Bio- and neuro-imaging method
From perceptron to artificial neurons : basic principles

Ronald Phlypo

24th November 2019

Some of the text and most of the figures come from the reference book by Peter Dayan and Laurence F. Abbott [Dayan and Abbott, 2005]. A free electronic version of the book is accessible from the Author's website. See also computer-age statistical inference by Trevor and Hastie.

The goal of this lab session is to implement a perceptron or an artificial neuron, and to understand its basic principles. First of all, we should not think of the artificial neuron (as a building block in an artificial neural network) as an implementation that mimics the functioning of a real physiological neuron. Indeed, the physiological neuron is governed by an exchange of ions through its cell membrane (from the axon to the intersynaptic cleft over the dendrite to the neuron's body) and has a membrane potential that polarises quickly (spike) once a threshold potential is exceeded, typically of −55 to −50mV, see Fig.1. Although we can model the membrane potential with differential equations as such integrating successive spikes — resulting in spiking (artificial) neural networks — we will not study this class of neural networks. Indeed, this class of neurons is only regaining interest because of the limited power consumption, but needs specifically targetted hardware. This is in contrast to models that mimic the activation of neuronal populations rather than individual neurons where classical Von Neumann architectures allow for efficient software implementations.

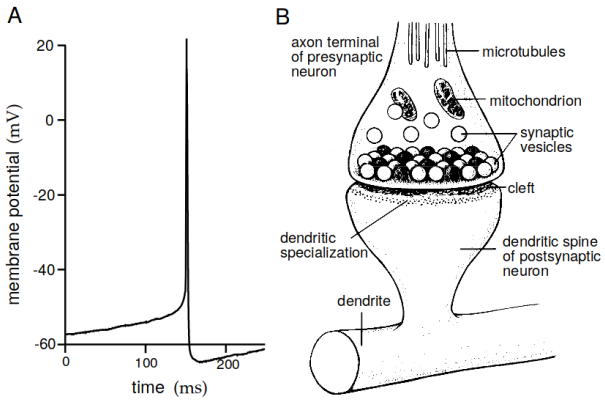

Figure 1: (A) Membrane potential of a firing neuron with a typical threshold value about −55 to −50mV and (B) a caricatural image of the ion exchange in the intersynaptic cleft. Figure from [Dayan and Abbott, 2005]

General notations

In what follows we will denote vectors by lower-case bold face characters, like $\boldsymbol{x}$, matrices as upper-case bold face characters, like $\boldsymbol{X}$, whereas scalars are light face characters, as x. All constructs are defined over the set of real numbers ($\mathbb{R}$).

Artificial neural networks for supervised learning paradigms

You will implement – from scratch – a single *artificial neuron* with its update rule for learning, and later an artificial neural network with multiple *artificial neurons*. We will first study the most general equations, and only then

specialise to the given neuronal model.

Notation

First, let us introduce some notation for a neural network with n neurons indexed by $i \in [\![0, n-1]\!]$:

$$\begin{cases} \text{out}_i & \text{output of } i\text{th neuron} \\ \text{pa}(i) & \text{parents of } i\text{th neuron} \\ \phi_i & \text{non-linearity (activation function) of } i\text{th neuron} \end{cases} .$$

Network topology

We see that the network topology is completely determined by the operator pa. For instance, a classical feed-forward, fully connected, multi-layer neural network layout contains different layers, where all neurons of a given layer project onto all neurons of the next layer. For a simple example of a two-layer neural network, see Figure 2.

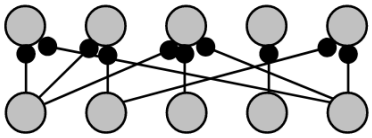

Figure 2: A simple network layout with two layers (not fully connected). The neurons in the lower layer project onto the neurons in the upper layer.

We say a neuron i projects onto a neuron j iff $j \in \text{pa}(i)$. Symbolically this is represented with an oriented arrow (or simply a line if the order is evident) from neuron j to neuron i. We distinguish neural networks on their principal structure from simple feed-forward networks (the neurons can be partially ordered from input to output), over lattice networks (containing lateral inhibition) and directed acyclic graphs (i.e., networks that do not contain cycles) to recurrent networks (networks containing feedback to previous *layers*).

