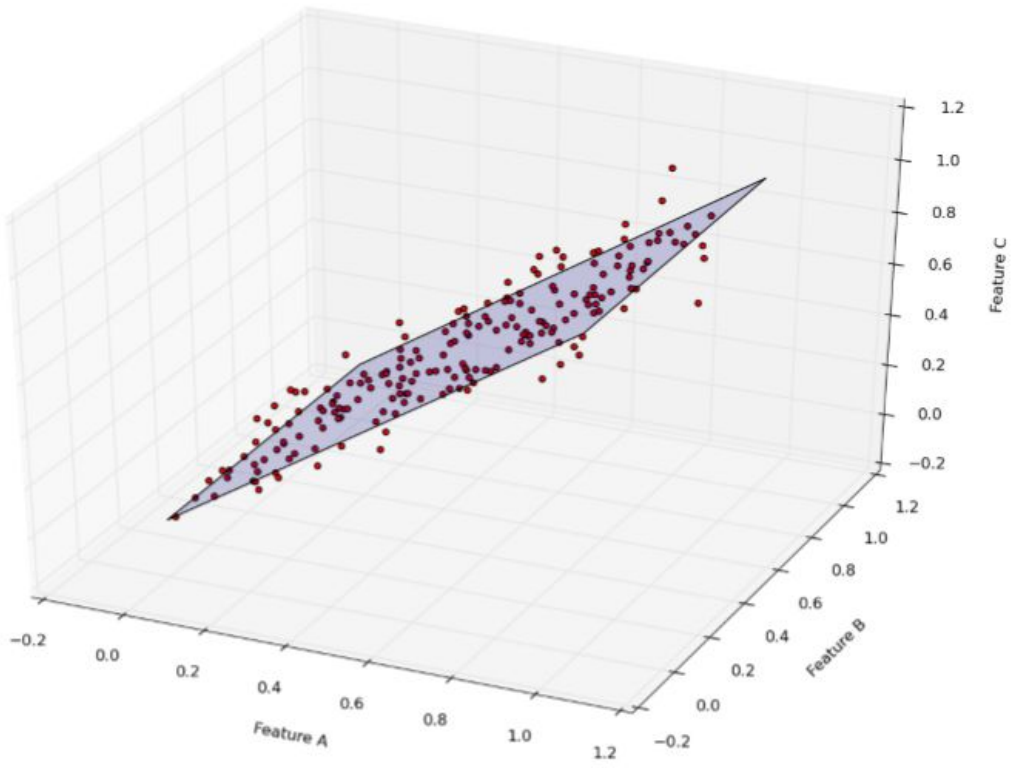


Illustration : [Stack Exchange]

Principal component analysis



Principal component analysis : covariance matrix

X and Y are two columns of the training set, each containing the value of one feature for all the examples.

$$\mathrm{Cov}(X,Y) \equiv \mathrm{E}[(X - \mathrm{E}[X])\,(Y - \mathrm{E}[Y])]$$

$$cov(X,Y) = \frac{1}{N}\sum_{i=0}^{n}(X_i - mean(X))(Y_i - mean(Y))$$

Covariance matrix:

$$\mathrm{Var}(\vec{X}) = \begin{pmatrix} \mathrm{Var}(X_1) & \mathrm{Cov}(X_1,X_2) & \cdots & \mathrm{Cov}(X_1,X_p) \\ \mathrm{Cov}(X_2,X_1) & \ddots & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{Cov}(X_p,X_1) & \cdots & \cdots & \mathrm{Var}(X_p) \end{pmatrix} = \begin{pmatrix} \sigma^2_{x1} & \sigma_{x1x2} & \cdots & \sigma_{x1x_p} \\ \sigma_{x2x1} & \ddots & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{x_px1} & \cdots & \cdots & \sigma^2_{x_p} \end{pmatrix}$$

Principal component analysis : algorithm

1. Normalize the dataset
2. Compute the covariance matrix
3. Compute the eigenvectors and eigenvalues of the covariance matrix
4. Sort the eigenvectors by eigenvalues
5. Build a matrix B with the k first eigenvectors (one per row)
6. Project the dataset with $p(x) = BX$

Support vector machine

And the kernel trick

Support vector machine

Objective: Find the hyperplane with the biggest margin separating two classes.

