Machine Learning

Some typical machine learning problems are:

- Predict whether a patient, hospitalized due to a heart attack, will have a second heart attack. The prediction is to be based on demographic, diet and clinical measurements for that patient
- Predict the price of a stock in 6 months from now, on the basis of company performance measures and economic data
- Identify the numbers in a handwritten ZIP code, from a digitized image

A typical task of learning from data has an outcome (target) measurement (quantitative or categorical) that we wish to predict based on a set of features.

Supervised vs. Unsupervised Learning

A big portion of the ML problems can be divided into Supervised and Unsupervised

- Supervised Learning
  - Supervised Learning is where both the features and the target are observed.
- Unsupervised Learning
  - In this situation, only the features are observed.
  - A common example is market segmentation where we try to divide potential customers into groups based on their characteristics
  - A common approach is clustering

Regression vs. Classification

- Supervised learning problems can be further divided into regression and classification problems.
- Regression covers situations where the target is continuous/numerical, e.g.
  - Predicting the value of the Dow in 6 months
  - Predicting the value of a given house based on various inputs
- Classification covers situations where Y is categorical, e.g.
  - Will the Dow be up (U) or down (D) in 6 months?
  - Is this email a SPAM or not?

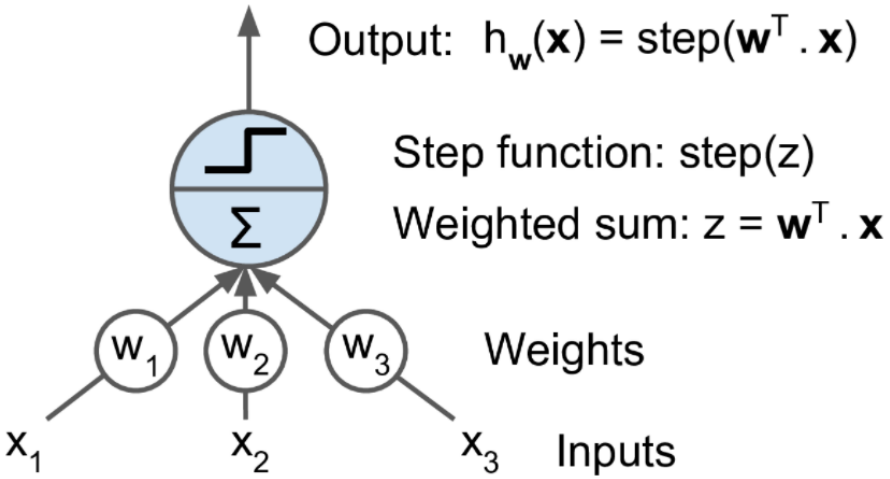
Neural Networks (Chap10 in [HO])

- Artificial Neural Networks (ANN) are a form of connectionism - computing systems inspired by the biological neural networks that constitute animal brains. ANN can be used to accomplish any of the ML tasks we mentioned and more
- An ANN is based on a collection of connected units called artificial neurons
- ANNs have been around for quite a while: they were first introduced in 1943 by the neurophysiologist W. McCulloch and the mathematician Walter Pitts
  - they presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic
  - the first ANN architecture (with binary inputs)
- But for long there were challenges with ANN architectures - causing the consecutive *ML winters*

- Perceptron - the simplest ANN architecture (F. Rosenblatt 1957), based on an artificial neuron called the linear threshold unit (LTU):
  - inputs and outputs are numbers instead of binary on/off values
  - each input connection has an associated weight
  - all inputs are aggregated as a weighted sum

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_k x_k = \boldsymbol{x}^T \boldsymbol{w}$$

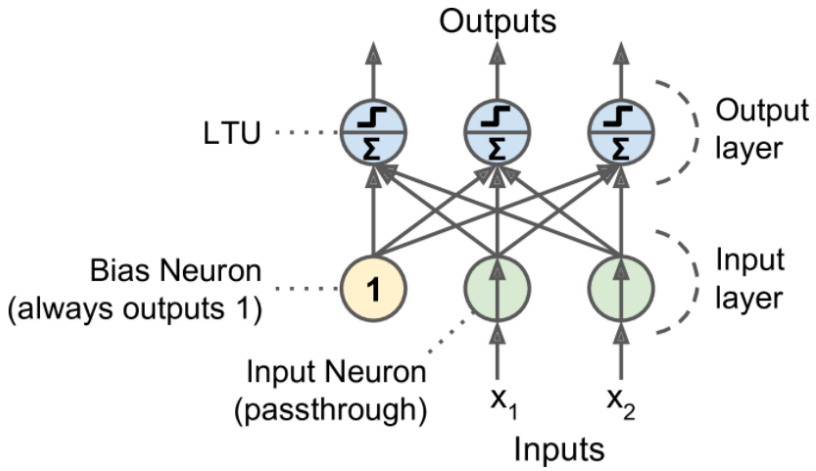
  - then a "step" function (an activation function) is applied to it:



most common activation function for **classification**:

$$step(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

- The Perceptron is composed of a single layer of LTUs
  - each neuron connected to all the inputs - which are usually represented as input neurons
  - input neurons: just output whatever input they are fed
  - an extra feature is added to the input neurons, called the bias neuron which just outputs 1 all the time
  - for example, a Perceptron capable of 3 different binary classes simultaneously (multioutput classifier):



If all the neurons in a layer are connected to every neuron in the previous layer, it's called a dense layer. The output in this case is

$$h_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{X}) = \phi(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b})$$

ϕ –activation function

$\boldsymbol{X}$ – inputs, $\boldsymbol{W}$ – weights, $\boldsymbol{b}$ - bias vector

- How is a Perceptron trained?
  - Training is inspired by how biological neurons are believed to be trained. Hebb suggested that when a biological neuron often triggers another neuron, the connection between these two neurons grows stronger
  - the Perceptron training's trying to replicate Hebb's rule:
    - the Perceptron is fed one training instance at a time, and for each instance it makes its prediction
    - for every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction

Weights Update

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

$w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.

x_i is the i^{th} input value of the current training instance.

$\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.

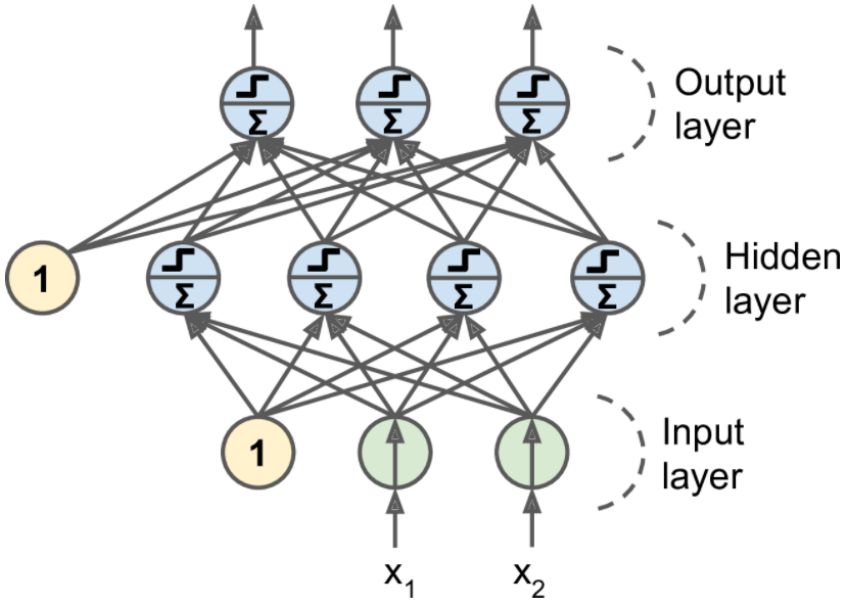
y_j is the target output of the j^{th} output neuron for the current training instance.

η is the learning rate.

Look also here:

https://en.wikipedia.org/wiki/Perceptron

- Multilayer Perceptron (MLP)
  - Consists of one input layer and one or more layers of LTUs, also called hidden layers (plus the output layer)
  - Every layer (except the output layer) includes a bias neuron and is fully connected to the next layer
  - When an ANN has two or more hidden layers then it is called a deep neural network (DNN) => Deep Learning



- How is an MLP / DNN trained?
  - for decades, there was no good solution for training the DNNs (caused an ML Winter)
  - in 1986 Rumelhart published the backpropagation training algorithm, which is basically Gradient Descent (GD) using reverse-mode autodiff
  - these two passes are called forward and backward

- Forward pass

  For each training instance, the algorithm feeds it to the network and computes the output of every neuron in each consecutive layer (just like when making predictions)

- Reverse pass: To measure the error gradient (for classification or regression) across all the connection weights in the network by propagating the error gradient backward in the network:

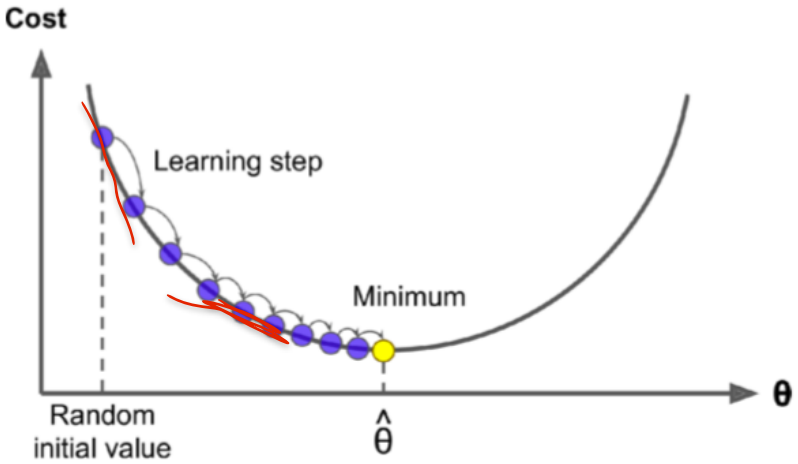
  – it measures the network's output error (i.e., the difference between the desired output and the actual output of the network), and it computes how much each neuron in the last hidden layer contributed to each output neuron's error and slightly tweaks the connection weights to reduce the error (Gradient Descent – see next slide)

  – it then proceeds to measure how much of these error contributions came from each neuron in the previous hidden layer - and so on until the algorithm reaches the input layer. This is why it is called backpropagation or backprop, as it propagates the errors back to the weights of the neurons which contributed to the error

What is Gradient Descent? (HO-Ch4)

- It is a generic optimization algorithm for finding optimal solutions to a wide range of problems.

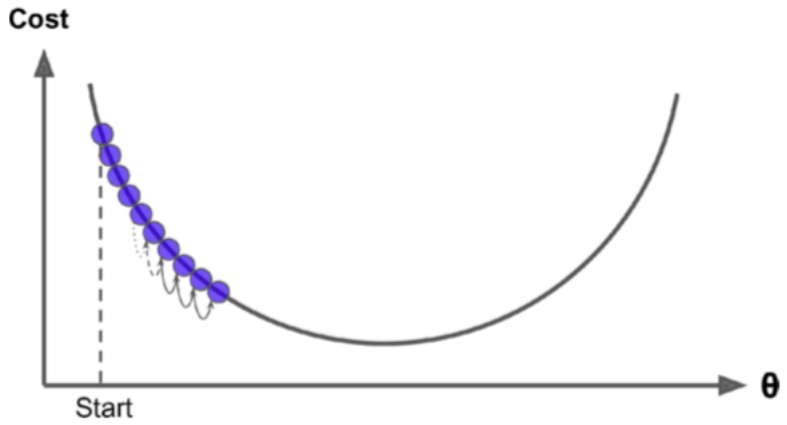
- The general idea of Gradient Descent is to tweak parameters $\boldsymbol{\theta}$ iteratively (starting with a random value) in order to minimize a cost function.

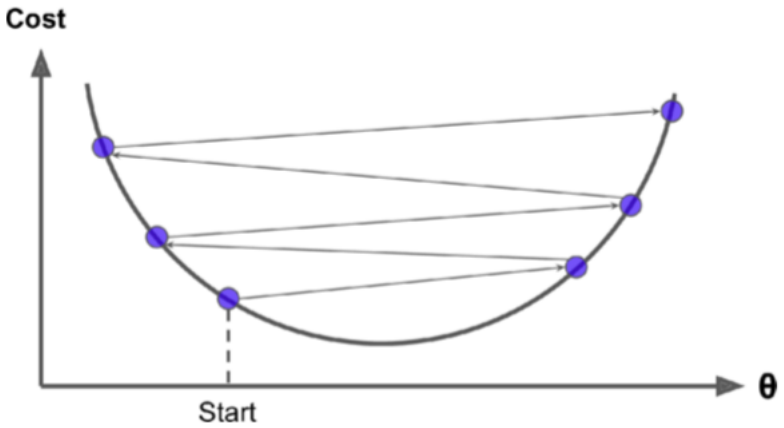


$$\nabla f(\hat{\theta}) = 0$$

- The *learning rate* determines the size of the steps. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (left plot).