Property Define $\text{pa}(S) \triangleq \bigcup_{i \in S} \text{pa}(i)$ to be the union of the sets of parents of each of the members of the set S. Define $\text{pa}^k(i)$ as the kth order parenthood of i, i.e., $\text{pa}^k(i) \triangleq \underbrace{\text{pa} \circ \text{pa} \circ \ldots \circ \text{pa}}_{\times k}(i)$. If there exists an integer $k \in [\![1, n-1]\!]$ such that $i \in \text{pa}^k(i)$, then the network contains a cycle of order k.

Python tip : a network topology is easily implemented by a list of tuples, e.g., `topology = [(1, 3), (2, 3)]`, where a tuple (i, j) means the jth neuron is a parent of the ith neuron. It is straightforward to get all children projecting on the neuron j together with their weights as a filtering operation, e.g., `[x for (x, y) in topology if y==j]` will return `[1, 2]`. How do you test if a network contains a cycle ? If we have no net input for a neuron we can use `(, 1)` or `(, 2)`, and for neurons that do not project to other neurons (3,). The complete network is then given by `topology = [(, 1), (, 2), (1, 3), (2, 3), (3, )]`.

Network parameters

The net input of the jth neuron of the network is defined by

$$\text{net}_j \triangleq \sum_{\{i | j \in \text{pa}(i)\}} w_i y_i + b_j$$

i.e., a weighted sum of the outputs of the set of neurons projecting onto the jth neuron and a bias term b_j. The neuron takes the net input to the output by mapping it through the non-linearity ϕ_j. Hence, the output of the jth neuron is given by

$$\text{out}_j \triangleq \phi_j\left(\text{net}_j\right) = \phi_j\left(\sum_{\{i | j \in \text{pa}(i)\}} w_{i,j} \text{out}_i + b_j\right) .$$

	$\mathrm{pa}(i)$	net_i	ϕ_i
input	$[\![0, n-1]\!] - \{\text{input}\}$	input feature i	Id
hidden	$[\![0, n-1]\!] - \{\text{input}\}$	$\sum_{\{i \mid j \in \mathrm{pa}(i)\}} w_i y_i + b_j$	(leaky)ReLu, tanh, σ
output	$\emptyset$	$\sum_{\{i \mid j \in \mathrm{pa}(i)\}} w_i y_i + b_j$	tanh, σ (classification), linear (regression)

Table 1: Different types of neurons in a (feed-forward) neural network.

The set of free parameters in the neural network is given by $\boldsymbol{\theta} = \left\{w_{i,j}, b_j \mid j \in [\![0, n-1]\!],\, i \in \left\{k \mid j \in \mathrm{pa}(k)\right\}\right\}$.

Python tip : Together with the network topology we can also store the weights as in a sparse matrix representation, e.g., `weights = [(1, 3, 0.1), (2, 3, -0.3)]`. Each weight is associated with a specific projection as $(i, j, w_{i,j})$, i.e., the weight $w_{i,j}$ associated with the projection of the ith neuron onto one of its parents $j \in \mathrm{pa}(i)$. The network parameters are completely defined by adding a list $\boldsymbol{b}$ containing the bias terms of each of the neurons.

Three types of neurons are considered in a (feed-forward) neural network, see Table 1.

Inference problem

A network takes features $\boldsymbol{x}$ as input, and predicts an output $\hat{\boldsymbol{y}}(\boldsymbol{x})$, where the functional dependence symbolises the network function. Each entry of $\hat{\boldsymbol{y}}$ is a non-linear function of its net input, itself a linear combination of outputs out_i of the neurons i for which $j \in \mathrm{pa}(i)$, etc.. In other words, we compute $\hat{\boldsymbol{y}}$ by propagating the features through the network (feed-forward). As an example, take a network of two input neurons (neurons 1 and 2), two neurons in the hidden layer (3 and 4) and one output neuron (5), we have

$$\begin{cases} \mathrm{out}_3 = \phi_3(\mathrm{net}_3) = \phi_3\left(w_{1,3}x_1 + w_{2,3}x_2 + b_3\right) \\ \mathrm{out}_4 = \phi_4(\mathrm{net}_4) = \phi_4\left(w_{1,4}x_1 + w_{2,4}x_2 + b_4\right) \\ \hat{y} = \phi_5(\mathrm{net}_5) = \phi_5\left(w_{3,5}\mathrm{out}_3 + w_{4,5}\mathrm{out}_4 + b_5\right) \end{cases}$$