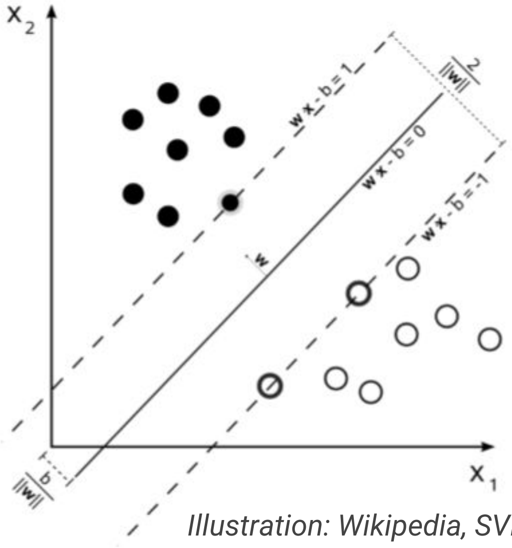
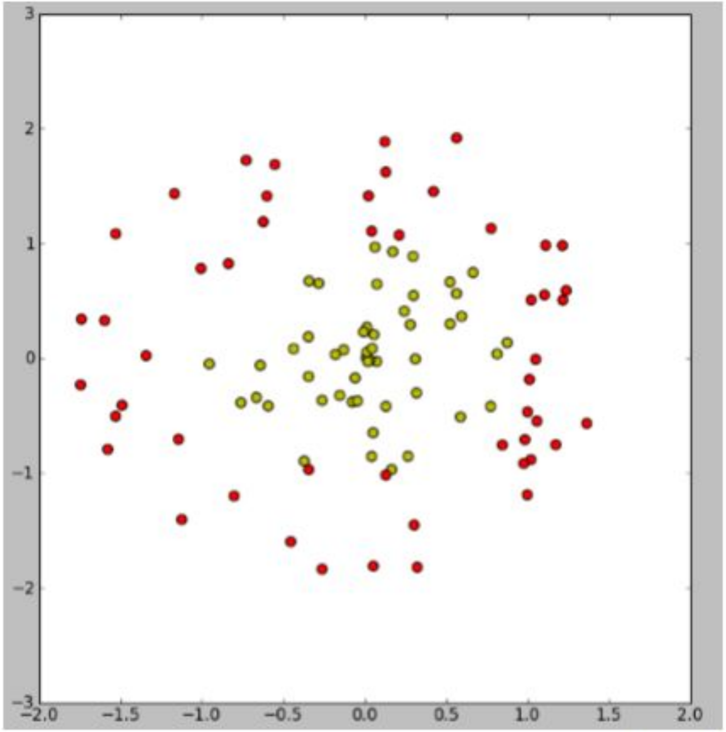
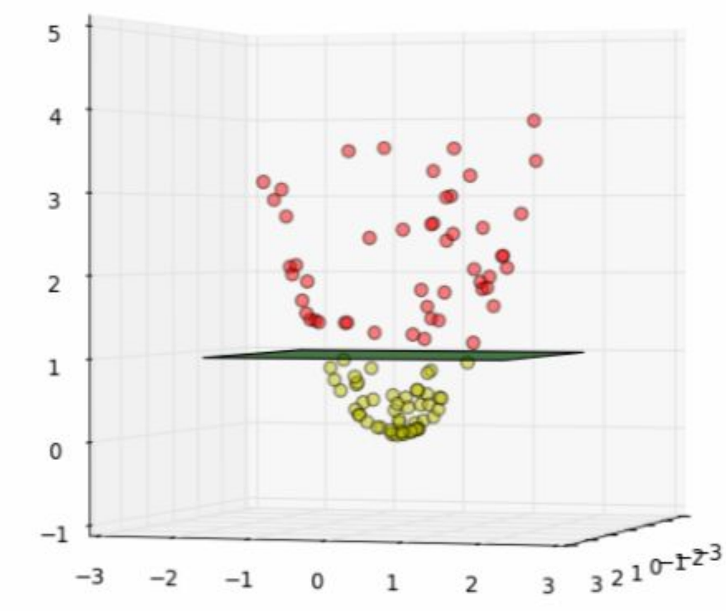


Illustration: Wikipedia, SVM

Support vector machine : nonlinear case



Kernel trick



$$\varphi : (x, y) \mapsto (x, y, x^2 + y^2)$$

Kernel trick

Some algorithms only require the dot product between 2 vectors

Instead of computing $\varphi(x) \cdot \varphi(y)$ **by:**

1. projecting in a higher dimensional space
2. compute the dot product

We can use the kernel trick:

$$K(x, y) = \langle \varphi(x), \varphi(y) \rangle$$

There exists some functions (Mercer’s condition) which are cheap to compute and can be expressed as a dot product in a higher dimensional space.

Kernel trick

- Polynomial kernel $K(x,y) = (x^\mathsf{T} y + c)^d$
- RBF kernel $K(\mathbf{x},\mathbf{x'}) = \exp\left(-\frac{||\mathbf{x}-\mathbf{x'}||^2}{2\sigma^2}\right)$

