Università di Pisa

Notes On Computational NeuroScience

Giacomo Lagomarsini

Based on the lectures by professor Claudio Gallicchio

Contents

Chapter 1

Neural Modeling

1.1 Modelling Neurons

The nervous system is constituted of **neurons**. A neuron is a cell that has the capability to communicate with other cells of the same type, through voltage propagation. We can see the neuron as an elementary processing unit of the nervous system, exactly like units in a neural network.

Neurons are typically constituted of three parts that have distinct functions:

- the dendrites can be seen as *input devices*, collecting signals from other neurons
- the soma is the central processing unit of the neuron. Its job is emitting **spikes** of voltage, if the signal collected by the dendrite is sufficiently high.
- the axons are the output devices.

1.1.1 Spike and Spike Trains

The neurons can emit electrical pulses of about 100 mV of amplitude in a time frame of about 1-2 milliseconds. We call these pulses action potentials, or spikes. A chain of action potentials is called a **spike train**.

The spikes in a spike trains are usually separated: we call the minimum time between two spikes the **refractory period**.

1.1.2 Synapse

The connection between an axon of a neuron and the dendrite of another neuron is called **synapse**. We refer to the sending (input) neuron as presynaptic, and to the receiving neuron as postsynaptic.

When an action potential reaches the synapse, it triggers a series of chemical processes, that lead to some molecules to reach the postsynaptic side. At this point, some specialized receptors detect the molecules and open channels that

cause ions (electrically charged atoms) to flow inside the cell from the extracellular liquid.

The ion flux leads to a change in the **membrane potential**, so that the chemical processes are in the end transformed in electrical responses. We will now see how to model these responses.

1.1.3 Simple Model of Spike Response

The membrane potential at the resting state is negatively polarized. The value at rest is typically around $u_{rest} = -65$ mV.

Let i be a neuron, and $u_i(t)$ its potential a time t. If a presynaptic neuron j fires its spike at time $t_0 = 0$, we call the difference of potential at time $t > 0$

$$e_{ij}(t) = u_i(t) - u_{rest}.$$

We call $e_{ij}(t)$ the **Post Synaptic Potential (PSP)** at time t. If $e_{ij}(t) > 0$ we have an excitatory PSP, otherwise, we have an inhibitory PSP.

Now let us consider more presynaptic neurons $j = 1, 2, ...$ We call $t_j^{(f)}$, where $f = 1, 2, ...$ the f-th firing time of neuron j. As long as there are few input spikes, the change of potential can be approximated as the sum of the individual PSPs,

$$u_i(t) = \sum_j \sum_f e_{ij}(t - t_j^{(f)}) + u_{rest}. \tag{1.1}$$

However, when $u_i(t)$ reaches a certain threshold ϑ, the critical value, we see a strongly non-linear behavior: first $u_i(t)$ exhibits a spike, then the potential drops even below the resting value. The second state explains the refractory period of neurons.

Let $\hat{t_i}$ be the last time in which neuron i spiked, i.e.

$$\hat{t_i} = \max\{t_i^{(f)} | t_i^{(f)} < t\}.$$

Because the action potentials have roughly the same form, we can model the trajectory of the spike as a function $\eta(t - \hat{t}_i)$. Putting all together, we obtain a new equation for the neuronal dynamics,

$$u_i(t) = \eta(t - \hat{t}_i) + \sum_j \sum_f e_{ij}(t - t_j^{(f)}) + u_{rest} \tag{1.2}$$

If the threshold ϑ is reached from below, so $u_i(t) = \theta$ and $\frac{d}{dt} u_i(t) > 0$, we observe a spike, $t = t_i^{(f)}$.

One simple way of modeling η is through a square wave and an exponential,

$$\eta(t - \hat{t}_i) = \begin{cases} \frac{1}{\Delta t} & \text{if } 0 < t - \hat{t}_i < \Delta t \\ -\eta_0 \exp\left(-\frac{t - \hat{t}_i}{\tau}\right) & \text{if } \Delta t < t - \hat{t}_i \end{cases}, \tag{1.3}$$

where η_0, τ, $\Delta t > 0$ are some parameters. When $\Delta t \to 0$, the square pulse appears as in Figure 1.1. We can see that if η_0 is sufficiently big, the model

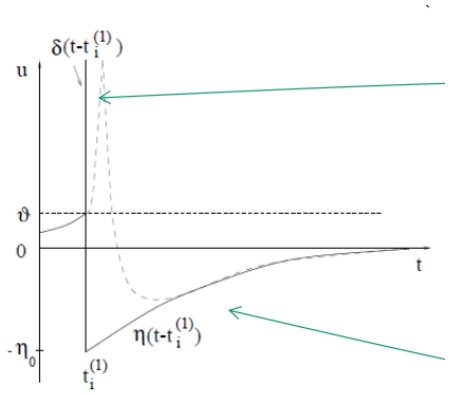


Figure 1.1: Square pulse when $\Delta_t \to 0$

explains why the neuron is not immediately excitable again, and the emission of a second pulse is more difficult.

This model is quite simple but has some limitations.

The first is that it cannot capture non linear interactions between the PSPs.

Moreover, the PSPs are all of the same shape. This in practice is not always true. In particular, when the neuron i is in the moment of depolarization that follows a spike, the PSPs e_{ij} are generally of lower intensity.

Another drawback is that the shape of η depends only on the last firing time $\hat{t}_i$. This implies that we cannot model some behaviors that we found in actual neurons.

Let us think of the following experiment: we give the neuron an input current of value I_1 for times $t < t_0$, and of value I_2 for $t \geq t_0$. This way, substituting in out model (1.2), we obtain

$$u_i(t) = \begin{cases} \eta(t - \hat{t}_i) + I_1 + u_{rest} & \text{if } t < t_0 \\ \eta(t - \hat{t}_i) + I_2 + u_{rest} & \text{if } t > t_0 \end{cases}$$

Being the input potential constant after time t_0, we expect from the model spikes at regular times (or no spikes if the input does not reach the threshold).

However in practice we observe a variety of behaviors (Figure 1.2):

- Taking $I_1 = 0$, $I_2 > 0$, we can observe three behaviors. The first is called **adaptation**: the neuron fires a spike train at time intervals that increase successively, until a steady state when the intervals are constant is reached.
- Another possible behavior when $I_1 = 0$ and $I_2 > 0$ is **fast**, regular spikes. This behavior can in fact be modeled by our equation.
- The third behavior encountered when $I_1 = 0$, $I_2 > 0$ is **bursting**: sequence of fast spikes interleaved by long intervals of no spikes.

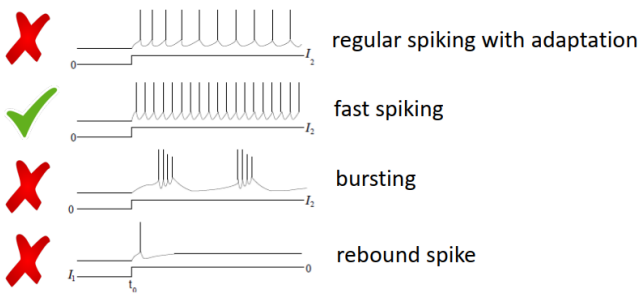


Figure 1.2: Possible behaviors of a neuron when a constant input is given by an external source. The formal pulse can only simulate the fast, periodic spiking, the second case in the figure.

- The fourth behavior can be observed when there is initially a negative potential $I_1 < 0$, and the potential is switched off at time t_o, so $I_2 = 0$. In this case we can observe a **rebound spike**.

The only behavior that can be modeled by the equation (1.2) is the fast spiking neurons. Therefore we need a more complex model of the neuron, that can capture the variety of possible responses.

1.1.4 Rate Codes

The next question we pose is how are the neurons encode the information in order to communicate, and what the spatiotemporal frequencies and patterns of the spike mean.

Rate codes are a way to try answering this question: the assumption is that the information are mostly contained in the **rates** or frequencies of spikes. Formally, given a time interval T, we can define the rate of the spikes

$$\nu = \frac{n_{sp}(T)}{T},$$

where n_{sp} is the number of spikes that a neuron fires in the time interval.

This solution is particularly convenient for encoding real numbers: just two spikes at intervals of $1/\nu$ can encode $\nu \in \mathbb{R}$.

However, the neuron cannot in most situations wait to perform a temporal average.

Other solutions include defining the spike rate as the **density** of spikes from the same neurons in K experiments,

$$\rho(t) = \frac{1}{\Delta t} \frac{n_K(t; t + \Delta t)}{K},$$

or the average population activity of N assumed identical neurons,

$$A(t) = \frac{1}{\Delta t} \frac{n_{act}(t; t + \Delta t)}{N}$$

1.1.5 Spike Codes

Another approach to answer the question is assuming that the timing of single of sequences of spikes as a role: in particular given an initial stimulus, the distance of the **first response** of the neuron can encode the relevance of the stimulus.

The **phase** of a spiking time with respect to a background oscillation can also be considered.

Lastly, similar neurons will respond almost simultaneously to the same stimulus, revealing a pattern of **firing synchrony**.

1.2 Hodgkin-Huxley Model

Defining the model of the spiking neuron, we assumed a certain critical value ϑ that when surpassed, makes the neuron fire a spike.

Defining the threshold is not a trivial task, and sometimes defining even the concept of a threshold seems arbitrary.

We decide then to model the neurons as non-linear dynamical systems. The resting state is an equilibrium of the dynamic, but the **excitability** of neurons means that the system is near to a bifurcation: the equilibrium is stable if the neuron fires one spike and then return to the resting state. It is unstable if the input causes a spike train. In the phase state (Figure 1.8) that translates in an orbit around the equilibrium point. In this section, we will define this dynamical model passing through the chemistry and the biology of neurons. In the following sections, our aim will be to generalize the approach, and step away from the bio-inspired systems, to build a model that is simpler but can nevertheless simulate all the relevant neuronal behaviors.

1.2.1 Ions and Ionic Currents

We already mentioned that changes of voltage in the membrane potential is due to **ionic currents**: the concentration of ions outside and inside the cell are different, so the ions want to move. The main ions that take part in this process are Na^+ (sodium), K^+ (potassium), Ca^{2+} (calcium) and Cl^- (chloride). Outside the cell there is a higher concentration of Na, Cl, and Ca, and a lower concentration of K. Moreover, the cell contains other anions (negatively charged ions), denoted with A^-, that cannot exit the membrane.

We can divide the gates in the membrane in two types:

- **Pumps** actively transport ions inside and outside. Their effect is that the ions concentration inside and outside the cell differs.
- **Channels** are crossed by ions leading to a passive distribution.

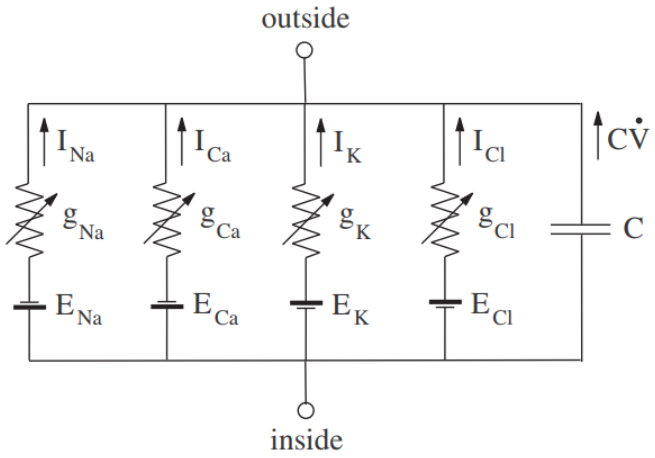


Figure 1.3: Equivalent circuit for the 4 ions.

1.2.2 Nernst Potential

The equilibrium of the potential for every ion channel is called **Nernst potential**.

The formula from the Nernst potential $E_{[\text{ion}]}$ is

$$E_{[\text{ion}]} = \frac{kT}{q} \ln \frac{n_{in}}{n_{out}}$$

where k is the Boltzmann constant, T is the temperature, q is the polarity of the ion (e.g. -1 for Cl^-, $+1$ for K^+) n_{in} is the number of ions inside the cell and n_{out} is the number of ions outside the cell.

Let Δu be the voltage difference between the outside and the inside of the cell $(u_{out} - u_{in})$.

If $\Delta u < E_{[\text{ion}]}$, then the ions flow into the cell, in order to decrease $n_{out} - n_{in}$, and increasing Δu. If $\Delta u > E_{[\text{ion}]}$, the ions flow outside the cell. For this reason the Nerst potential is also called **reversal potential**.

1.2.3 Equivalent Circuit

The electrical properties of the membrane can be seen as an electrical circuit, depicted in Figure The membrane of the cell acts like a capacitor (the symbol with two equal parallel segments).

The Nerst potential is represented by a battery (two parallel segments of different length).

If an input current is injected, it could either leak through the membrane (represented by the rectangles) or charge the capacitor C. We call $g_{[ion]}$ the

conductance of a certain ion channel.

If the membrane potential is u, the flow of ions is determined by $u - E_{[ion]}$, and the ion current is

$$I_{[ion]} = g_{[ion]}(u - E_{[ion]}) \tag{1.4}$$

Consider the ions of elements K, Cl, Na, and Ca. Knowing their concentrations inside and outside the cell, we can compute their Nernst potential. It turns out that

$$E_{\mathrm{K}} < E_{\mathrm{Cl}} < u_{rest} < E_{\mathrm{Na}} < E_{\mathrm{Ca}}.$$

Therefore, we have two inward currents (Na and Ca) and two outward currents (K and Cl).