Parameter estimation (learning)

Learning is based on an example set provided by *a teacher* $\left\{(\boldsymbol{x}_k, \boldsymbol{y}_k) \mid x_k \in \mathbb{R}^p,\, y_k \in \mathbb{R}^q\right\}$, which justifies the name of *supervised learning*. Given the parameter set $\boldsymbol{\theta}$ and a set of input features $\boldsymbol{x}$, one obtains an estimate of $\boldsymbol{y}$, noted $\hat{\boldsymbol{y}}$, at the output layer of the network. The goal is to get $\hat{\boldsymbol{y}}(\boldsymbol{x})$ as close to $\boldsymbol{y}$ as possible. The distance measure $\delta\left(\boldsymbol{y}, \hat{\boldsymbol{y}}(\boldsymbol{x})\right)$ depends on the problem:

classification For classification, the target output is a binary vector (1 for the correct class, 0 for the other classes), and the output $\hat{\boldsymbol{y}}$ is often associated to a probability, which can be obtained by choosing each output $\hat{y}_k = \frac{\exp(\mathrm{net}_k)}{\sum_{j \in \mathrm{output}} \exp(\mathrm{net}_j)}$ (we can easily check that for all outputs $\hat{y}_k \geq 0$ and $\sum_{k \in \mathrm{output}} \hat{y}_k = 1$). A distance function is then given by the cross-entropy

$$\delta\left(\boldsymbol{y}, \hat{\boldsymbol{y}}(\boldsymbol{x})\right) \sum_{k \in \mathrm{output}} -y_k \log \hat{y}_k(\boldsymbol{x})$$

regression For regression, the target output is a real vector and the output activation functions are linear. The output $\hat{\boldsymbol{y}}(\boldsymbol{x})$ is a prediction of $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$ where $\boldsymbol{f}$ represents a non-linear functional relation between $\boldsymbol{x}$ and $\boldsymbol{y}$. A distance function is often given by the mean squared error (the Euclidean distance)

$$\delta\left(\boldsymbol{y}, \hat{\boldsymbol{y}}(\boldsymbol{x})\right) = \left\|\boldsymbol{y} - \hat{\boldsymbol{y}}(\boldsymbol{x})\right\|_2^2$$

Learning the network parameters

Recall the chain rule for the derivative of a composite function $\frac{\mathrm{d}}{\mathrm{d}x} g(h(x)) = \left[\frac{\mathrm{d}}{\mathrm{d}y} g(y)\right]_{y=h(x)} \frac{\mathrm{d}}{\mathrm{d}x} h(x)$ as well as the total derivative $\mathrm{d}h\left(p(t), q(t)\right) = \frac{\partial}{\partial p} h(p, q) \frac{\partial}{\partial t} p(t)\,\mathrm{d}t + \frac{\partial}{\partial q} h(p, q) \frac{\partial}{\partial t} q(t)\,\mathrm{d}t$. We would like to minimise the distance

function δ with respect to the parameters of the network. Taking the example above with five neurons, and taking the derivative with respect to $w_{1,4}$, for instance, we have

$$\begin{cases} \frac{\mathrm{d}}{\mathrm{d}\hat{y}}\delta\left(y,\hat{y}\right) \\ \frac{\mathrm{d}}{\mathrm{dnet}_5}\hat{y}=\phi'\left(\mathrm{net}_5\right) \\ \frac{\partial}{\partial \mathrm{out}_4}\mathrm{net}_5=w_{4,5} \\ \frac{\mathrm{d}}{\mathrm{dnet}_4}\mathrm{out}_4=\phi'\left(\mathrm{net}_4\right) \\ \frac{\partial}{\partial w_{1,4}}\mathrm{net}_4=x_1 \end{cases} \implies \frac{\partial}{\partial w_{1,4}}\delta=\frac{\mathrm{d}}{\mathrm{d}\hat{y}}\delta\left(y,\hat{y}\right)\ \phi'\left(\mathrm{net}_5\right)\ w_{4,5}\ \phi'\left(\mathrm{net}_4\right)\ x_1$$

