MA2823: Foundations of Machine Learning
Chapter 10: Artificial Neural Networks

Lecturer: Chloé-Agathe Azencott Scribe: Meryll Dindin, Horace Guy

1 Introduction to human brain :

The motivation of neural networks comes from the humain brain. Our brain is made out of neurons, communicating through axones (10^{10} neurons and 10^4 connections). We recognize the advantages of parallel processing, distributed computation and memory, but it is also all about a robust system to noise and failures. To classify pictures thanks to a neural network will work the same way as our brain does, to identify features and be able to recognize the same time of pictures afterwards.

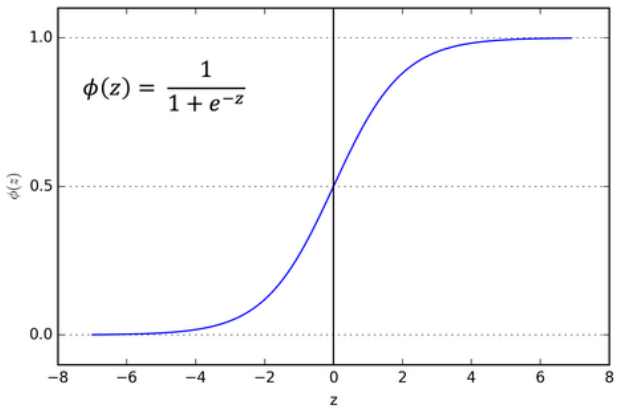
2 Perceptron

The model of perceptron has been introduced in 1958 thanks to Rosenblatt. The model presents layers : a labeled input, weighted connections and an output. It is a **parametric model**, since we know, before running the model, the shape of the model. Here it is a linear combination of the defined weights for each connection.

The model function looks like : $f(\boldsymbol{x}) = \sum_{j=1}^{p} w_j x_j + w_0 = \boldsymbol{w}^T \cdot \boldsymbol{x}$

Concerning the shape of the decision boundary, it is a **hyperplane**, as it is the case for *linear regression*. If we now want to output the probability of belonging to the positive class, we apply a **logistic function**. An example of it is the *sigmoid function* defined below :

$$\sigma(u) = \frac{1}{1 + exp(-u)}$$



To conclude with a short summary :

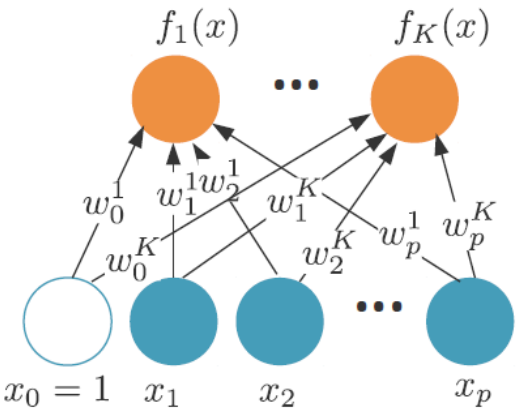
— Regression :

$$f(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + w_0$$

— Classification :

$$f(\boldsymbol{x}) = s(\boldsymbol{w}^T \boldsymbol{x}) = s(w_0 + \sum_{j=1}^{p} w_j x_j)$$

knowing $s(o(\boldsymbol{x})) = \mathbb{I}_{o(\boldsymbol{x})>0}$



Multi-Class One Layer Perceptron

For multiclass classification, the model is similar except for the specificity that we will rather use K output units. For each class, we reason as a binary decision (does it belong to the class or not). To get probabilities, you may use the **softmax** function, which is similar to the maximum, except that it is differentiable. If the output for one class is sufficiently larger than for the others, its softmax output will be close to 1.

$$f_k(\boldsymbol{x}) = \frac{\exp(o_k)}{\sum_{l=1}^{K} \exp(o_l)} \text{ with } o_k = \boldsymbol{w}^{kT} \cdot \boldsymbol{x}$$

Then we choose the class C_k if $f_k(\boldsymbol{x}) = \max_{l \in [1,K]} f_l(\boldsymbol{x})$

3 Training a perceptron

You generally have two possibilities : **online learning**, which roam every instances one by one, and the **batch learning**, that consider all the data-set in a whole. It is a flexible model, that may adapt to a changing problem over time. As a consequence comes the **generic update rule** which means that everytime we have a new training instance, each weight get updated, taking into account the previously established weights $w_j \leftarrow w_j + \Delta w_j$. The update term can be computed as follow, knowing the error function $Error(f(\boldsymbol{x}^i), y^i)$ and setting (arbitrary) the *learning rate* η :

$$\Delta w_j = -\eta \frac{\partial Error(f(\boldsymbol{x}^i), y^i)}{\partial w_j}$$

We will see that in each of the following cases, which present different error functions, this expression simplifies to the same form :

$$\Delta w_j = \eta(y^i - f(\boldsymbol{x}^i))x_j^i$$

Regression Having $Error(f(\boldsymbol{x}^i), y^i) = \frac{1}{2}(y^i - f(\boldsymbol{x}^i))^2 = \frac{1}{2}(y^i - \boldsymbol{w}^T\boldsymbol{x}^i)^2$, that leads to the above-stated formula.