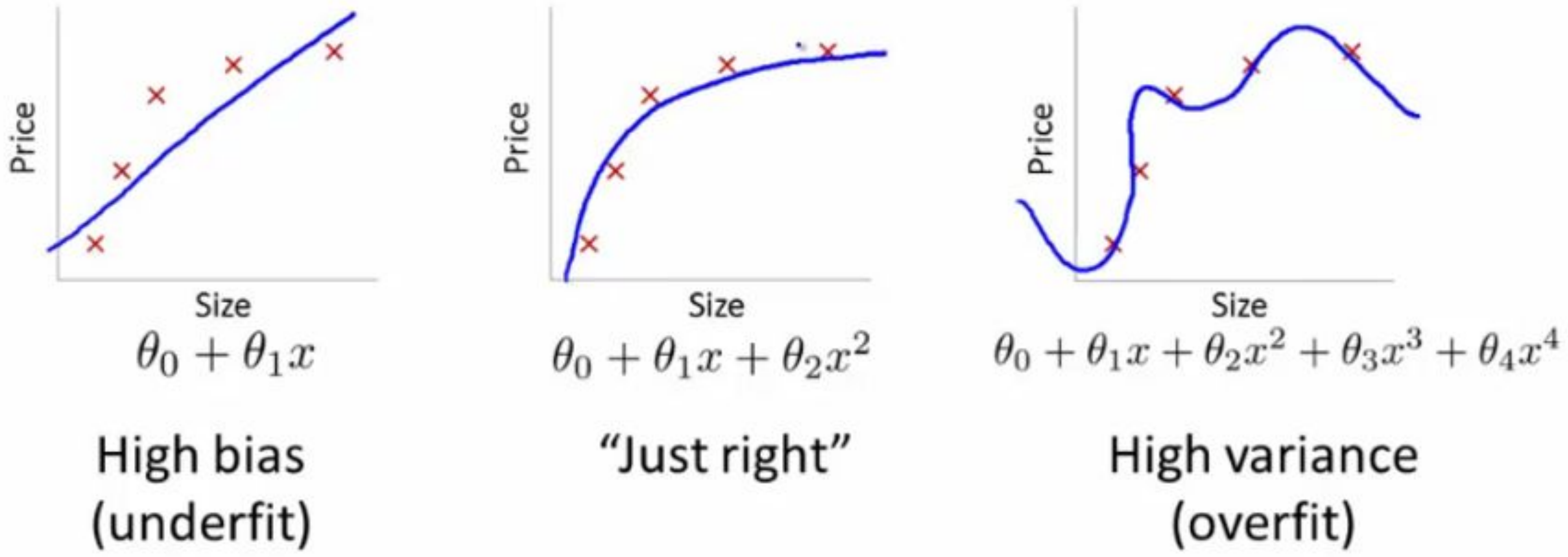
Bias and variance

Bias and variance

- When applying a ML algorithm, we do some assumptions to learn the model:
    - k-nn: neighborhood influence
    - linear model: linear mapping between input and output, loss function
    - perceptron: we can separate the data linearly, loss function
    - SVM: kernel type, hyperparameters
- Those assumptions will influence the **bias** and the **variance** of the model

Bias and variance

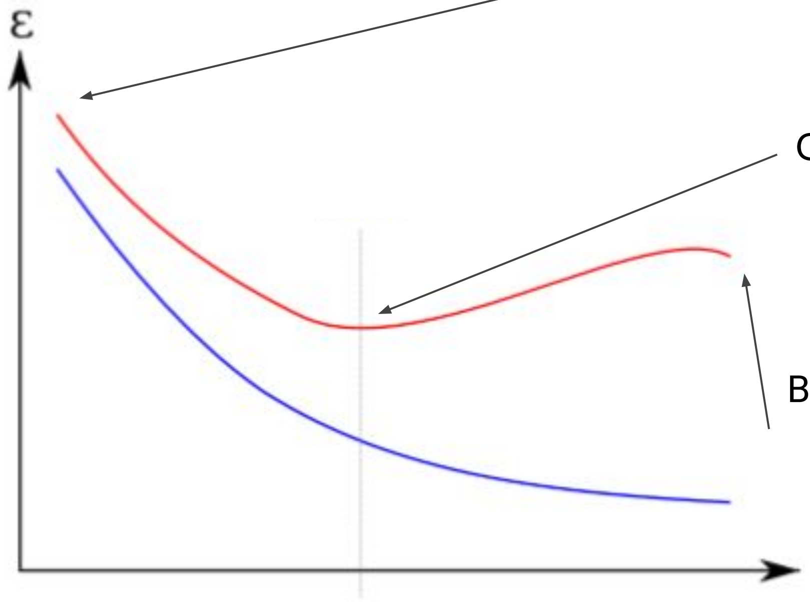
Illustration: Andrew Ng



Bias variance

Big error (underfitting):
- High bias
- Low variance

Good tradeoff



Big error rate (overfitting):
- Low bias
- High variance

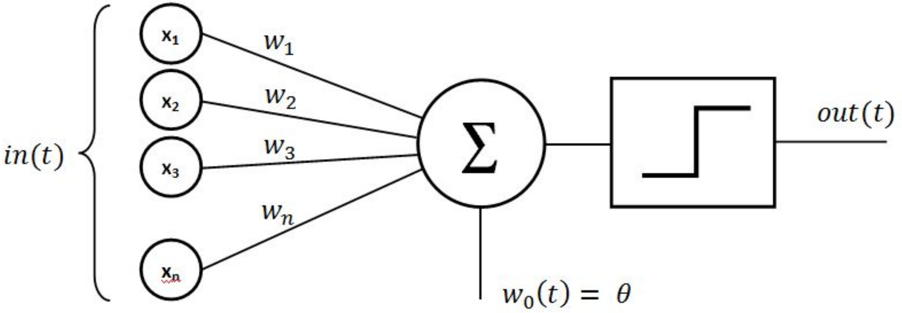
High bias Low bias

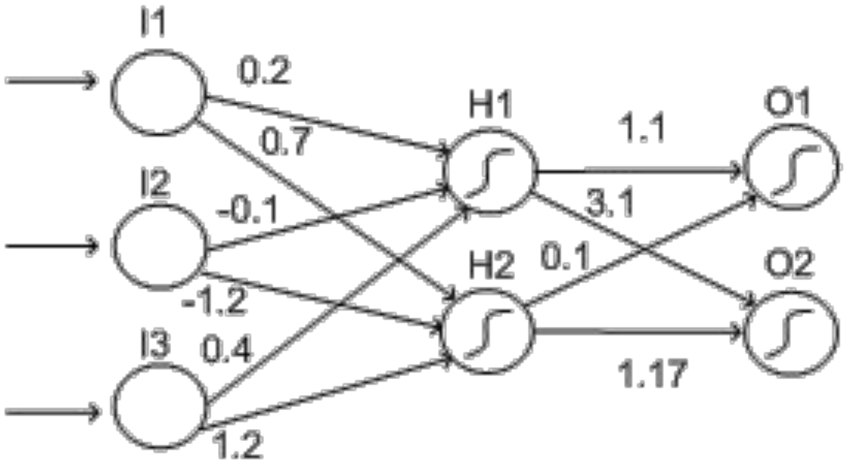
Neural networks

(Multilayer perceptron)

Neural networks

- From the **perceptron** seen previously we can build a multilayer perceptron





By chaining multiple non-linear function we allow the model to learn more **complex relationships** between input and output.