In general, we have

$$\begin{cases} \frac{\partial}{\partial\theta_k}\delta=\sum_i \frac{\partial}{\partial\hat{y}_i}\delta\left(\boldsymbol{y},\hat{\boldsymbol{y}}\right)\frac{\partial}{\partial\mathrm{net}_i}\hat{y}_i\sum_{j\in\{m|i\in\mathrm{pa}(m)\}}\frac{\partial}{\partial\mathrm{out}_j}\mathrm{net}_i\frac{\partial}{\partial\theta_k}\mathrm{out}_j \\ \frac{\partial}{\partial\theta_k}\mathrm{out}_j=\frac{\mathrm{d}}{\mathrm{dnet}_j}\mathrm{out}_j\sum_{i\in\{m|j\in\mathrm{pa}(m)\}}\frac{\partial}{\partial\mathrm{out}_i}\mathrm{net}_j\frac{\partial}{\partial\theta_k}\mathrm{out}_i \end{cases}$$

Reorganising terms under the sum gives a backward flow of the error through the network, where in each neuron we compute

$$\mathrm{net}_i^- = \sum_{j\in\mathrm{pa}(i)}\frac{\partial}{\partial\mathrm{out}_i}\mathrm{net}_j\ \phi_j'\left(\mathrm{net}_j\right)=\sum_{j\in\mathrm{pa}(i)} w_{i,j}\ \mathrm{out}_j^-$$

Indeed, this back-propagates the error through the network, starting from the output nodes until we reach the parameter θ_k of interest. The updates of the parameters are given by

$$\boldsymbol{\theta}\left(t+1\right)=\boldsymbol{\theta}\left(t\right)-\eta\nabla_{\boldsymbol{\theta}}\delta$$

where η is a small real constant (take, for instance, $\eta=10^{-2}$).

The network parameters can be learned by batch updates (taking the mean of $\nabla_{\boldsymbol{\theta}}\delta$ over a large subset of samples), or by stochastic updates (one sample at a time).

Exercise 1 (Simple classification task for a single neuron)**.** Create two Gaussians in a two-dimensional space, both with identity covariance and one with mean $(-a,0)$ and the other with mean $(a,0)$. Train a network with two inputs and a single output neuron on 100 samples of your data and test the network on 200 samples. Report the performance on your train and test data as a function of a. Justify the choice of your parameters, your distance function, etc.

Exercise 2 (A regression task)**.** Create a total of 1000 samples from a mixture of five Gaussians in a two-dimensional space, all five with identity covariance matrix and with equal probability. For each Gaussian, the mean will be arbitrarily drawn from a gaussian with covariance a multiple λ of the identity matrix and mean zero. Create a function $y=1-\exp\left(-\left(x_1-x_2\right)^2\right)$ where x_1 and x_2 will be the two-dimensional samples of the (random) mixture of Gaussians. Draw the function values over a grid and propose an appropriate network topology. Learn the network parameters and report the performance as a function of the position of the test point with respect to the sample density, and study the behaviour as a function of λ.

Since often the parameter space of a neural network is gigantic, one needs to regularise the solution. In addition to the objective function one will often find a regularisation term $J(\boldsymbol{W})$ such as $J(\boldsymbol{W})=\|\boldsymbol{W}\|_2^2$ or related. This is similar to Tichonov regularisation when solving the minimum squared error problem. Other options are ridge regression, sparsity inducing penalties, etc.

Self-organising map – or an unsupervised artificial neural network

A self-organising maps gives a low-dimensional representation (a map) of high-dimensional data, regrouping similar samples in close by regions of the map and distinct samples on opposite sides of the map. This is related to the associative (Hebbian) learning in spiking neural networks. Consider a two-dimensional map of $d_1\times d_2$ *pixels*. Any dimension of the incoming sample in the p-dimensional space is fully interconnected with each pixel in the map,

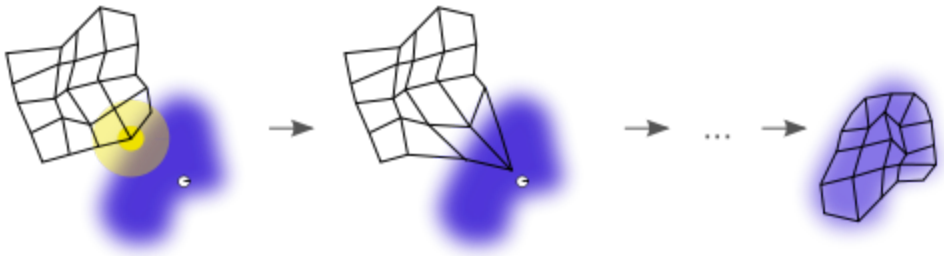

Figure 3: A self-organising map learning the topology of a sample distribution (from Wikipedia, by Dan Stowell aka Mcld, under CCA BY-SA 3.0)

whereas the map has connections between neighbouring pixels (one may consider a simple 4-neighbour or 8-neighbour connectivity pattern, although generally we also consider a fully interconnected map, with connection weights decaying with distance).