Let us now analyze the membrane. We model it like a capacitor with capacitance $C = \frac{q}{u}$. Therefore,

$$C\frac{du}{dt} = I_C,$$

where I_C is the current that charges the capacitor.

Let now I be the total current coming from the outside, and $I_{[ion]}$ the current flowing from the inside for each ion. Kirchoff current law tells us that current from the outside is equal to minus the currents from the inside i.e.

$$C\frac{du}{dt} = I - I_{\mathrm{Na}} - I_{\mathrm{Ca}} - I_{\mathrm{K}} - I_{\mathrm{Cl}} \tag{1.5}$$

Or equivalently, using equation (1.4),

$$C\frac{du}{dt} = I - g_{\mathrm{Na}}(u - E_{\mathrm{Na}}) - g_{\mathrm{Ca}}(u - E_{\mathrm{Ca}}) - g_{\mathrm{K}}(u - E_{\mathrm{K}}) - g_{\mathrm{Cl}}(u - E_{\mathrm{Cl}}) \tag{1.6}$$

1.2.4 Gates

The conductance of ions channel is non-Omhic and non constant.

That means that $g_{[ion]}$ can change over time: in practice the channels are controlled by **gates**, that can open or close.

We distinguish between two types of channel:

- We say that the conductance is **persistent** if the channel can only be activated.

  In practice let k be the number of sub-units of the gate. Let n be the gating variable, i.e. the probability that one sub-unit is open. Then the probability that the channel is open is $p = n^k$.

  The value of n depends on the voltage. Depolarization leads to an increase of n, and higher probability that the channel opens.

- We say that the conductance is **transient** if the channel can at some point be inactivated. That means that there is an inactivation gate that can block the channel. Similarly as before, if we have k activation gates,

the probability that they are all open is m^k. Moreover, we assume that the inactivation gate *doesn't block* the channel with probability h.

Putting all together, the channel is open with probability $m^k h$. Again, in case of depolarization m increases, and h decreases. On the other hand, in case of hyperpolarization, m decreases, while h increases.

Being n, m, h variable in time, they are also variables in our system of differential equations. In particular, the dynamic that governs the conductance is

$$\frac{dx}{dt} = \frac{1}{\tau_x(u)}(x_0(u) - x), \tag{1.7}$$

where

- x is any of the gating variables n, m, h
- τ_x is a function of u that is constant in time.
- $x_0(u)$ is the asymptotic value of the conductance with respect to u

We can then modify Equation (1.4) as follows:

$$I_{[ion]} = g_{[ion]} p_{[ion]} (u - E_{[ion]}) \tag{1.8}$$

where now $g_{[ion]}$ is the maximum possible value of conductance, and p is the probability that the channel is open, as described previously.

Putting all together, the dynamic looks as follows: u is governed by the equation

$$C\frac{du}{dt} = I - \sum_{ion} g_{[ion]} p_{[ion]} (u - E_{[ion]}) \tag{1.9}$$

With $p_{[ion]} = n_{[ion]}^{k_{[ion]}}$ in case of persistent conductance, $p_{[ion]} = m_{[ion]}^{k_{[ion]}} \cdot h_{[ion]}$ in case of transient conductance. The gating variables (for each ion) are governed by equation (1.7).

1.2.5 Hodgkin-Huxley Model

The **Hodgkin Huxley model** was formulated by Hodgkin and Huxley in the 50s, based on their studies on the giant squid axon. It is a very accurate representation of the real dynamic of neurons.

With our terminology set, we can simply view it as a particular instance of our general system of equations.

In particular, in this model we have three currents:

- A voltage-gated, persistent conductance, K^+ current with 4 activation gates $\implies I_K = g_K n^4 (u - E_K)$
- A voltage gated, transient conductance, Na^+ current with three activation gates $\implies I_{Na} = g_{Na} m^3 h (u - E_{Na})$

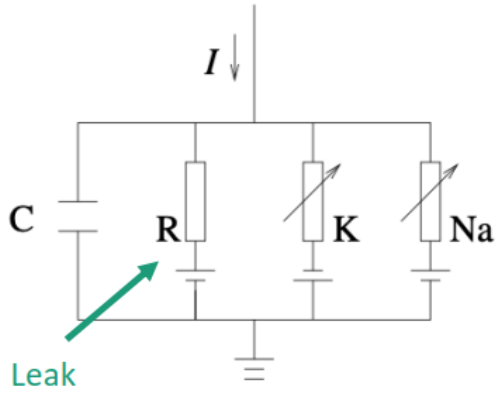


Figure 1.4: Circuit of the Hodgkin-Huxley model. It is a specific case of the general circuit shown in Figure 1.3

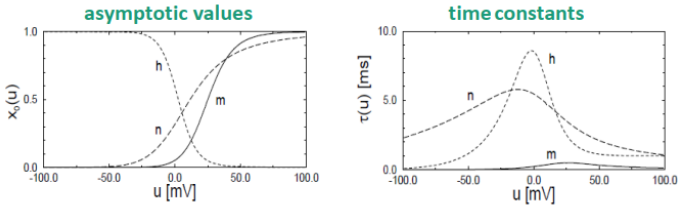


Figure 1.5: Asymptotic value x_0 and time constant τ for the three gates. We can clearly see how the time constant for m is smaller, and therefore the dynamic for m is faster.

- A leak, Ohmic current for the other ions, that does not change in time. We denote it with the subscript L. $I_L = g_L(u - E_L)$.

Putting these specifics in Equations (1.7) and (1.9), we get a system of four differential equations:

$$\begin{cases} C\frac{du}{dt} = I - g_K n^4(u - E_K) - g_{Na} m^3 h(u - E_{Na}) - g_L(u - E_L) \\ \frac{dn}{dt} = \frac{1}{\tau_n(u)}(n_0(u) - n) \\ \frac{dm}{du} = \frac{1}{\tau_m(u)}(m_0(u) - m) \\ \frac{dh}{du} = \frac{1}{\tau_h(u)}(h_0(u) - h) \end{cases} \tag{1.10}$$

In Figure 1.5, we can see how the asymptotic values and the τ change with u. In particular, we see that for every value of u, $\tau_m(u)$ is always small, compared to τ_h and τ_n. If we look at the equation for m in (1.10), this translates in a higher derivative for m. This means that m increases faster than how h decreases. This allows the gate to open. The variable n has a similar sigmoidal shape to

m for the asymptotic, but the dynamic is much slower. From the graphs, we can qualitatively describe what happens in the neuron:

- Because m is faster than n, initially only the channel for Na^+ opens, and the ions flow inside the membrane (recall that $u_{rest} > E_{\text{Na}}$). Because of their polarity, these ions lead to an increase of u.
- When the potential is high, the Na^+ gate finally closes, due to the vanishing of h. Meanwhile, the gate for K^+ has opened: this implies that K^+ can now flow outside the cell.
- This leads to a repolarization, i.e. a decrease in the value of u, and the K^+ channel closes.

The first point leads to an increasing in the membrane potential. If the potential increases enough, we observe the spike we described in the previuos section. The depolarization due to K explains the subsequent refractory period.

The Hodgkin-Huxley model is very accurate, and it takes also in account the biophysical mechanisms that are responsible of the dynamic. However, being a system of 4 equations, it is very difficult to visualize and computationally intensive to simulate. For this reason, in the next section we aim to approximate it with a 2-dimensional model, i.e. a system of two differential equations in two variables.

1.2.6 Two Dimensional Models

We now want to approximate the Hodgkin-Huxley model with a system of only two variables. We make these key observations, looking at Figure 1.5:

1. The dynamic of m is very fast when compared to n and h. We can then think of m as having an instantaneous dynamic, and replace m at time t with its asymptotic value $m_0(u) = m_0(u(t))$.
2. n and $1-h$ have comparable asymptotic values. We can model them with a unique time-dependent variable w.

   We can take $w = n/a = b - h$ for some appropriate constants a, b.

Putting these assumption in the Equations (1.10), we get

$$\begin{cases} C\dot{u} = I - g_{\text{Na}}(m_0(u))^3(b-w)(u-E_{\text{Na}}) - g_{\text{K}}(w/a)^4(u-E_{\text{K}}) - g_R(u-E_R) \\ \dot{w} = (w_0(u) - w)/\tau_w \end{cases} \tag{1.11}$$

Let now $R = g_L^{-1}$, $\tau = RC$. We can write equivalently:

$$\begin{cases} \dot{u} = \frac{1}{\tau}F(u,w) + RI \\ \dot{w} = \frac{1}{\tau_w}G(u,w). \end{cases} \tag{1.12}$$

Equation (1.12) can be instantiated to equation (1.11) with the right choice of functions $F(u, w)$ and $G(u, w)$, but it is more general. In particular we can further simplify the assumptions.

For example, let's see the **Morris-Lecar model**:

$$\begin{cases} \dot{u} = I - g_1 m_0(u)(u-1) - g_2 w(u - E_2) - g_L(u - E_L) \\ \dot{w} = \frac{1}{\tau(u)}(w_0(u) - w) \end{cases} \tag{1.13}$$

In this model, we can see that we did the following simplifications:

- The Ernst potential of one ion is normalized to be 1.
- We dropped the exponents 4 and 3 in the probabilities of gates opening.
- $h = (b - w)$ is omitted.

From (1.12) we can also derive the **FitzHugh-Nagumo** model,

$$\begin{cases} \dot{u} = u - \frac{1}{3}u^3 - w + I \\ \dot{w} = \epsilon(b_0 + b_1 u - w), \end{cases} \tag{1.14}$$

by choosing $F(u, w) = u - \frac{1}{3}u^3 - w$ and $G(u, w) = b_0 + b_1 - w$, plus some normalization to get dimensionless variables.

1.3 Dynamical Systems Basics

1.3.1 Definition of Dynamical Systems

A dynamical system is the mathematical description of a system that involves time.

It is composed of a **state**, i.e. the actual state of the variables that describe the system, and a **dynamic**, that describes how the system evolves over time. The set of possible states is often called **state space**.

We can distinguish two cases, based on how we view the time in which the system evolves:

- If we are considering **continuous time**, the dynamical system is described by a system of **differential equations**:

$$\frac{d\mathbf{x}}{dt} = F(\mathbf{x}, a) \tag{1.15}$$

- If the time is **discrete**, the system is ruled by **iterated maps**:

$$\mathbf{x}_{t+1} = G(\mathbf{x}_t, a) \tag{1.16}$$

In the equations (1.15) and (1.16), $\mathbf{x}$ is the vector of variables of the system, that represent the state, while a is all the additional parameters that, joint to the functions F or G, govern the dynamic.

We call **orbit** the sequence of states in the state space that the system exhibits during its evolution.

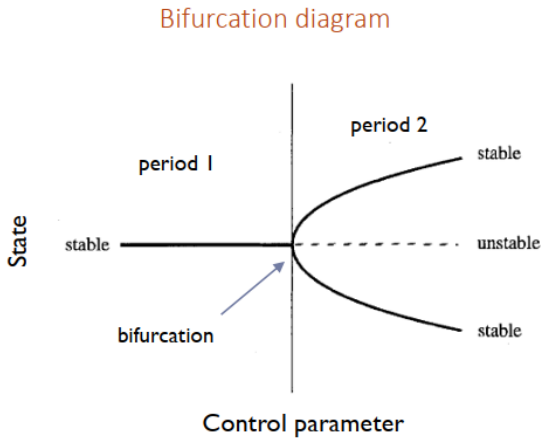


Figure 1.6: An example of bifurcation

1.3.2 Equilibria

The most important concept when talking about dynamical systems is the **equilibrium**:

Definition 1. We say that a system is at equilibrium state $\mathbf{x}^*$ when the system in $\mathbf{x}^*$ is at rest, i.e. $\mathbf{x}(t) = \mathbf{x}^*$ for all t. This is equivalent to

$$\frac{d\mathbf{x}}{dt}|_{\mathbf{x}^*} = F(x^*, a) = 0 \tag{1.17}$$

or, in the case of discrete systems,

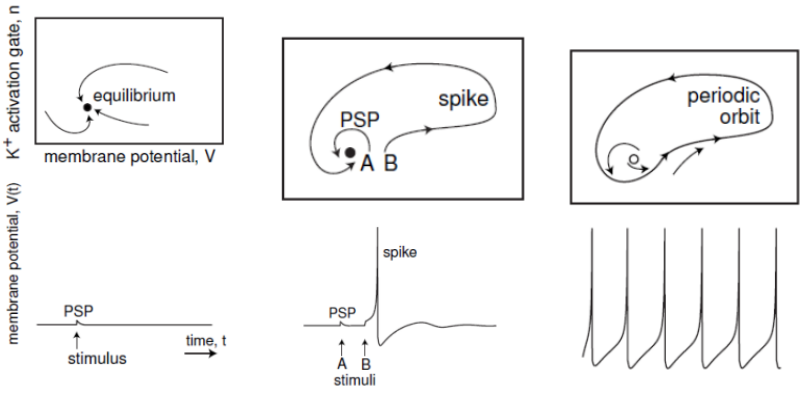
$$G(\mathbf{x}^*, a) = \mathbf{x}^* \tag{1.18}$$

More generally, we define a **limit cycle** as an equilibrium trajectory. This means that if the dynamics starts in the limit cycle, it continues staying in the cycle for all time-steps t.