Classification We introduce the sigmoid output $\sigma(u) = \frac{1}{1+\exp(-u)}$ which derives as $\sigma'(u) = u'\sigma(u)(1-\sigma(u))$; f expresses then as $f(\boldsymbol{x}^i) = \sigma(\boldsymbol{w}^T\boldsymbol{x}^i)$. We then use the cross-entropy error, which has the same purpose than the error function established for the regression case

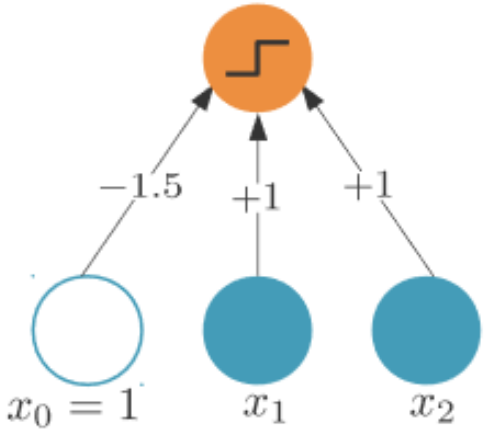
$$Error(f(\boldsymbol{x}^i), y^i) = -y^i \log f(\boldsymbol{x}^i) - (1 - y^i) \log(1 - f(\boldsymbol{x}^i))$$

K classes With the definition of the softmax outputs stated in section 2, we can compute the Cross-entropy error :

$$Error(f(\boldsymbol{x}^i), y^i) = -\sum_{k=1}^{K} y_k^i \log f_k(\boldsymbol{x}^i)$$

We observe that the generic rule for regression is exactly the same than for the classification model. To make it simple : the update rule is equal to the learning rate times the difference between the desired output and the actual output times the input. It comes to mind that if the desired output is equal to the actual output, there will be no change. But if they're not, the predictions will get less accurate.

Still, it has limitations. It can learn the logical AND but not the logical XOR (because data cannot be separated by a hyperplane). This is considered as a very limitated model.



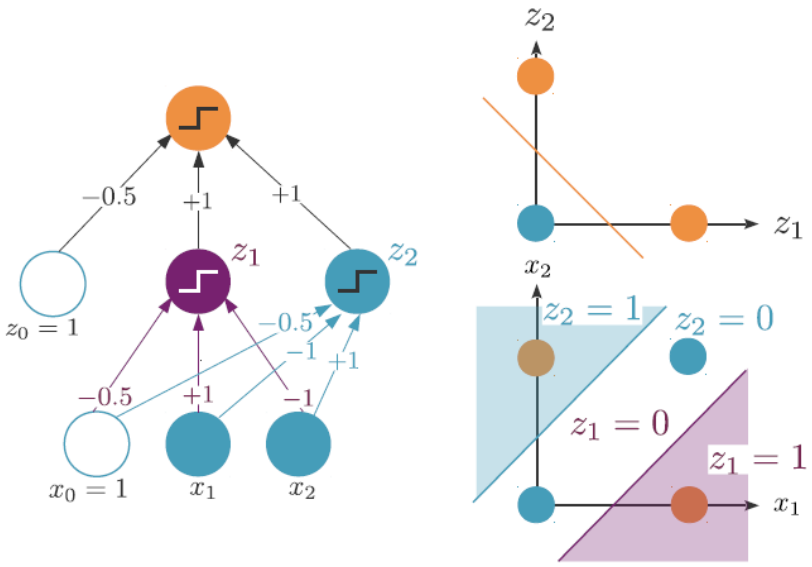
Learning of the logical operator AND, with a single layer perceptron.

To quote Minsky and Papert in 1969 : *"The perceptron has shown itself worthy of study despite (and even because of!) its severe limitations. It has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension to multilayer systems is sterile."*

4 Multilayer Perceptrons

Every node of the hidden layer is the output of a sigmoid perceptron. Then the output of the entire model is a linear combination of the different sigmoid perceptrons, which makes it non-linear in x ! We then came from a linear model and ended up with a non-linear model. As a consequence, the logical XOR can be learned !