Neural network : forward pass

The **forward pass** consists in evaluating the output of the neural network.

```
[ ]  def forward(nn, X):
       for layer in nn.layers:
         next_input = []
         for neuron in layer.neurons:
           next_input.append(neuron.activation(X * neuron.W))
         X = next_input
       return X
```

It compute the output from the first layer to the last layer.

Neural networks : backward pass

- We can compute easily the gradient of the error with respect to the weight of the last layer
- The gradient of the error with respect to the weight of the others layers are computable by applying the chain rule : $(f \circ g)' = (f' \circ g) \cdot g'$.

[ ]

```
def backward(nn, X, learning_rate):
  for layer in reversed(nn.layers):
    for neuron in layer.neurons:
      neuron.W -= learning_rate * gradient(nn.error, neuron)
```

Neural network : matrix optimization

It is possible to see the neural network as multiple matrix operations by considering each layer as a matrix of weights.

It simplify the algorithm and it’s way faster at training time!

- Hardware optimization
- Can be executed on GPU

```
        self.layers[-1].initialize(self.rng)
        self.params = self.params + self.layers[-1].params

    def output(self):
        assert (len(self.layers) > 0), "The network needs to contain at least one layer."
        return self.layers[-1].output()

    def pop_layer(self):
        if len(self.layers) > 0:
            self.layers.pop()
            self.params = []
            for layer in self.layers:
                self.params += layer.params

    def build_train(self, learning_rate, regularization_factor):
        assert (len(self.layers) > 0), "The network needs to contain at least one layer."
        labels = T.matrix("labels", dtype=theano.config.floatX)

        cost = T.sum((self.output() - labels) ** 2)

        for layer in self.layers:
            if layer.regularization() is not None:
                cost = cost + regularization_factor * layer.regularization()
        gparams = [T.grad(cost, param) for param in self.params]
        updates = [
                (param, param - learning_rate * gparam) for param, gparam in zip(self.params, gparams)
                ]

        return theano.function(
            inputs = [self.layers[0].input, labels],
            outputs=cost,
            updates=updates
```

Deep learning

A gentle introduction

Autoencoders

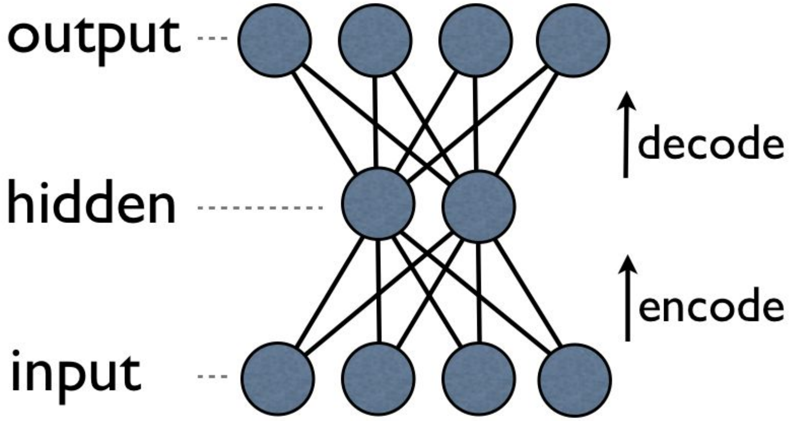
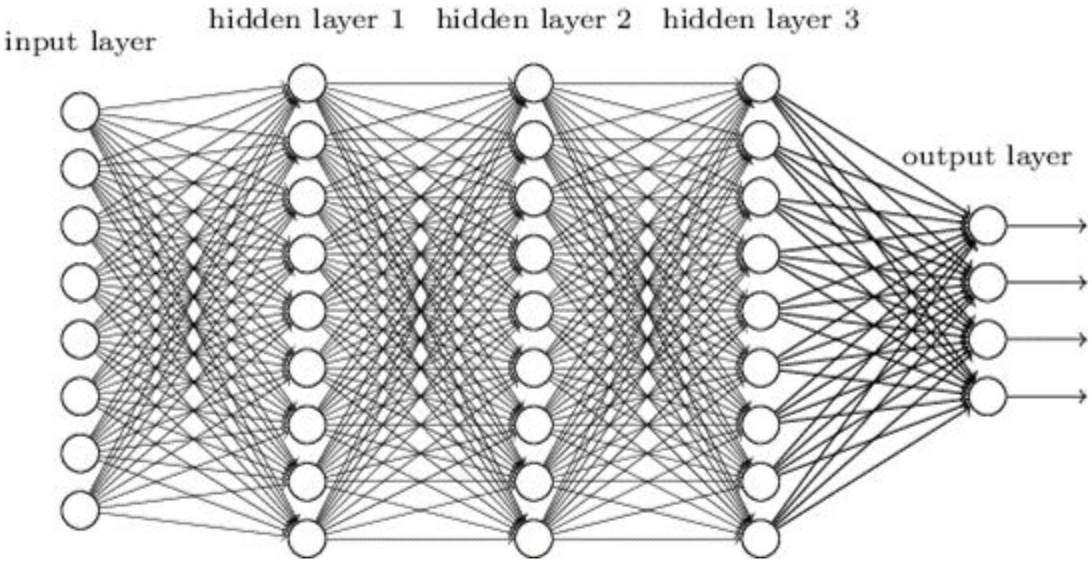