When we find an equilibrium point, we can study how the trajectory changes for small perturbations around the point. We can observe *stable* and *unstable* behaviors. We say that a certain region $\mathcal{A}$ of the space is an **attractor** if all trajectories converge to said region, from a larger region $\mathcal{B} \supseteq \mathcal{A}$. We call $\mathcal{B}$ the **basin of attraction** of $\mathcal{A}$.

Given a certain equilibrium point, we call the **repeller** the set of points in the neighborhood of the equilibrium that move away from it.

Changing the parameters a of the system can lead to changes in the dynamic. We call **bifurcation** the qualitative change in the phase portrait of the system. Figure (1.6) shows a paradigmatic example of bifurcation in one dimension: when the control parameter is less than 0, the fixed point $x = 0$ is stable. When



(a) Stable state (b) Single spike (c) Periodic spike

Figure 1.7: Three possible behavior. In 1.7a, the input is so small that the resting equilibrium point is stable. In 1.7b, we stimulate the neuron with a larget input: the neuron spikes, and in the phase potrait we can see a sbustantial deviation from the equilibrium. Eventually though, the state returns to the orignal equilibrium. In 1.7c, the input is so strong that the system is attracted to a new equilibrium, the stable limit cycle. The neuron will continue to emit spikes. This is possible because the state is near to a bifurcation.

the parameter becomes greater than 0, the fixed point becomes unstable, and a positive (negative) small input leads the dynamic of the system to go on the positive (negative) half-line.

1.3.3 Neurons as Dynamical Systems

We have already seen that neurons can be modelled as dynamical systems. With the theory we have developed in the previous section we can do some other considerations: we have that the variables of our systems are the membrane potential and the excitation/recovery variables. In particular in the Hodgkin-Huxley model, we have a total of four variables.

With phase portraits we can visualize the behavior of the system in two dimension, taking two variables as coordinates. Doing this we can see whether, for example, an equilibrium point is stable or unstable.

The rich set of possible behaviors of a neuron is caused by the fact that the neuron is close to a bifurcation, from resting to spike activity. In particular, the dynamic is governed by transient inputs in the current. As we show in Figure 1.7, there are equilibrium points and limit cycles that can coexist.

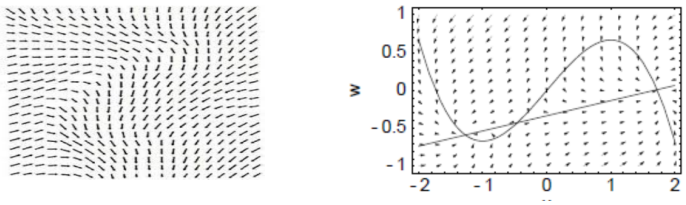


Figure 1.8: Phase plane of a dynamical system, in the right figure we can see the flow represented by small arrows, in the left figure the nullclines of the two variables,

1.3.4 Stability of Fixed Points

Let us now restrict to two-dimensional dynamical systems. As already said, we can visualize the dynamics in a phase plane. Figure 1.8 shows an example of phase plane. In the figures are also depicted the two **nullclines** of the system, i.e. the zones where the flow is stationary along one of the two direction: In case of the u nullcline, $\dot{u} = 0$ and the arrows are vertical, viceversa, in case of the w nullcline, $\dot{w} = 0$ and the arrows are horizontal. The fixed points are those situated in the intersection of the two nullclines:

$$\begin{pmatrix} \dot{u} \\ \dot{w} \end{pmatrix} = \begin{pmatrix} F(u,w) \\ G(u,w) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Now we see how we can identify whether a fixed point is stable of instable. The idea is that the stability depends only on how the dynamical system behave in a neighborhood of the fixed point: visually, an equilibrium point is stable if in the phase plane the arrows around it point towards it.

Mathematically, in the one-dimensional case, a point is stable if $\frac{dF}{du} < 0$, and it is instable if $\frac{dF}{du} > 0$. This can be extended in two (or more) dimensions by using a linearization of the dynamic: around the fixed point $x_* = (u_*, w_*)$, the dynamic is well approximated by

$$\frac{d}{dt}(x - x_*) = \begin{pmatrix} F_u & F_w \\ G_u & G_w \end{pmatrix} (x - x_*) \tag{1.19}$$

Where $F_u = \frac{\partial F}{\partial u}(u_*, w_*)$ etc.

We can find if the equilibrium is stable by looking at the eigenvalues of the matrix.

- If the eigenvalues are both real and negative we have a stable node, i.e. the equilibrium is stable and the dynamic near the equilibrium converges without obscillating
- If the eigenvalues are both real and positive, we have an unstable node.

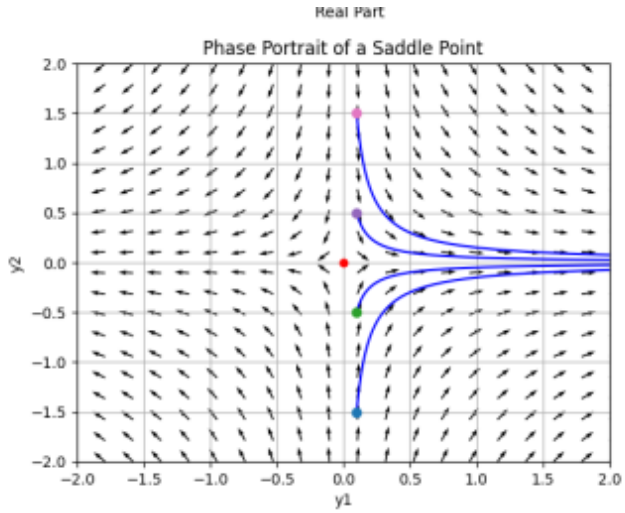


Figure 1.9: The saddle point in the figure separates the plan in 4 areas: if the dynamic starts in one of the quadrants, it cannot end in another quadrant.

- If the eigenvalues are both real, one positive and one negative we have a saddle point: along one direction the dynamic converges to the equilibrium, but along another direction the dynamic diverges.
- If the eigenvalues are complex conjugate with negative (positive) real parts, we have a stable (unstable) node: the dynamic converges (diverges) obscillating around the equilibrium.
- If the eigenvalues are pure imaginary, we have a center: around the equlibrium the trajectories look like ellypses. This holds for the linearized system, but in general it tells us nothing about the original system.

In particular, saddle points separate the phase plan in different areas where the dynamic is qualitatively different (Fig. 1.9).

1.3.5 Bifurcation

Let us consider again Figure 1.8, which represents the nullclines of the FiztHugh-Nagumo model. In this state, the system has 3 equilibrium states,

1.4 Dynamics of Spiking Neuron Models

1.4.1 Integrate and Fire and variants

In this section, we will see some simplified models that can be used to efficiently and accurately model spiking neurons. We already seen some **formal threshold**

models, i.e. models where the spike is an event that always occurs when the potential reaches a certain threshold ϑ. In this case, the spikes are always the same shape, and the aim is to model the threshold dynamics and predict the firing times of the spikes.

The Hodgkin-Huxley model is a very accurate model of what actually happens in the neuron, but it is very complicated, difficult and slow to simulate.

The **integrate and fire** model can be defined by a circuit only having a capacitor. In this case, the equation for the potential becomes

$$C\frac{du}{dt} = I(t) \Rightarrow \frac{du}{dt} = \frac{I(t)}{C} \tag{1.20}$$

where C is the capacity constant. Again, here, the spikes are defined as formal events, happening when $u^{(t_f)} = \vartheta$.

After the spike, we reset the potential at the rest state u_{rest}, often assumed to be 0:

$$t^{(f)} \text{ s.t. } u(t^{(f)}) = \theta \tag{1.21}$$

$$\lim_{t \to t^{(f)+}} = u_{rest} \tag{1.22}$$

This dynamic can also be written in integral form, and solved exsplicitly if $I(t) = I_0$ is constant. Let us assume that the last spike was at time $t^{(1)}$, therefore $u(t^{(1)}) = u_{rest} = 0$. Then we can write

$$u(t) = \int_{t^{(1)}}^{t} \frac{I_0}{C} ds = \frac{I_0}{C}(t - t^{(1)}). \tag{1.23}$$

We also impose a refractory period where the integration is suspended for a certain period of time Δ.

Leaky integrate and fire add the membrane conductance as a constant leakage term, i.e. we have to add to the integrate and fire equation (1.20) a term $-I_R$. This term can be rewritten as $I_R = u(t)/R$, where R is a certain constant, that tells us the conductance of the membrane. This is realistic for small fluctuations around u_{rest}. From the circuit we can therefore write

$$I_C = I(t) - I_R \Rightarrow C\frac{du(t)}{dt} = I(t) - \frac{u(t)}{R} \tag{1.24}$$

$$\Rightarrow RC\frac{du}{dt} = -u(t) + RI(t) \tag{1.25}$$

$$\Rightarrow \tau_m \frac{du}{dt} = -u(t) + RI(t) \tag{1.26}$$

where now $\tau_m = RC$. Equation (1.26) is the equation of **leaky integrate-and-fire** model. The equations (1.21) and (1.22) hold as well to determine the fire time and the reset condition.

1.4.2 Izhikevic Model

Izhikevic model is a simple 2 dimensional model that is surprisingly good at simulating neurocomputational features.

We have two variables:

- A fast voltage variable u
- A slow recovery variable w that is equivalent to both the activation of K / the inactivation of Na in the Hodgkin Model

The system iss a generalization of the integrate and fire model, that models the dynamic of u as a quadratic differential equation

$$\begin{cases} \dot{u} = I + p_2(u) - w \\ \dot{w} = a(bu - w) \\ \textbf{reset condition:} \quad \text{if } u \geq \theta \text{ then } u \leftarrow c, w \leftarrow w + d \end{cases} \tag{1.27}$$

Here p_2 is a generic polynomial of second grade in the variable u, and $a, b, c, d, I \in \mathbb{R}$ are the parameters.

In particular, a standard form that is found by fitting the dynamics of cortical neurons is

$$\begin{cases} \dot{u} = 0.04u^2 + 5u + 140 - w + I \\ \dot{w} = a(bu - w) \\ \textbf{reset condition:} \quad \text{if } u \geq 30 \text{ then } u \leftarrow c, w \leftarrow w + d \end{cases} \tag{1.28}$$

This system of differential equation can model a vast amount of behaviors of neurons: in particular, integrator and resonators are modelled, by changing the parameters . Notice how the dynamic has no formal threshold: the third condition is used only to **reset** the dynamic to a given state. Let see what are the meaning of the parameters:

- a controls the time scale of the recovery variable. Because the dynamic of w is much slower than the dynamic of u, a is usually a small number. A typical value is 0.02
- b controls how sentitive the recovery variable w is to obscillations in the membrane potential u. A typical value is 0.2
- c and d control the after-spike reset values of u and w respectively. c is typically set to -65.

Despite its simplicity, this model is able to simulate almost all the computational features of the Hodgkin Huxley model, using only two variables, u and w, instead of four.

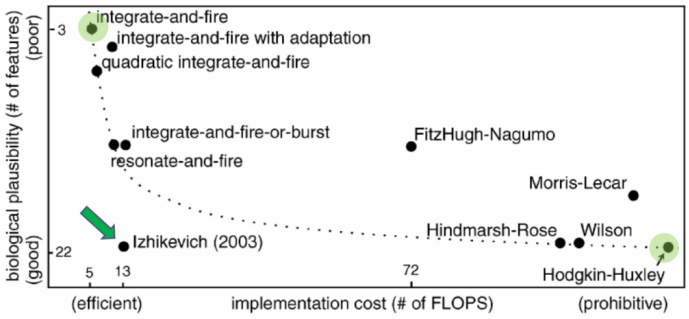


Figure 1.10: Comparation of neuronal models, in terms of computational features the models are able to simulate w.r.t. the cost of implementation of the model in floating points operations

1.5 Networks of Spiking Neurons

We have seen how to model the dynamic of a single neuron. We now pass to a network of connected neuron. This situation is seen in the brain in neocortical circuit: those are layered recurrent circuits, which have neurons that lie in 6 layers. Each layer has feed-forward connections to the neurons in a higher layer, and recurrent connection in-between layers. Top-down connections are also present,

A **network of spiking neurons** is a pool of connected neuronal units, in which each neuron receives in input the post-synaptic current generated by each one of its presynaptic neurons, and an additional external input I_{ext}.

The strength of the synaptic connection between a presynaptic neuron j and a postsynaptic neuron i is modeled by a **synaptic weight** w_{ij}.

We can distinguish between excitatory synapses, when $w_{ij} > 0$, and inhibitory synapses, when $w_{ij} < 0$.

1.5.1 Patterns of Connectivity

Fully Connected

The simplest network we can think of is a network of fully-connected homogeneous neurons, where all neurons are identical, and receive the same input. Moreover, the strength of each synapse is equal: $w_{ij} = \frac{J_0}{N}$.

Sparsely Connected

We can also think of sparse network. where either there is a fixed coupling probability for two neurons to be coupled, e.g. 10%, or we fix a number of postsynaptic partners for each neuron, i.e. each neuron has exactly C postsynaptic neurons, chosen randomly.

Two Populations

One other important example is having two populations of neurons:

- a pool of inhibitory neurons (around $\approx 20\%$ of the total), modeled by fast spiking neurons
- a pool of excitatory neurons (around $\approx 80\%$), modeled by regular spiking neurons.

1.5.2 Synaptic Plasticity

The weights w_{ij} are not constant: they can change over time because of learning or memory, or other mechanisms.