- If it is too large, it might make the algorithm diverge (right plot)





- General optimization problem and Gradient Descent (GD) solution

Problem $\underset{w}{\text{minimize}}\ f(w)$ We denote the optimizing input w_*

Solution (iterative) $w_{k+1} = w_k - \gamma_k \nabla f(w_k)$

Step size

Next iterate Previous iterate Gradient of f

- One example, is the Linear Regression problem. Change notation: $w \leftrightarrow \theta$

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_k$$
$$\hat{y} = h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^T \boldsymbol{x}$$

Here we want to find $\widehat{\boldsymbol{\theta}}$ which minimizes the criterion (loss)

$$f(\theta_k) = MSE(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}\right)^2, \quad \nabla f = \frac{2}{m}\mathbf{X}^{\mathrm{T}}(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

- Illustration: Gradient Decent and Linear Regression

Equation 4-7. Gradient Descent step

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$$

where

$$\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^\intercal (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

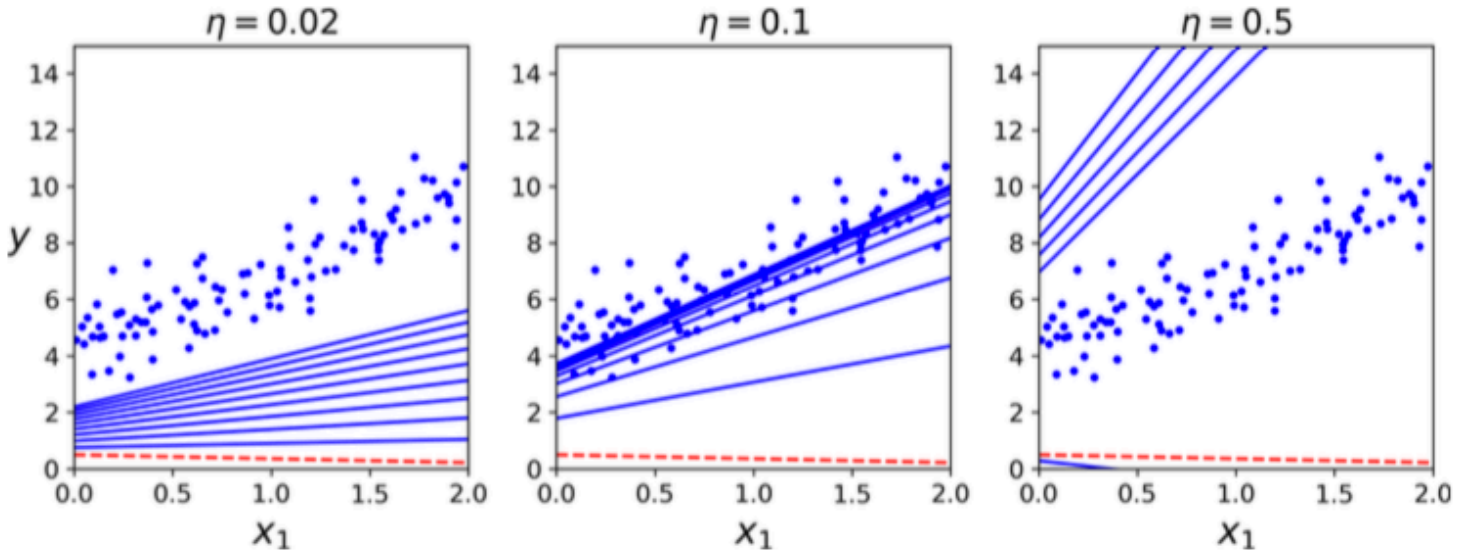
```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\intercal \mathbf{X})^{-1} \quad \mathbf{X}^\intercal \quad \mathbf{y}$$

- The above quick implementation finds the exact soultion! But if the learning rate is different this may not happen as is illustrated with the first 10 steps below



- What was illustrated above is the so-called Batch GD (use all the observatoins at once)
- To understand better how the learning rate affects converegence in GD consider also Quadratic Optimization problem

The Quadratic (Positive Definite) problem

$$f(w) = \frac{1}{2} w^\top A w - b^\top w$$
$$\nabla f(w) = Aw - b$$
$$w_* = A^{-1} b$$

$$w_{k+1} = w_k - \gamma \nabla f(w_k)$$

$$\begin{aligned}
\|w_{k+1} - w_*\| &= \|w_k - \gamma(Aw_k - b) - w_*\| \\
&= \left\|w_k - \gamma A(w_k - A^{-1}b) - w_*\right\| \\
&= \|w_k - \gamma A(w_k - w_*) - w_*\| \\
&= \|(I - \gamma A)(w_k - w_*)\| \\
&\leq \|I - \gamma A\| \, \|w_k - w_*\|