Illustration: *[Stack exchange]*

- Learn the identity function (unsupervised learning)
- The data is compressed then reconstructed
- The hidden layer is called bottleneck
- It contains an “embedding” of the input

In [3]:
```
def build_nn():
    print("Building NN")
    nn = fullyconn.MLP(40 * 40 * 3)
    # ReLU
    nn.add_layer("bottleneck", 90)
    nn.add_layer("reconstruction", 40 * 40 * 3)
    return nn
```

Shallow VS deep networks



Deep learning : convolutional neural network

- High resolution images contains O(millions) of pixels
- A neural network which can handle that kind of images would also have O(millions) of weight

Solution: Repeat the same part of the network over the whole image area

Deep learning : convolutional filter

Example: Edge detection

Deep learning : convolutional neural network

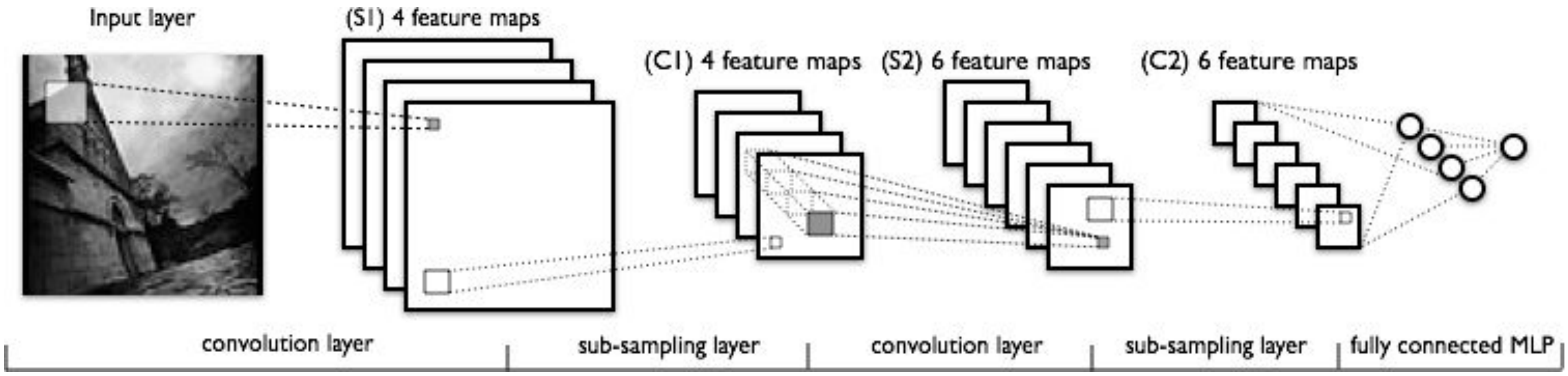


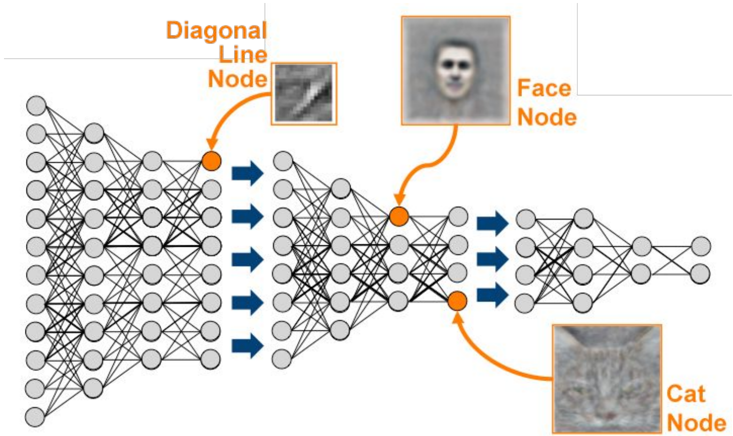
Illustration: Theano documentation

This kind of architecture will learn filters and build an internal representation of the input data using many stacked layers and finally use this representation on a classification task.

Learn high level features of a cat

“Best neuron” activation heat map

- Training: 16.000 CPU during 3 days
- Learned high levels features of cats, human faces by watching Youtube videos
- Totally unsupervised : unlabeled data



Building High-level Features Using Large Scale Unsupervised Learning
Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, Andrew Ng