This possibility to change is called synaptic plasticity. We distinguish between **long term potentiation** (LTP), where we have a persistent change in synaptic transmission efficiency (the weights grow), and long term depression (LTD), where a stimulation pattern leads to the weights to shrink.

This is similar to learning in traditional Machine Learning settings: the weights are just the parameters of the net, and can be optimized to improve the performance of the model in a given task (classification, regression).

The typically used learning rule in this setting is Hebbian Learning. The changes in synaptic transmission is due to the correlation of pre and post synaptic activity, being them potentials or firing rates.

Hebb's Postulate

when an axon of cell A is near enough to excite cell B or repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both the cells such that A's efficiency, as one of the cells firing B, is increased

With a simple experiment, we can observe long-term potentiation that is due to synaptic plasticity and it is coherent with Hebb's postulate (Figure 1.11):

- First we apply a small pulse to a presynaptic fiber, and measure the postsynaptic response. This response will be typically very mild
- Then we apply a stronger stimulation. The input would lead to a postsynaptic potential spike
- Then we inject again another small pulse, equal to the first one, and measure the postynaptic response: the response it will be stronger.

1.5.3 Spike-Time Dependent Plasticity

The timing of pre and post-synaptic spikes plays a major role in determining the change in the synaptic strength. Let $t_j^{(f)}$ be the time of last pre-synaptic spike, and $t_i^{(f)}$ the time of the last post-synaptic spike. Then the change Δw_{ij} is a

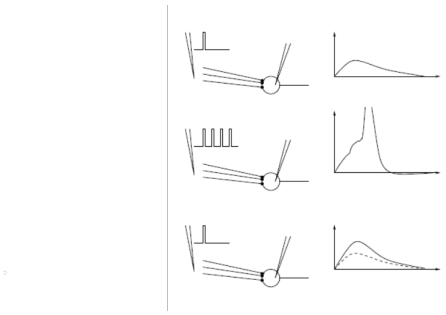


Figure 1.11: Experiment that demonstrates Hebb's postulate

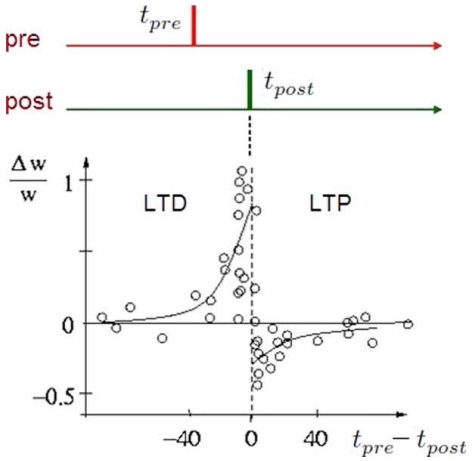


Figure 1.12: Spike time dependent plasticity: if $t_i^{(f)} - t_j^{(f)} > 0$, we have long term potentiation (LTP), otherwise we have long term depression (LTD)

function of $t_j^{(f)} - t_i^{(f)}$: if this difference is close to 0 and positive, the change will be very positive, otherwise if it is close to 0 and negative, this means that the post-synaptic spike is not caused by the pre-synaptic spike, thus Δw_{ij} will be negative, and the synapse will weaken. This means that the change in synaptic weights is determined by the timing of both spikes.

We can model the change in w_{ij} at time t as a dynamical system. First, we define the neural response function of a neuron

$$\rho(t) = \sum_f \delta(t - t^{(f)}), \tag{1.29}$$

where $t^{(f)}$ is the f-th firing time. Here δ is the Dirac's delta, defined by

$$\delta(t) = \begin{cases} 1 & \text{if } t = 0 \\ 0 & \text{otherwise} \end{cases}$$

If i and j are two neurons, we refer to their neural response functions as ρ_i and ρ_j. Both the pre and post synaptic spikes produce a hebbian term and a non-hebbian term. The non-hebbian term can be simply modeled as $\rho_i(t) \cdot a_1$, where a_1 is a constant.

The hebbian terms are modeled as a correlation (convolution) between the functions $a_2(s)$, that represents the amount of chnge when a presynaptic spike follows a postsynaptic spike after a delay s and the neural response at time $t-s$. Putting together, we have

$$\begin{aligned}\frac{d}{dt}w_{ij}(t) = a_0 + \rho_j(t)\left(a_1^{pre} + \int_0^\infty a_2^{pre,post}(s)\rho_i(t-s)ds\right) \\ + \rho_i(t)\left(a_1^{post} + \int_0^\infty a_2^{post,pre}(s)\rho_j(t-s)ds\right)\end{aligned} \tag{1.30}$$

The first term in the right side of (1.30) is a activity independent term. The second term is the effect of pre-synaptic spikes: a_1^{pre} is the non-hebbian term, the integral is the hebbian term. The terms are both multiplied by the neural reponse function of the pre-synaptic neuron. The third term is the effect of post-synaptic spikes.

Let us focus on the two hebbian terms, and in particular on the meaning of $a_2^{pre,post}$ and $a_2^{post,pre}$:

- $a_2^{post,pre}$ gives the amount of change when a pre-synaptic spike is followed by a post synaptic spike fire with delay s. Therefore, we want it to be high when s is close to 0, and to vanish when s is high.

  This way, if for a small s the neural response function of j is equal to 1, i.e. at time $t-s$ close to t (the time of the post-synaptic spike), there was a pre-synaptic spike, the hebbian term will be big.

- $a_2^{pre,post}$ is the amount of change when a pre-synaptic spike follows a post-synaptic one after a delay s. We want this to be very negative when s is small, and to vanish when s is large.

We can also directly define a function on all $\mathbb{R}$,

$$W(s) = \begin{cases} a_2^{post,pre}(-s) & \text{if } s < 0 \\ a_2^{pre,post}(s) & \text{if } s > 0. \end{cases}$$

In particular, a typical choiche is an exponential form

$$W(s) = \begin{cases} c_+ \exp(-s/\tau_1) & \text{if } s < 0 \\ c_- \exp(-s/\tau_2) & \text{if } s > 0. \end{cases}$$

where c_+ is a positive constant and c_- is negative. In this case we can also only consider the effect of the last spike, because the function decays rapidly if $|s|$ is large, and obtain again Figure 1.12

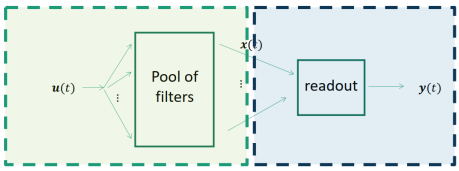


Figure 1.13: A LSM is composed by a pool of spiking neurons, that define a dynamical system that acts as a filter, and of a readout layer that can adapt to the given task.

1.5.4 Liquid State Machines

A **Liquid State Machine** is a particular type of recurrent network of spiking neurons. The network is implemented by a pool of randomly connected neurons. Our objective is, given an input sequence $u(t)$, predicting an output sequence $y(t)$. To do so, we use a large quantity of neurons as dynamical **filters**. A filter is a function that maps sequences into other sequences.

The analogy is with throwing stones in a liquid, and understand from the dynamic of the waves that are created, what is the patterns in stones thrown.

In practice, the readout is implemented as a recurrent network, that transform the input $u(t)$ at state t in a **liquid state** $x(t)$.

Being recurrent, the liquid memorizes the previous state $x(t-1)$, and also uses it as an input for the next step. In spiking neurons implementations, this means that the network sees which neurons spiked at time $x(t-1)$, and passes the spikes in the postsynaptic neurons of the neurons that have spiked.

The other component of a LSM is the **readout**. The readout has no memory and can be adapted to transform the liquid state $(x(1), ..., x(t))$ into the outputs $(y(1), ..., y(t))$.

Mathematically, the liquid is an input driven dynamical system $x(t) = F^L(u(\cdot))$, and the readout is a function $y(t) = F^R(x(t))$. Notice how F^R takes in input only the last $x(t)$, while the liquid could take at each time step the whole sequence. Temporal filters through the liquid have two major properties:

- **Time invariant**: a temporal shift of the input determines a temporal shift of the output of the filters of the same amount. This means that if $u'(t) = u(t-k)$ for each t, then $x'(t) = F^L(u'(t)) = F^L(u(t-k)) = x(t-k)$.

- **Fading memory**: the output of the filters for an input sequence u_1 can be approximated by the output of the filters for another input sequence u_2 , if u_2 approximates well u_1 over a long (recent) time interval. For example, if from a certain point t_0, $u_1(t) = u_2(t)$ for each $t > t_0$, then $x_1(t)$ will eventually approximate $x_2(t)$.

  This means that the outputs will depend only on the most recent inouts signal (suffixes). The state space has a suffix-based markovian organization.

There is a theoretical guarantee on the LSM performances: suppose that the liquid has the point-wise separation property.

Definition 2. Let u, v be two sequences, and let t_0 be a time-step such that $u(t) \neq v(t)$ for some $t < t_0$. Then a family of filters has the **point-wise separation property** if it exists a filter F^L such that

$$F^L(u(t_0)) \neq F^L(v(t_0))$$

Suppose also that the readout has the **universal approximation property**, i.e. it can approximate arbitrarily well any continuous function on a compact domain. artificial NNs with one hidden layers have this property. Then it holds an approximation property for the LSM:

Theorem 1. *A Liquid State Machine can implement any time-invariant temporal filter with fading memory, provided that*

- *the liquid satisfies the point-wise separation property;*
- *the readout satisfies the universal approximation property.*

To implement a LSM with the techniques of the last sections, we can use any interconnected network of spiking models of the neurons. Moreover the codes that go in the liquid state can be both spike codes, i.e. binary values that tell us if a given neuron has spiked, or temporal codes, like the firing rate of the neurons. We will explore more in detail firing rate models in the next chapter.

Chapter 2

Unsupervised and Representation Learning

2.1 Hebbian Learning for Firing Rate Models

2.1.1 Synaptic Plasticity for Firing Rate Models

Now, we are moving towards a new paradigm: before we focused on the timing of the spikes. Now our aim is expressing the probability/ density of spikes averaged in time.

We discuss groups of neurons organized in connected networks. Each neuron i receives inputs $\{\alpha_j(t)\}_j$ from its presynaptic neighbors j, and emits some output α_i. The transmissions of signals are modulated by weights w_{ij}, like in artificial neural networks. These weights represent how much signal the synaptic channels let go through. Moreover, the neuron can also receive an external input $I^{ext}(t)$, that acts as a bias term.

Neurons are typically organized in layers: there are layers closer to input (lower layers) and higher layers, further from the input. The layers can be connected by

- Feed forward connections from lower to higher layers.
- Recurrent or Feedback connection, connecting neurons in the same layer or higher to lower layers.

How does the brain learn the connections that modulates the transmission of electrical inputs?

The weights in the synapses can be changed changed through synaptic plasticity:

- In the **long term potentiation** regime, we have a increase in the synaptic efficacy $\Rightarrow$ the weight w_{ij} increases.

- In **long term depression**, we observe a decrease of synaptic efficacy ⇒ the weight w_{ij} decreases.

In particular, these two behaviors are determined by the correlation of pre and post synaptic activities in a given neuron, i.e. if the activity is strong both in neuron j and in neuron i, the weight w_{ij} will increase in magnitude (LTP), otherwise it decreases (LTD). We already encountered learning paradigm, called **Hebbian Learning**, for timing based models. Now we want to extend the rules for firing rate models.

2.1.2 Learning Paradigms Taxonomy

Hebbian Learning and synaptic plasticity fall in the broader **Unsupervised Learning** paradigm: in this learning technique, the data are not provided with any label: the network adjusts its weights only based on synaptic plasticity.

In contrast, **Supervised Learning** techniques, such as classification or regression, assume a teacher that gives ground truth labels associated with the data, and the network can adapt its weights until the desired behaviors (e.g. classifying correctly images of cats or dogs) emerges.

Another learning paradigm is **Reinforcement Learning** (RL), where the feedback is given in terms of a reward function, that tells the network how good it is doing.

2.1.3 Firing Rate Models

The type of models on which we want to apply Hebbian learning are network of firing rates neurons. We already seen these kind of models in the first section: these models are simpler than models that consider the magnitude of action potential over time as a continuous variable, but cannot account temporal aspects of the dynamic.

Recall the **neural response function**

$$\rho(t) = \sum_i \delta(t - t_i), \tag{2.1}$$

where t_i are the times of spikes. δ is a function that is equal to 1 only when its input is 0, i.e. when t is equal to t_i.

We can approximate the neural response function as its mean over a time interval Δt: we call the **firing rate**

$$r(t) = \frac{1}{\Delta t} \int_t^{t+\Delta t} \langle \rho(\tau) \rangle d\tau. \tag{2.2}$$

The total synaptic input of a neuron is then determined by the total firing rate of presynaptic neurons. The input in turn determines the postsynaptic firing rate of the neuron.

We denote with I_s the total synaptic input, with $\mathbf{u}(t)$ the **presynaptic firing rate** and with $v(t)$ the **postsynaptic firing rate**. If $(1, ..., N_u)$ are the presynaptic neurons, then $\mathbf{u}(t) = (u_1(t), ..., u_{N_u}(t))$.

Let $i \in (1, ..., N_u)$ be a presynaptic neuron. A spike from i generates a current in the postsynaptic neuron equal to $w_i K_s(t)$, where w_i is a weight, and K_s is the synaptic kernel, which is a certain function of the time t.