\end{aligned}$$

Let μ be the minimum eigenvalue of A and L the max. Then the extremal eigenvalues of $I - \gamma A$ are $1 - \mu\gamma$ and $1 - L\gamma$, so:

$$\|I - \gamma A\| = \max\{|1 - \mu\gamma|, |1 - L\gamma|\}$$

If we use two large a step size $1 - L\gamma$ becomes negative. A standard value is $\gamma = 1/L$ which gives:

$$\|w_{k+1} - w_*\| \leq \left(1 - \frac{\mu}{L}\right) \|w_k - w_*\|$$

- We can see that the convergence of optimization methods depends on the condition number

Inverse of the condition number

$$\|w_{k+1} - w_*\| \leq \left(1 - \frac{\mu}{L}\right) \|w_k - w_*\|$$

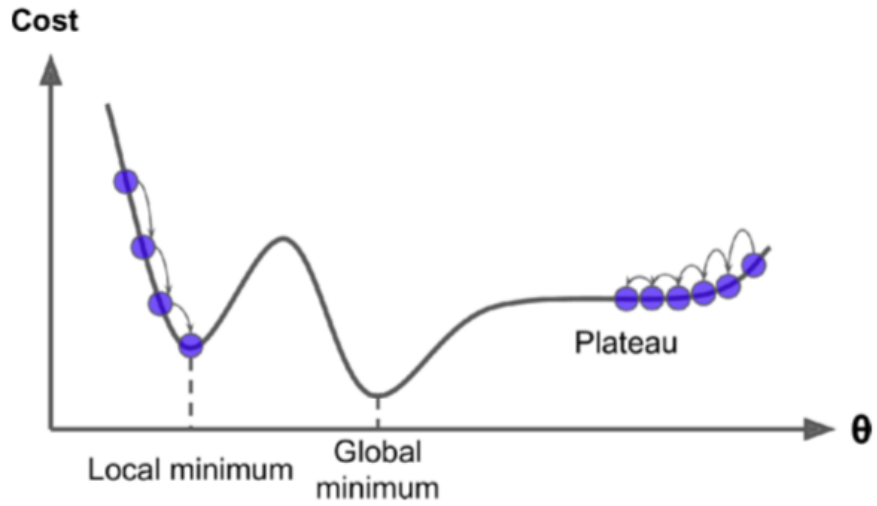
- Although the condition number makes sense formally only for simple problems ("strongly convex") we can still use it informally.

- This prompts us to take care of the learning rate – it is thus a hyper parameter.

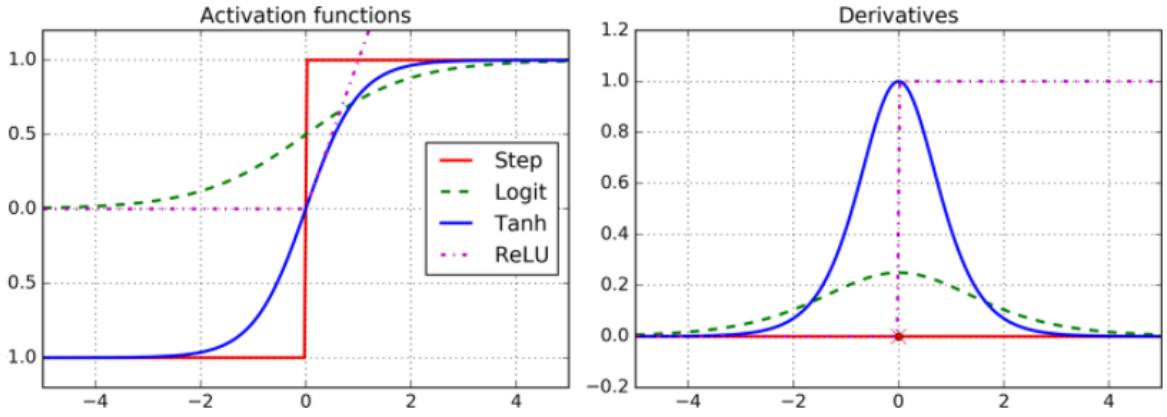
- A Neural Network can have multiple hyper parameters – we will consider them one by one

- Not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult.

- Plot below, shows the two main challenges with Gradient Descent. If the random initialization starts the algorithm on the left, then it will converge to a *local minimum*. Starting to the right, then it will take a very long time to cross the plateau



- Since GD does not work on a flat surface, the step activation function of the LTUs needed to be replaced with something that is continuous and differentiable everywhere for the backprop to work => activation function is a hyper-parameter too.
- Activation functions
  - ReLU – most used (not differentiable at zero, but fast to compute)
  - Hyperbolic tangent function (tanh)
  - Logit function
  - Sigmoid (not as good as ReLU, but biologically more accurate)



Neural Network Hyper parameters

- Number of hidden layers (MLP with just one hidden layer can theoretically model even the most complex functions, provided it has enough neurons)
- Number of neurons per hidden layer
- Learning rate
- Optimizer
- Batch size
- Activation functions

Perceptron Python code