Deep learning : convolutional neural networks

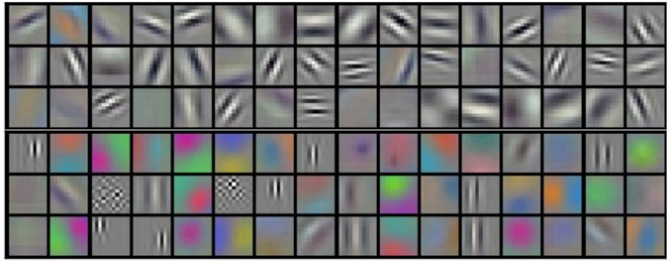
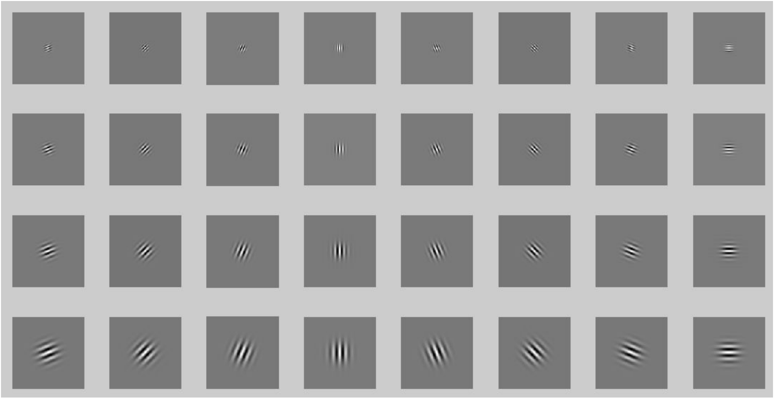


Figure 3: 96 convolutional kernels of size $11\times11\times3$ learned by the first convolutional layer on the $224\times224\times3$ input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.

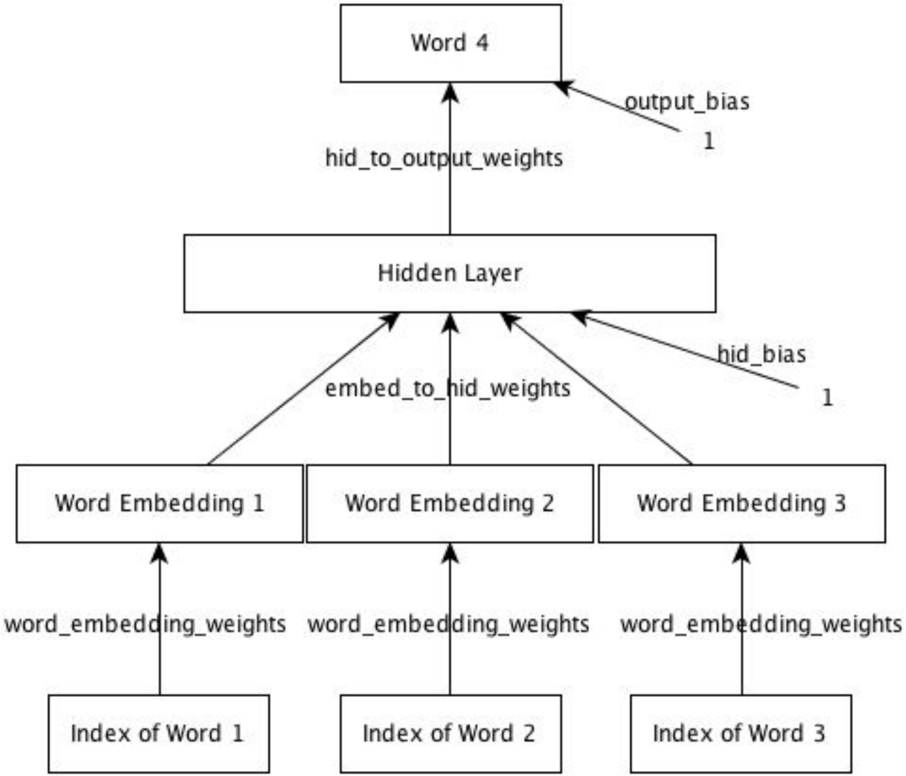
The model learns some edge detection filter.
We find a similar process in the cells of the primary visual cortex of the human brain

Edge detectors filters :



ImageNet Classification with Deep Convolutional Neural Networks
Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton

Word embedding models



Word2vec : distance metric

Nearest neighbors of “France” :

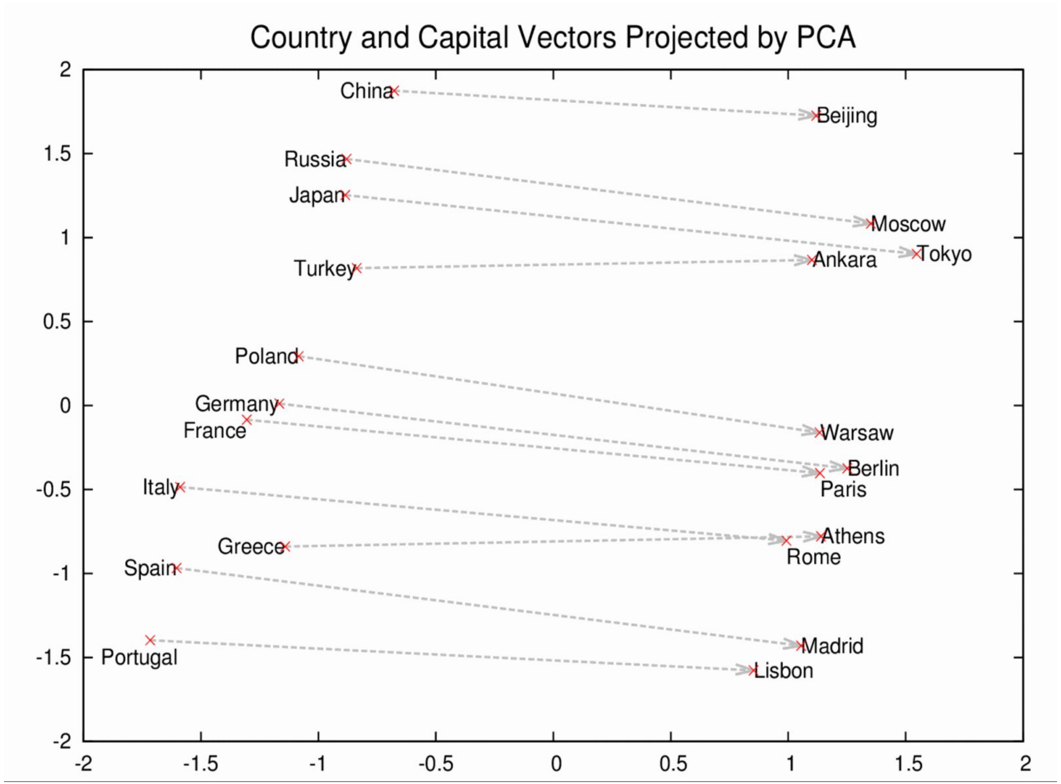
Word	Cosine distance
spain	0.678515
belgium	0.665923
netherlands	0.652428
italy	0.633130
switzerland	0.622323
luxembourg	0.610033
portugal	0.577154
russia	0.571507
germany	0.563291
catalonia	0.534176