Therefore, if we assume that the effect of spikes in a single synapse sum linearly, the total contribution of i to the potential until time t is equal to

$$w_i \sum_{t_j<t} K_s(t - t_j) = w_i \int_{-\infty}^{t} K_s(t - \tau)\rho_i(\tau)d\tau \tag{2.3}$$

Then the total input is (again we assume linearity, this time for inputs for different atoms)

$$I_s = \sum_{1}^{N_u} w_i \int_{-\infty}^{t} K_s(t - \tau)\rho_i(\tau)d\tau \approx \sum_{1}^{N_u} w_i \int_{-\infty}^{t} K_s(t - \tau)u_i(\tau)d\tau \tag{2.4}$$

Now we make an assumption in the kernel form: we take $K_s(t) = \frac{1}{\tau_s} e^{-\frac{t}{\tau_s}}$. Substituting in (2.4), we can solve the integral and obtain

$$\tau_s \frac{dI_s}{dt} = -I_s + \sum_{i=1}^{N_u} w_i u_i = -I_s + \mathbf{w}^T \mathbf{u}$$

Then, if the output is not time dependent, we take $v(t)$ as a non-linearity applied to I_s : $v = F(I_s)$.

In conclusion we have the two equations of the firing rate:

$$\tau_s \frac{dI_s}{dt} = -I_s + \mathbf{w}^T \mathbf{u} \tag{2.5}$$

$$v = F(I_s) \tag{2.6}$$

If the output is also time dependent, we have another equation:

$$\tau_s \frac{dI_s}{dt} = -I_s + \mathbf{w}^T \mathbf{u} \tag{2.7}$$

$$\tau_r \frac{dv}{dt} = -v + F(I_s) \tag{2.8}$$

Typically, we assume either a much faster output ($\tau_r >> \tau_s$)dynamic, and obtain

$$\tau_r \frac{dv}{dt} = -v + f(\mathbf{w}^T \mathbf{u}) \tag{2.9}$$

. If we assume $\tau_r << \tau_s$, we obtain (2.5) and (2.6).

2.1.4 Basic Hebbian Rule

Let us now focus on a postsynaptic neuron, with N_u presynaptic inputs. From (2.9), taking a linear activation function f, we get

$$\tau_r \frac{dv}{dt} = -v + \mathbf{w}^T \mathbf{u}.$$

Here, we are assuming again an instantaneous output firing rate.

We want to learn our weights w by an equation

$$\tau_w \frac{d\mathbf{w}}{dt} = G(v, \mathbf{u}, \mathbf{w})$$

The basic Hebbian rule makes the weights strengthen if there are simultaneous presynaptic and postsynaptic activities:

$$\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u} \tag{2.10}$$

The averages Hebb rule takes the average over the input examples:

$$\tau_w \frac{dw}{dt} = \langle v\mathbf{u} \rangle \tag{2.11}$$

This actually translates in a covariance rule: the weight change is given by $\langle v\mathbf{u} \rangle = \langle \mathbf{u}\mathbf{u}^T\mathbf{w} \rangle = \langle \mathbf{u}\mathbf{u}^T \rangle w$, therefore defining the **correlation** matrix $Q = \langle \mathbf{u}\mathbf{u}^T \rangle$, where $Q_{i,j} = \langle u_i u_j \rangle$, we get

$$\tau_w \frac{d\mathbf{w}}{dt} = Q\mathbf{w}. \tag{2.12}$$

This form of the hebbian rule is called **correlation-based plasticity rule** The main issue with this rule is its instability: the learning rule does not promote competition among the synapses, and the weigths magnitude grows to infinity: in fact, by deriving the square euclidean norm of w we can see

$$\frac{d\|\mathbf{w}\|^2}{dt} = 2\mathbf{w}\frac{d\mathbf{w}}{dt} = 2w\frac{v\mathbf{u}}{\tau_w} = \frac{2v^2}{\tau_w}$$

Because the last term is always positive, the norm has an unbounded growth.

2.1.5 Other Hebbian Rules

Covariance Rule

Basic hebbian rule, as it is, implements only the long term potentiation regime. There is no way to decrease the magnitude of weights. One way to do this could be to impose a threshold, either on pre-synaptic or on post.synaptic activity:

$$\tau_w \frac{d\mathbf{w}}{dt} = (v - \theta_v)\mathbf{u}, \tag{2.13}$$

or

$$\tau_w \frac{d\mathbf{w}}{dt} = v(u - \theta_u), \tag{2.14}$$

where θ_v and θ_u are some positive thresholds.

One common way to choose these threshold is taking the average over the input patterns: $\theta_v = \langle v \rangle_u = \langle \mathbf{w}^T \mathbf{u} \rangle_u$, $\theta_u = \langle \mathbf{u} \rangle_u$. With these choice, both the rules (2.13) and (2.14) are equivalent to an online version of the **covariance rule**

$$\tau_w \frac{d\mathbf{w}}{dt} = \langle (\mathbf{u} - \langle \mathbf{u} \rangle)\mathbf{u}^T \rangle \mathbf{w} = C\mathbf{w} \tag{2.15}$$

where C is the **covariance matrix**, defined by $C = \langle (\mathbf{u} - \langle \mathbf{u} \rangle)(\mathbf{u} - \langle \mathbf{u} \rangle)^T \rangle = \langle \mathbf{u}\mathbf{u}^T \rangle - \langle \mathbf{u} \rangle \langle \mathbf{u} \rangle^T = \langle (\mathbf{u} - \langle \mathbf{u} \rangle)\mathbf{u}^T \rangle$.

Proof. The rules are identical:

$$\begin{aligned}(v - \langle v \rangle)\mathbf{u} &= (v - \langle \mathbf{w}^T \mathbf{u} \rangle)\mathbf{u} \\ &= (v - \mathbf{w}^T \langle \mathbf{u} \rangle)\mathbf{u} \\ &= v\mathbf{u} - \mathbf{w}^T \mathbf{u} \langle \mathbf{u} \rangle \\ &= v(\mathbf{u} - \langle \mathbf{u} \rangle)\end{aligned}$$

.

Moreover, we show that the rule with θ_v is equal to the rule with C. We have shown $C = \langle (\mathbf{u} - \langle \mathbf{u} \rangle)\mathbf{u}^T \rangle$.

Multiplying by $\mathbf{w}$, we get

$$\begin{aligned}C\mathbf{w} &= \langle (\mathbf{u} - \langle \mathbf{u} \rangle)\mathbf{u}^T \rangle \mathbf{w} \\ &= \langle (\mathbf{u} - \langle \mathbf{u} \rangle)v \rangle \\ &= \langle v(\mathbf{u} - \langle \mathbf{u} \rangle) \rangle\end{aligned}$$

□

Despite implementing also the long term depression, this rule is also instable: In fact, omitting the τ_w term,

$$\begin{aligned}\frac{1}{2}\frac{d\|w\|^2}{dt} = \mathbf{w}^T C \mathbf{w} &= \mathbf{w}^T (\langle \mathbf{u}\mathbf{u}^T \rangle \mathbf{w} - \langle \mathbf{u} \rangle \langle \mathbf{u} \rangle^T \mathbf{w}) \\ &= \mathbf{w}^T \langle \mathbf{u}\mathbf{u}^T \rangle \mathbf{w} - \mathbf{w}^T \langle \mathbf{u} \rangle \langle \mathbf{u} \rangle^T \mathbf{w} \\ &= \langle (\mathbf{w}^T \mathbf{u})(\mathbf{w}^T \mathbf{u}) \rangle - \langle \mathbf{w}^T \mathbf{u} \rangle \langle \mathbf{w}^T \mathbf{u} \rangle \\ &= \langle v^2 \rangle - \langle v \rangle^2\end{aligned}$$

The last term is the variance of v, that is always positive if v, except the trivial case where v is constant. Therefore, once again, the norm of $\mathbf{w}$ grows indefinitely.

The covariance rule does not enforce competitivity either: all the components of $\mathbf{w}$ can grow independently to infinity.

BCM Rule

BCM rule, from Bienenstock, Cooper, and Munro, requires both presynaptic and postsynaptic activity for the neuron to enter long term potentiation. This is modeled by multiplying the standard hebb's rule, that gives the presynaptic term, by a thresholded postsynaptic value:

$$\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u}(v - \theta_v). \tag{2.16}$$

This rule is stable if θ_v grows more rapidly than the output firing rate. For example we use

$$\tau_\theta \frac{d\theta_v}{dt} = v^2 - \theta_v \tag{2.17}$$

With this rule, we also enforce competition: if a synapse strengthen, v grows, and then the threshold also grows, making weak synapses' LTP less probable.

Subtractive Normalization

Subtractive normalization rule imposes a constraint on the sum of the weights, that cannot change. The sum of the components of a vector can be also written as $\mathbf{w}^T\mathbf{n}$, where $\mathbf{n} = (1, ..., 1)^T$. The rule is

$$\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u} - \frac{v(\mathbf{u}^T\mathbf{n})\mathbf{n}}{N_u} \tag{2.18}$$

We can see that the sum of the weights of $\mathbf{w}$ does not change by taking its derivative:

$$\frac{d\mathbf{w}^T\mathbf{n}}{dt} = v\mathbf{u}^T\mathbf{n} - \frac{v(\mathbf{u}^T\mathbf{n})\mathbf{n}^T\mathbf{n}}{N_u} = v\mathbf{u}^T\mathbf{n} - \frac{v(\mathbf{u}^T\mathbf{n})N_u}{N_u} = 0.$$

This rule is clearly competitive, but it becomes unstable without constraints that prevent the weights from being negative.

Oja Rule

One way to prevent the weights to go too much negative is imposing a constraint on the sum of the squared weights. Oja rule imposes a dynamic constraint, through a hyperparameter α:

$$\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u} - \alpha v^2\mathbf{w} \tag{2.19}$$

By deriving the squared norm of $\mathbf{w}$, we get

$$\frac{d\|\mathbf{w}\|^2}{dt} = v\mathbf{u}^T\mathbf{w} - \alpha v^2\|\mathbf{w}\|^2 = v^2(1 - \alpha\|w\|^2).$$

The steady state, obtained by putting the derivative to 0, is $\|\mathbf{w}\|^2 = 1/\alpha$.

2.1.6 Hebbian Learning as an Unsupervised Technique

We have seen how the dynamic of the hebbian rule $\mathbf{w}$ can be described by a multiplication by the correlation matrix Q.

Let's study the eigenvectors of Q: since Q is symmetric and positive semidefinite, its eigenvector form an orthonormal basis of $\mathbb{R}^{N_u}$, $\mathcal{B} = \{\nu_1, \nu_2, ..., \nu_{N_u}\}$ and its eigenvalues are all real and non negative. Let the eigenvalues be $\lambda_1 \geq \lambda_2 \geq \lambda_{N_u}$.

We can write $\mathbf{w}$ in coordinates of $\mathcal{B}$, as

$$w(t) = \sum_{i=1}^{N_u} c_i(t)\nu_i(t),$$

with $c_i(t) = \mathbf{w}(t)^T\nu_i$ In this new basis the system is a linear diagonal system, and can be therefore decomposed in N_u independent differential equation. The explicit solution of the system $dc_i(t)/dt = \lambda_i c_i(t)/\tau_w$ is

$$c_i(t) = c_i(0)\exp(\lambda_i/\tau_w).$$

Therefore, putting the N_u equations together, we can write

$$\mathbf{w}(t) = \sum_{i=1}^{N_u} \exp\left(\frac{\lambda_i t}{\tau_w}\right)(\mathbf{w}(0)^T\nu_i)\nu_i. \tag{2.20}$$

Assuming that all the eigenvalues are different, the exponential factors are dominated by the largest eigenvalue λ_1. This means that $\mathbf{w}(t)$ converges, as t goes to infinity, to a multiple of ν_1, called the **principal eigenvector** of Q [1].

If $\mathbf{w}$ converges to $\mathbf{w}(\infty)$, after normalization, the response v converges to the projection of the input vector onto the principal eigenvector,of Q, $\mathbf{u}^T\nu_1$. Thus, the Hebbian learning is analogous to performing a Principal Component Analysis on the input dataset $\{\mathbf{u}^k\}_k$.

2.2 Modeling Memory: Hopfield Networks

2.2.1 Associative Memory

Now, we will present a model that describe the function of associative memory in the human brain. Associative memory is the mechanism that allows humans to complete partially incomplete information. For example, if we see half of an orange hidden behind a coffee mug, we are still able to recognize it as an orange. More generally, we can say that a noisy input triggers the recall of the most similar prototype that we have in our mind (Figure 2.1).

A simple recall algorithm is searching through all the possible protoypes and find the one that is most similar to our input x. If the prototypes are $p^1, ..., p^M$, then the most similar prototype is p^α such that $|x - p^\alpha| \leq |x - p^\mu|$ for each μ.

[1] Hebbian Learning is equivalent to applying the power method to the matrix Q

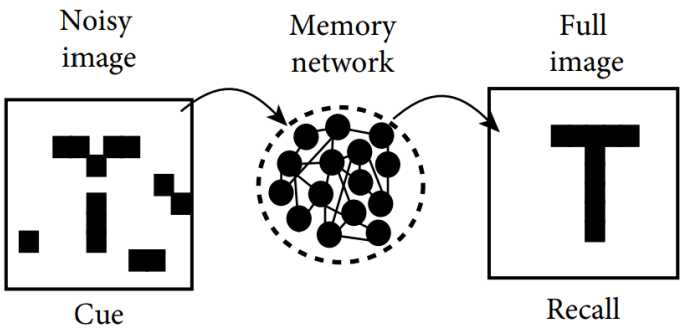


Figure 2.1: A noisy input recalls the most similar prototype. In this case some pixel in this binary image are switched.