Suppose we index the pixels in the map by i, $(0 \leq i \leq d_1 d_2 - 1)$ and the incoming sample dimensions by j, $\left(0 \leq j \leq p-1\right)$, then we have a weight matrix $\boldsymbol{W} \in \mathbb{R}^{p \times d_1 d_2}$ with entries $w_{j,i}$ that represent the weights each dimension of the sample takes in a given pixel. The activation map is then given by

$$\boldsymbol{y} = \boldsymbol{W}^t \boldsymbol{x}$$

If both the columns $\boldsymbol{w}_i$ of $\boldsymbol{W}$ and the samples $\boldsymbol{x}$ are normalised[1] as $\|\boldsymbol{x}\|_2 = 1$ and $\forall i, \|\boldsymbol{w}_i\| = 1$, we have $y_i = 1 - \frac{1}{2}\|\boldsymbol{x} - \boldsymbol{w}_i\|_2^2$. We call the best matching unit (BMU), the unit indexed by i in the map that has smallest Euclidean distance with $\boldsymbol{x}$, in other words

$$\text{BMU} = \arg\max_i \boldsymbol{w}_i^t \boldsymbol{x} \ .$$

Training a self-organising map

These maps are trained using competitive learning, i.e., a pixel that takes the upper-hand in the learning process inhibits its neighbours, resulting in a winner-takes-all configuration. A training example entering the network will activate all neurons that compose the map, with a BMU that will be identified. The goal is to update the weights of the BMU in the direction of the observed sample, whereas we will inhibit the other neurons in the map to fire with the same sample by downdating their weights relative to the distance with the BMU. Taking half the Euclidean distance $\frac{1}{2}\|\boldsymbol{x} - \boldsymbol{w}_k\|$ as the objective function that will be minimised for the best matching unit ($k = i$) and its neighbours, its derivative with respect to $\boldsymbol{w}_k$ is given by $\boldsymbol{w}_k - \boldsymbol{x}$, and hence the update rules are

$$\tilde{\boldsymbol{w}}_k^{(t+1)} = \boldsymbol{w}_k^{(t)} + \theta_t(i,k) \cdot \alpha_t \cdot \left(\boldsymbol{x} - \boldsymbol{w}_k^{(t)}\right)$$

$$\boldsymbol{w}_k^{(t+1)} = \frac{\tilde{\boldsymbol{w}}_k^{(t+1)}}{\left\|\tilde{\boldsymbol{w}}_k^{(t+1)}\right\|}$$

One can see directly that by choosing the step size $\theta_t(i,k) \cdot \alpha_t$ equal to one, that $\boldsymbol{w}_k^{(t+1)} = \boldsymbol{x}$.

The step-size is the product of a distance function with respect to the BMU, θ, and a monotonically decreasing learning rate. An example for the neighbourhood function is $\theta_t(i,k) = \exp\left(-h(t)\|\boldsymbol{c}_i - \boldsymbol{c}_k\|\right)$ for some positive α, $\boldsymbol{c}_i$ and $\boldsymbol{c}_k$ the grid coordinates in the map of the BMU (i) and the neuron for which the weights are updated, and h a monotonically increasing function. Through time, the neighbourhood function is concentrating more and more around the BMU.

Exercise 3 (A self-organising map for learning a classification of wines)**.** Load the wine dataset (see here for an indication on how to load the data with scikit-learn) and classify the wines using a self-organising map. It might be interesting to see how the weight initialisation influences on the result. How to deal with feature scaling? Can you find a biological equivalent function?

[1] This is not a necessity, but is often considered so. If this is not true, the distance must be explicitly computed rather than the linear activation.

References

[Dayan and Abbott, 2005] Dayan, P. and Abbott, L. F. (2005). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Computational Neuroscience. MIT Press, 1st edition.