The model learn embeddings (a float vector) to represent words such as two words close in the semantic space are close in the embedding space

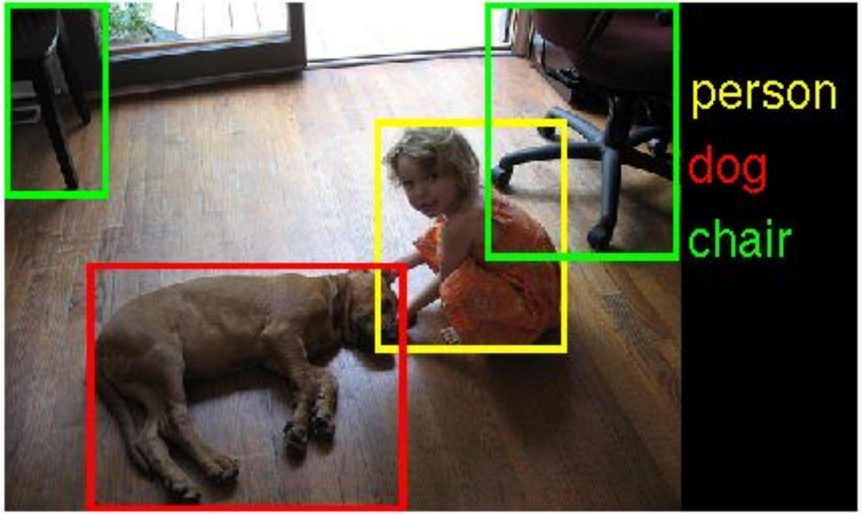
Cosine distance (~ L2 distance when vectors are normalized)

word2vec - https://code.google.com/p/word2vec/

Word2vec : PCA data visualisation



ImageNet challenge



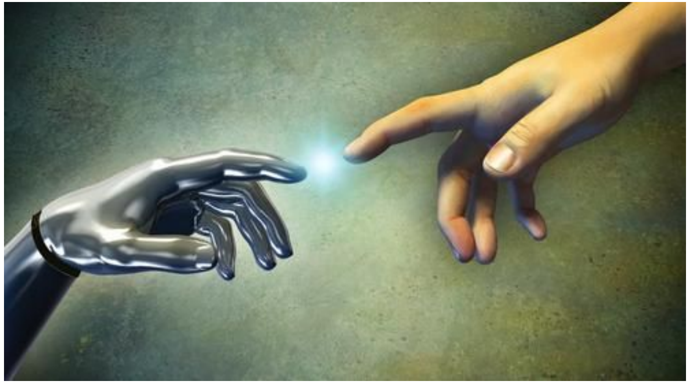
Visit : http://www.image-net.org

- Detection and classification of images over 1000 different classes
- Deep learning leads to a breakthrough in prediction quality
- GoogleNet is the architecture who won the challenge in 2014