Our aim is replacing this search algorithm with interacting neurons, in a distributed and automatic fashion.

This models are part of our discussion on dynamical networks: time plays a role in this model by means of feedbacks:

- local feedbacks are applied to a single neuron in the network;
- global feedbacks are applied in one or more layers.

2.2.2 Lyapunov Method For Stability

In associative memories, and in general in dynamical network models, we are interested in the concept of stability of a fixed point. One way to prove that a fixed point is stable is via a **Lyapunov function**. Some stronger forms of stability are the following:

Definition 3. An equilibrium state $\bar{x}$ of a dynamical system is said to be **uniformly stable** if, for any positive constant ϵ, there exists another positive costant $\delta = \delta(\epsilon)$ such that if

$$\|x(0) - \bar{x}\| < \delta,$$

then

$$\|x(t) - \bar{x}\| < \epsilon \text{ for all } t > 0.$$

This means that if we want the dynamic to stay in a certain neighborhood of the equilibrium point, we just need a initial condition near enough to the equilibrium.

Another notion is the convergence of the dynamic to the stable point:

Definition 4. An equilibrium state $\bar{x}$ is said to be **convergent** if there exists a positive constant $\delta > 0$ such that if

$$\|x(0) - \bar{x}\| < \delta,$$

then $x(t) \to \bar{x}$ as $t \to \infty$.

Combining the two notions, we have **asymptotic stability**.

Moreover we say that $\bar{x}$ is **globally asymptotically stable** if it is uniformly stable and *all* the trajectories converge to $\bar{x}$ as $t \to \infty$, i.e. is convergent with $\delta = +\infty$.

In this last case, the system cannot have any other equilibrium points.

The stability of a fixed point can be proved through the use of the direct method of Lyapunov. This method involves finding a so called **Lyapunov function** for the system.

Definition 5. Let $\bar{x}$ be an equilibrium point, and $U = U(\bar{x})$ a neighborhood of $\bar{x}$. A Lyapunov function $V(x) : U \to \mathbb{R}$ is a scalar function such that

1. $V(x)$ is continuously differentiable in U.
2. $V(\bar{x}) = 0$.
3. $V(x) > 0$ for all $x \in U \setminus \bar{x}$.

Right now, we did not say anything about the stability of $\bar{x}$. If we find a Lyapunov function that has negative derivatives, then the point is stable:

Theorem 2. *If it exists a Lyapunov function for $\bar{x}$ such that*

$$\frac{d}{dt}V(x) \leq 0 \text{ for all } x \in U \setminus \bar{x},$$

then $\bar{x}$ is stable

If the derivatives are strictly negative, then the equilibrium is asymptotically stable:

Theorem 3. *If it exists a Lyapunov function for $\bar{x}$ such that*

$$\frac{d}{dt}V(x) < 0 \text{ for all } x \in U \setminus \bar{x},$$

then $\bar{x}$ is asymptotically stable

Intuitively, around the equilibrium point the Lyapunov function is convex, and if $V(x(0)) = c$ a trajectory can only move towards an area where $V(x(t)) \leq c$ (Theorem 2) or $V(x(t)) < c$ (Theorem 3). In the latter case, the trajectory can only converge towards $\bar{x}$, because the derivative is strictly negative.

In the former case, it is possible that a limit cycle $\mathcal{C}$, where $V(x) = c'$ for all $c' \in \mathcal{C}$, exists around the equilibrium point.

2.2.3 Neurodynamic Models

A neurodynamic model is a network of neuron, being it artificial or bio-inspired, that is viewed as a dynamical system. The main characteristics we want from these models are:

- A large number of **degrees of freedom**: the network is comprised by a large number of independent parameters
- **Non-linearity**: we want a non-linear component. In artificial neural networks, this is done by interleaving affine transfoermations with element-wise non linearities.
- **Dissipation**: the state-space volume converges onto a manifold of lower dimensionality as time goes on.

The model we analyze is an additive model. We can see it as an electrical circuit, where we have N inputs $x_i(t)$, each one passing through resistances, that are then summed.

An external input current I_j is also added, to act as a bias term. Therefore, the total current flowing towards the input node of the non-linearity is

$$v_j = \sum_{i=1}^{N} w_{ji} x_i(t) + I_j \tag{2.21}$$

The potential v_j is then passed through a non-linearity ϕ. Therefore, the output is $x_j(t) = \phi(v_j(t))$.

The current also flows through another resistance R_j and a capacitor C_j. The total current flowing away from the node is given by

$$I = C_j \frac{dv_j}{dt} + \frac{v_j}{R_j}. \tag{2.22}$$

Applying Kirchoff's Current Law, obtain

$$C_j \frac{dv_j}{dt} + \frac{v_j}{R_j} = \sum_{i=1}^{N} w_{ji} x_i(t) + I_j, \tag{2.23}$$

or, equivalently

$$C_j \frac{dv_j}{dt} = -\frac{v_j}{R_j} + \sum_{i=1}^{N} w_{ji} x_i(t) + I_j. \tag{2.24}$$

Putting the equations together for all j, we can describe the dynamic of the model with a system of first order, non-lineare differential equations. The dynamic is non linear because of the presence of $x_i(t) = \phi_i(v_i(t)$, with ϕ_i that are, in general, non linear transformations, altough the identity is also used, making the dynamic linear. An example of additive models are Hopfield Networks.

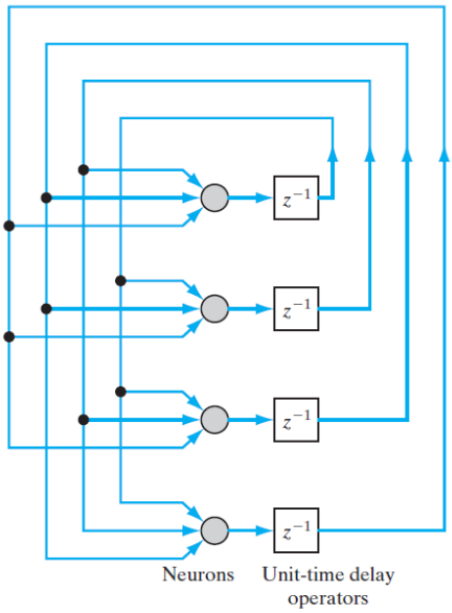


Figure 2.2: Circuit of a Hopfield Network

2.2.4 Continuous Hopfield Networks

A Hopfield Network is a recurrent additive models that incorporates time delays. In figure 2.2, we can see the typical architecture of a continuous time Hopfield Network: The outputs at a certain time step are fed again to the network as inputs. z^{-1} represents the time-delay operator, but we consider it as a general non-linearity ϕ.

Applying Equation (2.24) for this architecture, gives us the additive model

$$C_j \frac{dv_j}{dt} = -\frac{v_j}{R_j} + \sum_{i=1}^{N} w_{ji}\phi(v_i(t)) + I_j. \quad \forall\, j = 1, ..., N \tag{2.25}$$

We make further assumptions on the structure of the model:

- We assume that the weight matrix is symmetric, i.e. $w_{ij} = w_{ij}$.
- Typically, we assume no self-feedback, i.e. $w_{jj} = 0$.

For this equations, we can find a Lyapunov function that guarantees that the dynamic converges to an equilibrium: the energy function of the network, defined by

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ji}x_i x_j + \sum_{j=1}^{N}\frac{1}{R_j}\int_0^{x_j} \phi^{-1}(x)dx - \sum_{j=1}^{N} I_j x_j. \tag{2.26}$$

One can prove that the derivative of E is negative along the trajectories of $\mathbf{v}$, therefore, by Theorem 3, the equilibrium point is stable. Thus, as t grows, the dynamic approaches the equilibrium point.

2.2.5 Discrete Hopfield Networks

The discrete version of the Hopfield Network operates similarly to the McCullocs Perceptron, by outputting binary values. The non-linearity is the *sign* function, defined by

$$\phi(x) = sign(x) = \begin{cases} +1 & x > 0 \\ -1 & x < 0. \end{cases}$$

The $+1$ state indicates that the neuron is on, the -1 means that the neuron is switched off. This non-linearity can also be approximated by the *tanh* function, defined by

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

Here, a is a parameter that describes the *gain* of one neuron. When a is infinitely large, this function exactly converges to the *sign* function. Moreover, in discrete Hopfield Networks we typically assume no self-loops and $I_j = 0$. Therefore, the energy function becomes

$$\begin{aligned} E &= -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ji}x_i x_j + \sum_{j=1}^{N}\frac{1}{R_j}\int_0^{x_j} \tanh_a^{-1}(x)(x)dx \\ &\xrightarrow{a\to\infty} -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ji}x_i x_j. \end{aligned}$$

We can use the discrete Hopfield network as a content-addressable memory (in contrast to typical memories that work by location or name). The goal of the network is memorizing M patterns and retrieving the correct pattern, given a noisy version of it.

2.2.6 Retrieval Dynamic in Hopfield Networks

Now we explain how to retrieve the patterns in a Hopfield Network. Let's suppose that the networks has memorized M patterns $p^1, ..., p^M$. We call these patterns **fundamental memories** In the discrete Hopfield Network, we have a discrete dynamical system

$$x_j(t+1) = sign\left(\sum_{i=1}^{N} w_{ji}x_i(t)\right). \tag{2.27}$$

We actually want to update the current state asynchronously. at each iteration we update just one component j: Therefore, the pseudocode for the network is:

Algorithm 1 Update of the discrete Hopfield Network state

Require: $x(0)$, the initial pattern
  $x \leftarrow x(0)$
  while convergence is not reached **do**
    π = random permutation of $\{1, ..., N\}$
    for $j \in \pi$ **do**
      $x_j \leftarrow sign\left(\sum_{i=1}^{N} w_{ji} x_i\right)$
    end for
  end while

To measure the goodness of the retrieved pattern, we compute for each pattern an **overlap function**

$$m^{\mu}(x(t)) = \frac{1}{N} \sum_{i=1}^{N} x_i(t) p_i^{\mu}$$

Being the vectors all binary, the maximum value of the overlap function is 1, when $x(t) = p^{\mu}$, and the minimum value is -1, when $x(t) = -p^{\mu}$, i.e. all the values of $x(t)$ are the opposite of the values in p^{μ}.

2.2.7 Storage in Hopfield Networks

Since now, we assumed that the weights of the model were the optimal weights that allow us to perform an effective retrieval. But how do we learn these weights? Or equivalently, how do we successfully store the fundamental memories? The weights are set to the correlation between each fundamental memory, We can do this by an outer product:

$$W = \frac{1}{N} \sum_{\mu=1}^{M} p^{\mu} p^{\mu T} \tag{2.28}$$

Therefore, $w_{ij} = w_{ji} = \frac{1}{N} \sum_{\mu} p_i^{\mu} p_j^{\mu}$. Because in the diagonal we have always a correlation of 1 for each pattern, we subtract MI. The final formula for W is

$$W = \frac{1}{N} \left(\sum_{\mu=1}^{M} p^{\mu} p^{\mu T} - MI \right). \tag{2.29}$$

Then, in the retrieval phase, if we have a new example p^{probe}, we take $x(0) = p^{probe}$, and apply Algorithm 1. The retrieval will eventually end when $x(t)$ is attracted to a stable fixed point. There are more than one fixed point, that corresponds to a local minimum of the energy function, each one corresponding to one particular pattern, but there are also spurious fixed points, corresponding to mixtures of two or more fundamental memories.

Chapter 3

Neural Networks for Sequence Modeling

3.1 MultiLayer Perceptron

In the previous chapter, we have seen various bio inspired networks, that worked in a mostly unsupervised way, like Hopfield Networks. Now we abstract the concept and step away from the biology realm. We introduce the so-called **artificial neural networks**. Usually, linear transformations are not enough to represent or approximate a lot of functions. Moreover, composing linear/affine transformations just results in another linear transformation, so stacking multiple layers of linearities does not make sense.

Definition 6. A **multi layer perceptron** (MLP) with d hidden layers is an interleaved composition of $d+1$ affine transformation and $d+1$ (element-wise) non-linearities. The first d transformations are called hidden layers, while the last one is the output layer.

For example, a MLP of just one hidden layer and linear output layer is composed of

- One affine transformation $z = W_1 x + b_1$ that transforms $x \in \mathbb{R}^{N_x}$ in $h \in \mathbb{R}^{N_h}$. Thus, $W_1 \in \mathbb{R}^{N_h \times N_x}$, $b_1 \in \mathbb{R}^{N_h}$.
- One non-linearity $\Phi = (\phi, \ldots, \phi) : \mathbb{R}^{N_h} \to \mathbb{R}^{N_h}$, such that $h = \Phi(z) = (\phi(z_1), ..., \phi(z_{N_h})$
- One last linear tranformation $o = W_2 h + b_2$ with $o \in \mathbb{R}^{N_o}$, $W_2 \in \mathbb{R}^{N_h \times N_o}$, $b_2 \in \mathbb{R}^{N_o}$.
- The last transformation is just the identity in this case. For different tasks, there are plenty of possible output transformations: for classification tasks, the most used is typically the *softmax* activation function

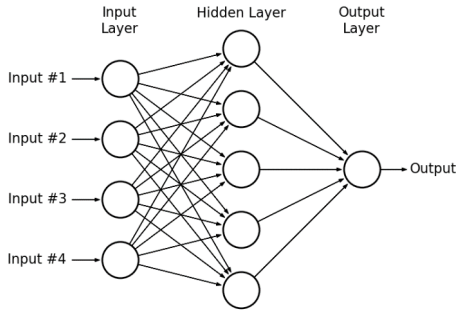


Figure 3.1: MLP with one hidden layer of 5 units and a single unit output layer.