Universal Approximation : Any continuous function on a compact subset of $\mathbb{R}^n$ can be approximated to any arbitrary degree of precision by a feed-forward multi-layer perceptron with a single hidden layer containing a finite number of neurons.



Learning of the logical operator XOR.

4.1 Backpropagation

This is one of the most powerful tool in deep-learning. The question of who invented it is still being discussed.

Backpropagation, an abbreviation for "backward propagation of errors", is a common method of training artificial neural networks used in conjunction with an optimization method such as gradient descent. It calculates the gradient of a loss function with respect to all the weights in the network, so that the gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function.

Source : https://en.wikipedia.org/wiki/Backpropagation

This method may be used for binary model or multiclass perceptron. The mathematics hidden behind this process may be summed up in the following lines :

$$z_h = \frac{1}{1 + e^{-\boldsymbol{w}_h^T \cdot \boldsymbol{x}}}$$

$$f(\boldsymbol{x}) = v^T \cdot \boldsymbol{x} = v_0 + \sum_{h=1}^{H} \frac{v_h}{1 + e^{-\boldsymbol{w}_h^T \cdot \boldsymbol{x}}}$$

$$\frac{\partial E}{\partial w_{h_j}} = \frac{\partial E}{\partial f(\boldsymbol{x})} \cdot \frac{\partial f(\boldsymbol{x})}{\partial z_h} \cdot \frac{\partial z_h}{\partial w_{h_j}}$$

Concerning the generic update rule, we obtain, thanks to the previously exposed formulas the next updated weights :

— **Regression :**

$$\Delta w_{jh} = -\eta \frac{\partial E^i}{\partial w_{jh}} = -\eta (y^i - f(\boldsymbol{x}^i)) v_h z_h^i (1 - z_h^i) x_j^i$$

— **Classification :**

$$\Delta w_{jh} = -\eta \frac{\partial E^i}{\partial w_{jh}} = \eta \sum_{i=1}^{n} (y^i - f(\boldsymbol{x}^i)) v_h z_h^i (1 - z_h^i) x_j^i$$

— **K Classes :**

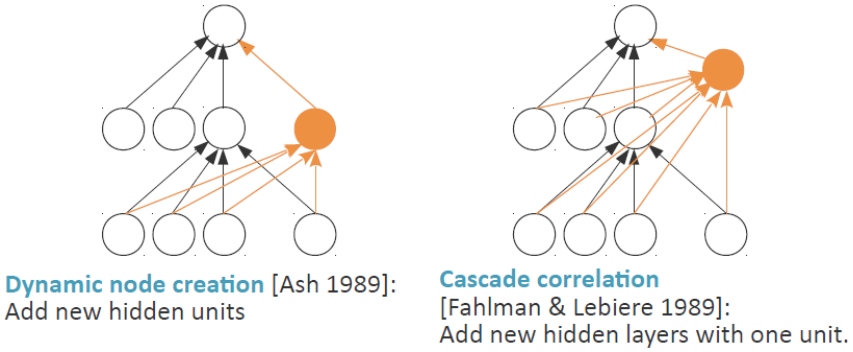
$$\Delta w_{jh} = -\eta \frac{\partial E^i}{\partial w_{jh}} = \eta \sum_{i=1}^{n} \left(\sum_{k=1}^{K} (y_k^i - f_k(\boldsymbol{x}^i)) v_{kh} \right) z_h^i (1 - z_h^i) x_j^i$$

4.2 Deep-Learning

Deep-learning can then be introduced thanks to hidden multilayers perceptrons. The more the hidden layers, the more complex and 'deep' is the learning.

5 Neural Network Magic

The architecture is the following : Start with one hidden layer, stop adding layers when you're overfitting, never use more weights than training samples ! Concerning the **weight sharing**, different units may have connections to different inputs but may be sharing the same weights. Another flexibility remain in modifying the **network size**, in a destructive or constructive way, as much as **overtraining** is.



Growing network until satisfactory error rate is reached.

There exist a lot of optimization algorithms : **Batch Learning** where the weights are updated after a complete pass over the training set, **Mini-Batch Learning** where the weights are updates after a pass over a set of training points of fixed size, Quasi-Newton Methods (Levenberg-Marquardt), Conjugate Gradient Descent, ... But still, **neural newtorks are hard to train**, and they imply a lot of methods to be implemented, such as *preconditionning* that standardizes inputs and targets, initializes weights carefully to avoid saturation (implied by large weigths) and uses local learning rates

5.1 For fun :

Playing with a neural network : http://playground.tensorflow.org/