ImageNet : Humans vs Machine

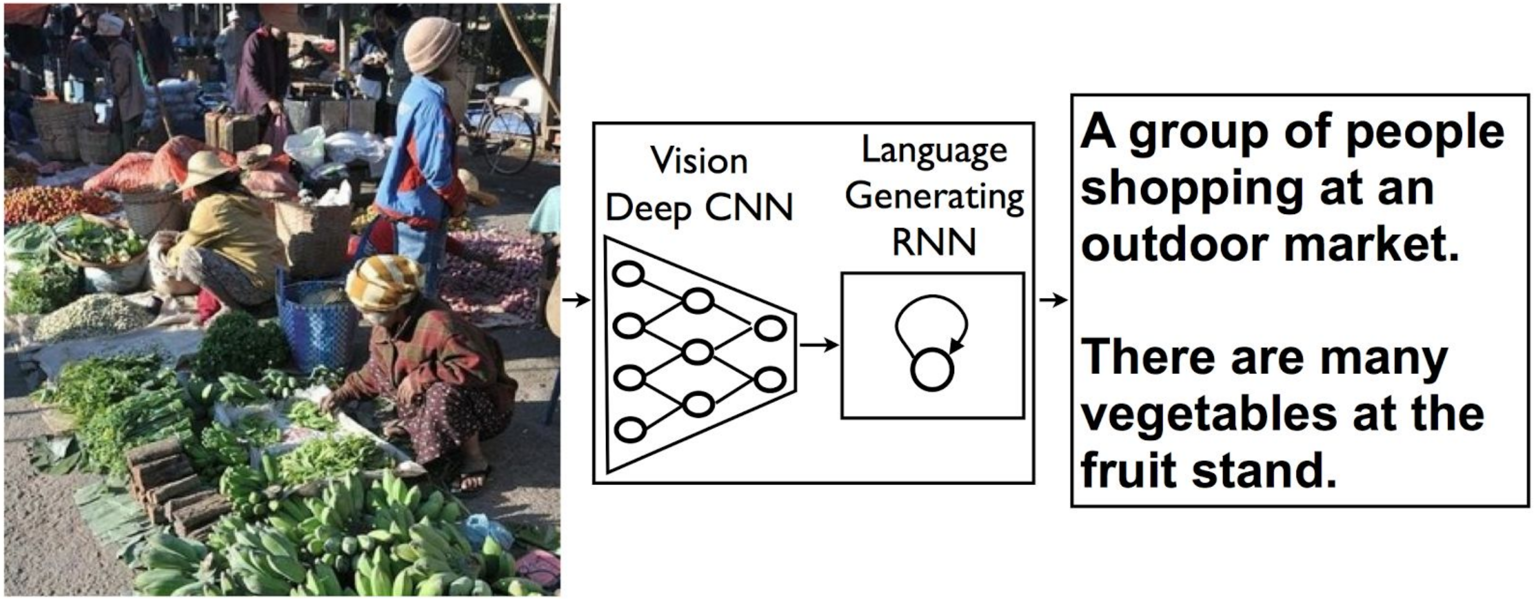
Relative Confusion	A1	A2
Human succeeds, GoogLeNet succeeds	1352	219
Human succeeds, GoogLeNet fails	72	8
Human fails, GoogLeNet succeeds	46	24
Human fails, GoogLeNet fails	30	7
Total number of images	1500	258
Estimated GoogLeNet classification error	6.8%	5.8%
Estimated human classification error	5.1%	12.0%

Table 9 Human classification results on the ILSVRC2012-2014 classification test set, for two expert annotators A1 and A2. We report top-5 classification error.



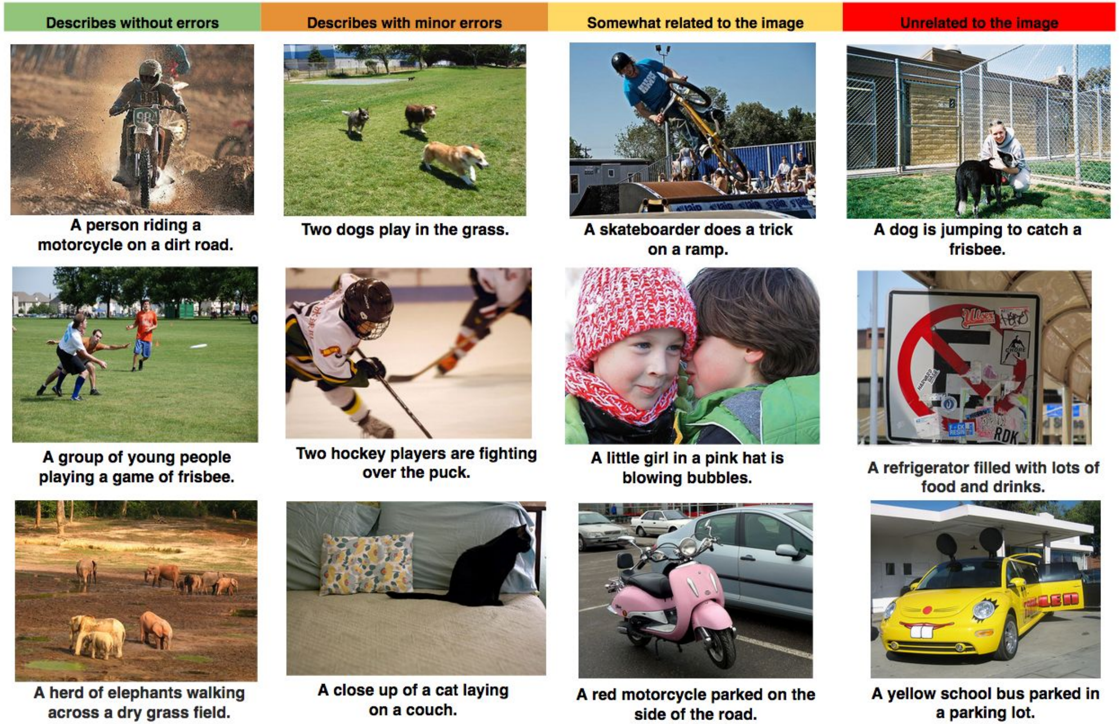
ImageNet Large Scale Visual Recognition Challenge
arXiv:1409.0575 - Olga Russakovsky, Jia Deng et al.

Image description : conv-nn + LSTM



Show and Tell: A Neural Image Caption Generator
arXiv:1411.4555 - Oriol Vinyals, Alexander Toshev, Samy Bengio, Dumitru Erhan

Image description : conv-nn + LSTM



Show and Tell: A Neural Image Caption Generator

arXiv:1411.4555 - Oriol Vinyals, Alexander Toshev, Samy Bengio, Dumitru Erhan