MLPs are typically visualized as connected graphs (Figure 3.1), where each arrow represent one single weight $W_{jk}^{(i)}$, and the nodes represent the results of applying the non-linearity to the scalar product: $h_j^{(i+1)} = \phi^{(i)}\left(W^{(i)}[j,:] \cdot h^{(i)}\right)$, where we have considered the jth is the component of the $(i+1)$th hidden layer. One can prove that with enough hidden states (so with N_h big enough), this function can approximate any function $f : \mathbb{R}^{N_x} \to \mathbb{R}^{N_o}$ (provided some regularity assumptions on f), so in a sense a single hidden layer MLP is a *universal approximator*.

3.1.1 Activation functions

The non linearities are typically called **activation functions**. The most common are:

- ReLU, given by the formula $\text{ReLU}(x) = \max\{0, x\}$.
- Logistic (or Sigmoid) function, defined by $\sigma(x) = \frac{1}{1+e^{-x}}$. This function has two horizontal asymptotes, $\lim_{x\to-\infty}\sigma(x) = 0$, $\lim_{x\to\infty}\sigma(x) = 1$
- Hyperbolic tangent function, defined by $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. This function is similar to the sigmoid function, but takes values in $(-1, 1)$

3.1.2 Information Flow in a Neural Network

When training, i.e. adjusting the weights W_i, b of our networks to fit our task, the information is propagated in two steps.

- First, we apply the network to our input, and compute the output. This step is called **forward propagation**.
- To adjust the network, we need to understand how the weights influence the error that the model makes. In the supervised learning context, our dataset is composed of pairs (x_i, y_i), where xs are the examples we want to give to the model, and ys is the labels we want the model to predict.

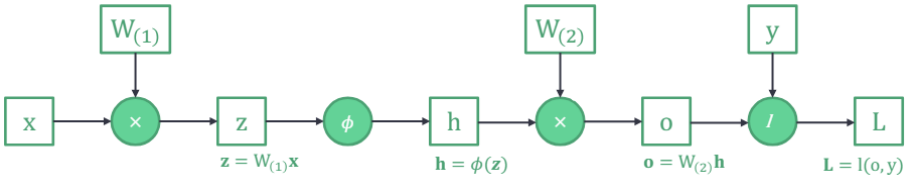


Figure 3.2: Computational graph of a MLP with one hidden layer.

Let L be a **loss function**, i.e. a scalar function that measures the error made by the model. Our goal is to minimize $L(o, y)$ given $o = model(x)$, and y the real label of x. One typical way to do this is through gradient descent, where we modify the weights with the following rule:

$$W \leftarrow W - \eta \frac{\partial}{\partial W} L$$

Crucially, computing the gradient $\frac{\partial}{\partial W} L$ efficiently is very important. We achieve this by using the computational graph of the network, and by looking at the **backward propagation** of the information. Deep Learning and automatic differentiation tools like Tensorflow of Pytorch use the computational graphs to implement backpropagation very efficiently. Let's look at the computational graph of a one layer MLP, shown in Figure 3.2.

Let us compute the gradient of the loss function $\mathcal{L} = \mathcal{L}(o, y)$ with respect to the weights W_l at layer l. If $l = L + 1$, so we are in the output layer, we get, using the chain rule of derivatives,

$$\frac{\partial \mathcal{L}}{\partial W_{L+1}} = \frac{\partial o}{\partial W_{L+1}} \frac{\partial \mathcal{L}}{\partial o} = \frac{\partial \mathcal{L}(o, y)}{\partial o} h_L^T$$

To compute the gradient with respect to W_L, we need to pass through $z_L = W_L h_{L-1}$ and, before that, $h_L = \phi(z_L)$. We obtain

$$\frac{\partial \mathcal{L}}{\partial h_L} = \frac{\partial o}{\partial h_L} \frac{\partial \mathcal{L}}{\partial o} = W_{L+1}^T \frac{\partial \mathcal{L}}{\partial o}$$

To differentiate w.r.t z, we simply have to multiply element-wise the vector $\frac{\partial \mathcal{L}}{\partial h_L}$ by the derivatives of the activation functions $\frac{\partial \phi}{\partial (z_L)_i}$. We use a diagonal matrix D_{z_L} such that $(D_{z_L})_{ii} = \frac{\partial \phi}{\partial (z_L)_i}$ Putting together,

$$\frac{\partial \mathcal{L}}{\partial W_L} = \frac{\partial z_L}{\partial W_L} \frac{\partial \mathcal{L}}{\partial z_L} = D_{z_L} W_{L+1}^T \frac{\partial \mathcal{L}}{\partial o} h_{L-1}^T$$

At layer $L - 1$, we have

$$\frac{\partial \mathcal{L}}{\partial h_{L-1}} = \frac{\partial z_L}{\partial h_{L-1}} \frac{\partial \mathcal{L}}{\partial z_L} = W_L^T D_{z_L} W_{L+1}^T \frac{\partial \mathcal{L}}{\partial o}$$

and we can similarly compute the derivatives w.r.t. z_{L-1} and W_{L-1}.

At the generic step l then, unrolling the recursion we get

$$\frac{\partial \mathcal{L}}{\partial z_l} = D_{z_l} W_{l+1}^T D_{z_{l+1}} W_{l+2}^T ... D_{z_{L-1}} W_L^T D_{z_L} W_{L+1}^T \frac{\partial \mathcal{L}}{\partial o} = \prod_{k=l}^{L} \left(D_{z_k} W_{k+1}^T \right) \frac{\partial \mathcal{L}}{\partial o} \tag{3.1}$$

Crucially, we do not actually need to compute all the product, because we can just simply write

$$\frac{\partial \mathcal{L}}{\partial z_l} = D_{z_l} W_{l+1}^T \frac{\partial \mathcal{L}}{\partial z_{l+1}},$$

and $\frac{\partial \mathcal{L}}{\partial z_{l+1}}$ was already computed at the previous step. For W_i we obtain the equation

$$\frac{\partial \mathcal{L}}{\partial W_l} = \prod_{k=l}^{L} \left(D_{z_k} W_{k+1}^T \right) \frac{\partial \mathcal{L}}{\partial o} h_{l-1}^T \tag{3.2}$$

3.1.3 Vanishing and Exploding Gradients

The product in Equation (3.2) can cause some problems in training, if the network is particularly deep. In particular, if the elements of D_{z_k} are small, we can encounter **vanishing gradient** problems, i.e. the backpropagated gradient becomes smaller and smaller, and the first layers do not get almost any update. This also couples with the product of $l+1$ matrices: if the matrices all have small spectral radius, they tend to shrink the gradients even more. The product of the matrix can also become increasingly large. This is called **gradient exploding** problem. In this case the gradient can become unbounded, and cause severe instability in the training.

Across the years, deep learning practitioners have come up with solutions to solve these two problem:

- Firstly, the activation function of choice in very deep network is ReLU, or variants of it like GELU or Leaky ReLU. Sigmoidal, squashing functions saturate to soon and are a prominent cause of gradient vanishing.
- To contrast exploding gradients, expecially in recurrent networks, **gradient clipping** is used: If the norm of the gradient, $\|g\|$, is larger than a certaing threshold θ, we substitute g with $g \cdot \frac{\theta}{\|g\|}$.
- Smarter weights initialization like Xavier, or initializing to an orthogonal matrix (so that initially, $\|W_k\|$ is close to 1) are used nowadays.

As we will see, vanishing and exploding gradients problems are an issue also in recurrent networks, not caused by the deepness of the network, but by the possible length of the sequences.

3.2 Modeling Sequences

Multi-layer perceptrons can only handle fixed size length inputs: they are not suitable for handling sequences. Moreover, they do not have a way to deal with time in their inputs.

There are plenty of different tasks that involve sequences:

- One-to-one tasks involve, for each input x, producing an output o. The output depends only on x. This is not a sequence task, and can be handled by MLPs. One example is an image classification task.
- Many-to-one tasks involve producing an unique output $o = o(T)$, after having seen the entire sequence $x(0), ..., x(T)$. One example is a sequence classification task, like classifying the correct movement from sensor measuring.
- Sequence transduction involve, for each input at time t, $x(t)$, producing an output $o(t)$. The output can depend, directly or indirectly, on the previous (in same case even on the following) inputs $x(t)$. One example is classifying each frame of a video.
- One-to-many tasks involve producing a sequence output for each input. An example is image classification
- Many-to-many tasks involve transforming a sequence in another sequence, possibly of different lengths. An example is machine translation.

3.2.1 Autoregression Tasks

Now, we will focus on a particular sequence transduction task, called an autoregressive task. The objective is predicting the next step of a time series from the previous observations. More broadly, we want to estimate the probability of seeing a certain value, given all the previous,

$$\mathbb{P}(x(t)|x(t-1), ..., x(1)). \tag{3.3}$$

This task cannot be handled trivially by a model with a fixed input size. There are two ways to approach the problem

1. **Windowing**: Restrict the observations to a fixed window length k: this is equivalent to approximate $\mathbb{P}(x(t)|x(t-1), ..., x(1))$ with $\mathbb{P}(x(t)|x(t-1), ..., x(t-k))$. This allow us to use any classic machine learning approach we can think of, like MLPs.
2. **Latent models**: create a fixed-size representation of the sequence history. This is equivalent to approximate $\mathbb{P}(x(t)|x(t-1), ..., x(1))$ with $\mathbb{P}(x(t)|h(t))$, where $h(t)$ can be viewed as a summary of all the information given by $x(1), ..., x(t-1)$. More precisely, $h(t)$ is a function of $x(t-1)$ and $h(t-1)$. $h(t)$ is called latent because it is not a random variable of which we can ever have any observation in our training set –from the external–.

3.2.2 Sequence Modeling

Autoregressive tasks are a fundamental part of **sequence modeling**. A sequence modeling task involve predicting the probability of a given sequence $(x_1, ..., x_T)$. Using the chain rule of probability we can decompose the task in an autoregressive task:

$$\mathbb{P}(x_1, ..., x_T) = \mathbb{P}(x_1) \prod_{t=2}^{T} \mathbb{P}(x_t | x_{t-1}, ..., x_1) \tag{3.4}$$

If the task is too hard, or if it useful due to prior knowledge, we can impose a Markovian condition on the model: we assume that the future is independent of the past, given the recent history. In other words, we fix a window size in the conditioning variables. For example a first order Markov model decomposes the joint probability $\mathbb{P}(x_1, ..., x_T)$ as

$$\mathbb{P}(x_1, ..., x_T) = \mathbb{P}(x_1) \prod_{t=2}^{T} \mathbb{P}(x_t | x_{t-1}).$$

A general k-th order Markov model is

$$\mathbb{P}(x_1, ..., x_T) = \mathbb{P}(x_1) \prod_{t=2}^{T} \mathbb{P}(x_t | x_{t-1}, ..., x_{t-k}) \tag{3.5}$$

3.3 TDNN and 1D Convolution

3.3.1 TDNN

The decomposition of Equation (3.5) is the idea that defines **Time Delay Neural Networks** (TDNN).

A TDNN applies the same feedforward neural network to the whole input series, taking in consideration only a fixed size window at a time. This way we can capture short-term local structures. This clearly enforces a Markovian condition of order equal of the window.

For each time step t, we apply an affine transformation and a non-linearity to a window of size $k + 1$, obtaining a fixed size representation $h(t)$. Then, we obtain the output $o(t)$ through another linear operation on $h(t)$.

$$h(t) = \phi \left(W_1 x[t - k, ..., t] + b_1 \right) \tag{3.6}$$

$$o(t) = W_2 h(t) + b_2 \tag{3.7}$$

3.3.2 1D Convolution

Nowadays, TDNNs are implemented through 1D Convolutional Neural Networks. A convolution is a linear operator that involve computing the product between a filter and the input. 2D convolution is the state-of-the-art approach

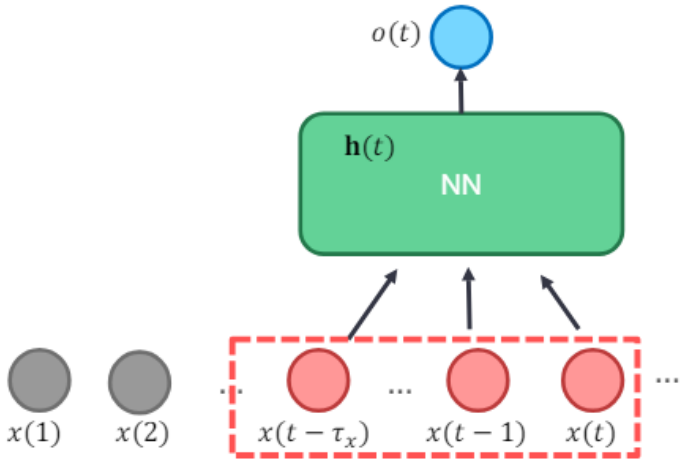


Figure 3.3: Architecture of a TDNN

for computer vision (Figure 3.4). 1D convolution involves applying the convolution operator along the temporal dimension of a time series. The kernel is a vector with a specific size. We can use more than one kernel for each convolutional layer, in order to capture different information from the sequence.

The kernel can be applied skipping a fixed number of time steps each step of the convolution. The number of time steps skipped is called **stride**. The TDNN presented earlier is a form of convolutional network with stride 1, i.e. the sliding window moves for one step at a time. Moreover, padding can be added to allow the convolution to include elements on the border of the time series.

The other core operation of convolutional is **pooling**. Fxixing a window size, max pooling takes the maximum over each window, average pooling takes the average over each window.

In the last layer, if we want a fixed size output, we may want to perform global pooling, i.e. pooling along the entire sequence.

Convolutional layers can also be effective wen stacked, to improve the receptive field: **Wavenet** (2016) stacks layers of dilated 1D convolution (Figure 3.5). A dilated convolution multiplies each kernel not by consecutive elements of the input series, but it skips a fixed number every time: the operation is $K * [x_t, x_{t+p}, x_{t+2p}, ..., x_{t+kp}]$, where K is the kernel of size $k+1$ and p is the dilation factor. **ROCKET** (2020) uses a large number of random convolutional filters and global max pooling to perform accurate time series classification in very few time (Figure 3.6).

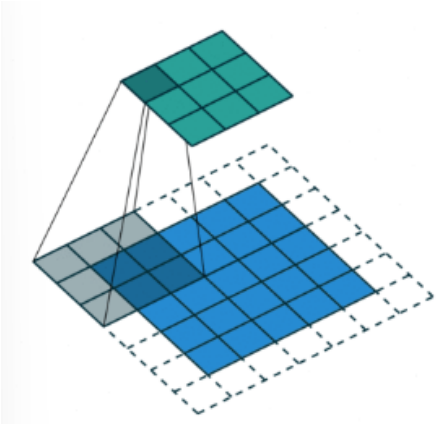


Figure 3.4: 2D convolution

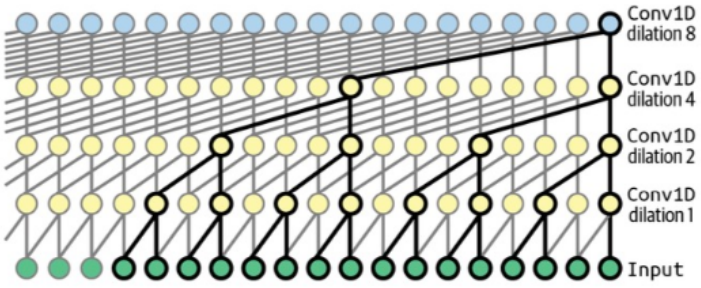


Figure 3.5: Wavenet architecture

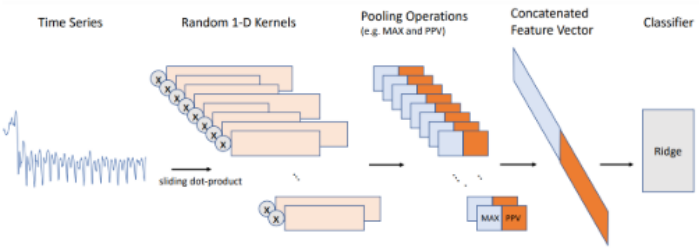


Figure 3.6: ROCKET architecture

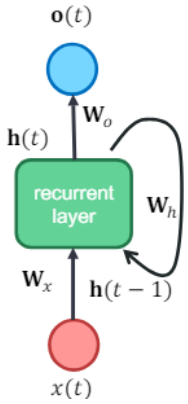


Figure 3.7: Computational graph of a RNN. The computation presents a loop, because h_{t-1} is needed to compute h_t.

3.4 Vanilla RNN

We have seen TDNN as an example of explicit Markov model of order k, where k is the window size.

The other type of model that we have described is latent space models, where we approximate the next step prediction $\mathbb{P}(x_t|x_{t-1}, ..., x_1)$ with a conditioning to a latent space variable, $\mathbb{P}(x_t|h_{t-1})$ The hidden state h_{t-1} is used to store all the information up until time $t-1$: $h_t = f(x_t, h_{t-1})$. In Recurrent Neural Networks, the function f is a composition of affine transformations and non-linearity:

$$h_t = \phi(W_x x_{t-1} + W_h h_{t-1} + b). \tag{3.8}$$

The output is then obtained through another non-linearity, $o(t) = W_o h_t + b_o$ The computation of h_t is the same at every step t: the parameters W_x, W_h, b are shared between the time steps. This makes the network recurrent, because now its computational graph is not directed, but it has a loop.

Vanilla RNNs models are based on some important assumptions:

- Causality: a system is causal if the output at time t_0 only depends on input at time $t < t_0$, and not on the future states. This assumption is fundamental, because the hidden state is only a function of the past
- Stationarity: the input distribution are time invariant. This is fundamental, because a RNN has the same set of parameters for every time step. This assumption allows as to have a number of parameters that is fixed in the sequence length, but the network may fail if the distribution of the input shifts.
- Adaptivity: this is an assumption on the model: the state transition function is realized by a neural net with free parameters, hence it is learnt from the data.

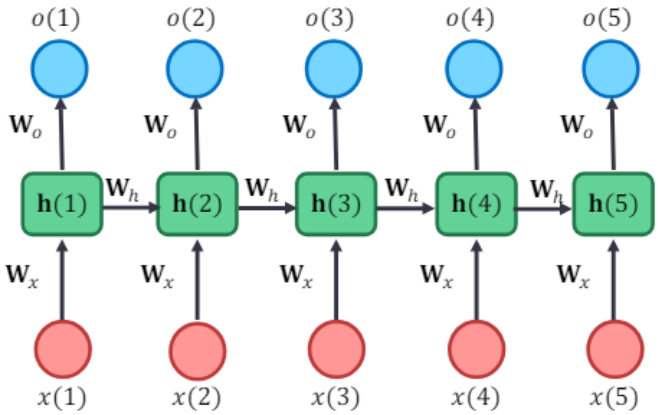


Figure 3.8: Unfolded computational graph of a RNN.

3.4.1 Unfolding the Computational graph

A recurrent network is deep even if it is composed of just one recurrent layer: the recurrence adds another dimension. This comes at a cost of potential instabilities in the forward and in the backward propagation, just as it happens in deep neural networks.

In fact, the main algorithm that has been used to train RNNs is a variation of backpropagation called backpropagation through time (BPTT). This algorithm performs backpropagation on the unfolded graph (Figure 3.8)

If the input sequence has length T, the algorithm is very similar to perform backpropagation on a feedforward network with T layers. Moreover, the recurrency matrix is always the same, therefore in Equation (3.2), we are multiplying T times the *exact same matrix*. For this reasons, the vanishing and exploding gradient issues of deep NNs are even more prominent in RNNs. If in deep feedforward networks there was the possibility for some matrices W_k to have $\rho(W_k) > 1$, and for some other matrices $\rho(W_{k'}) < 1$, effectively counterbalancing the gradient vanishing/exploding effect, this is just not possible in recurrent networks.

Let's explicitely write the equations for the gradient w.r.t W_h and W_x. The only difference with respect to Equation (3.2) is that we have to sum the gradient at each time-step, being the matrices the same. Let N be the sequence length (so that it is not confused with the transpose)

$$\frac{\partial \mathcal{L}}{\partial W_h} = \sum_{t=1}^{N} \prod_{k=t}^{N} \left(D_{z_k} W_h^T \right) W_o^T \frac{\partial \mathcal{L}}{\partial o} h_{t-1}^T \tag{3.9}$$

$$\frac{\partial \mathcal{L}}{\partial W_x} = \sum_{t=1}^{N} \prod_{k=t}^{N} \left(D_{z_k} W_h^T \right) W_o^T \frac{\partial \mathcal{L}}{\partial o} x_t^T \tag{3.10}$$

The product of at most N times W_h (plus the diagonal matrix of tanh derivatives) makes it so that, if N is large, we loose almost surely information on the first elements of the sequence. One way to prevent this is controlling the spectral radius of W_h. This idea is used in reservoi computing, where W_h is left untrained.

Truncated BPTT

In the full BPTT algorithm, we need to store all the T intermediate activations. This can be impractical for very long time series. A way to fix this problem is truncating the backpropagation after k steps. This is feasible, because in most cases after few steps we have anyway a loss of almost all the information. The Equations (3.9) and (3.10) become

$$\frac{\partial \mathcal{L}}{\partial W_h} = \sum_{t=N-k+1}^{N} \prod_{k=t}^{N} \left(D_{z_k} W_h^T\right) W_o^T \frac{\partial \mathcal{L}}{\partial o} h_{t-1}^T \tag{3.11}$$

$$\frac{\partial \mathcal{L}}{\partial W_x} = \sum_{t=N-k+1}^{N} \prod_{k=t}^{N} \left(D_{z_k} W_h^T\right) W_o^T \frac{\partial \mathcal{L}}{\partial o} x_t^T \tag{3.12}$$

3.4.2 Alternatives to BPTT

3.5 Reservoir Computing

We have seen that backpropagation through time has many issues in terms of vanishing/exploding memory. Furthermore, the backpropagation mechanism is especially biologically implausible as a way to learning sequences.

This brought researcher to develop a broad area of research, that studies how to drop the learning algorithm from the DL pipeline, and only keep the architectural biases. This techniques take the general name of reservoir computing. We have already seen an example of reservoirs of spiking neurons, with Liquid State Machines. Now, we will develop a theory about randomized artificial recurrent neural networks. We are especially interested in **echo state networks** (ESN). ESNs follow the very same equation of RNNs (3.8), but leave the three parameters W_h, W_x, b untrained and initialized under stability conditions. Often the dimensionality of $h(t)$ is very large. Therefore, the hidden states $h(t)$, $t = 1, ..., N$, create a large reservoir of units.

The only part that is trained is the **readout** layer

$$o_t = W_o h_t + b_o,$$

using simple linear regression methods.

This allows the training phase to be very fast, since we do not need to backpropagate information all the way across the input sequence.

The high dimensionality of the reservoir gives us some guarantees that a linear regression as a readout is sufficient: Cover's Theorem guarantees that non-linear embeddings in a high-dimensional space make the problem more likely to be linearly separanble.

Moreover, Echo State Networks are universal, in the sense that they can approximate arbitrarily well any **fading-memory filter**, i.e. any function that maps sequences into sequences $F : (\mathbb{R}^N)^{\mathbb{Z}} \to (\mathbb{R}^N)^{\mathbb{Z}}$, such that the output of a sequence u_1 approximate arbitrarily well the output $F(u_1)$ of a sequence u_2, if u_1 approximate well u_2, from a certain point on.

3.5.1 Mathematical Description

The reservoir is a discrete input-driven dynamical system, defined by the state function

$$F : \mathbb{R}^{N_x+N_h} \to \mathbb{R}^{N_h} \tag{3.13}$$

with

$$F(x_t, h_{t-1}) = tanh(W_x x_t + W_h h_{t-1}) \tag{3.14}$$

Iterating Equation (3.13), we can define a function $\hat{F} : \mathbb{R}^{N_x{}^*} \times \mathbb{R}^{N_h} \to \mathbb{R}^{N_h}$ such that

$$\hat{F}(s, h_0) = \begin{cases} h_0 & \text{if } s = [\,] \\ F\left(x_t, \hat{F}([x_1, ..., x_{t-1}], h_0)\right) & \text{if } s = [x_1, ..., x_t] \end{cases}$$

Definition 7. We say that a filter $\hat{F}$ satisfies the Echo State Property if for all $s \in \mathbb{R}^{N_x \cdot \mathbb{N}}$, $h_0, z_0 \in \mathbb{R}^{N_h}$, then

$$\left\| \hat{F}\left(s[1:N], h_0\right) - \hat{F}\left(s[1:N], z_0\right) \right\| \xrightarrow{N\to\infty} 0 \tag{3.15}$$

This means that the final state does not depend on the initial condition h_0, but only on the driving input sequence s.

There are two known conditions for the ESP to hold, one necessary and one sufficient for the Echo State Networks (3.14):

1. Sufficient condition: involves controlling the L2-norm of W_h, or equivalently the largest singular value σ_1
2. Necessary condition: involves controlling the largest eigenvalue of W_h, λ_1.

The two theorem are stated as follows:

Theorem 4 (Sufficient condition)**.** *If the maximum singular value of W_h is smaller than 1 then (under mild assumptions) the ESN satisfies the ESP for any possible input.*

Theorem 5 (Necessary Condition)**.** *If the spectral radius $\rho(W_h)$ of W_h is larger than 1 then (under mild assumptions) the ESN does not satisfy the ESP.*

If $\rho(W_h) < 1$, we guarantee that the dynamic is globally asymptotically stable around 0.

The necessary condition says that we want a contractive dynamic for every input: $\|W_h\|_2 < 1$. As a known linear algebra fact, $\|W\|_2 \geq \rho(W)$, where ρ is the spectral radius, i.e. the largest eigenvalue in module. So, the hypothesis in the necessary condition is weaker than the hypothesis in the sufficient condition, as we want.

The sufficient condition is often too strong in practice, and the necessary condition is often used to initialize the reservoir matrix.

3.5.2 Initializing the Reservoir

Initializing W_h

- Generate a random matrix $\tilde{W}_h$, for example in $[-1, 1]$.
- Compute the spectral radius $\rho(\tilde{W}_h) = \tilde{\rho}_h$.
- Scale by the desired spectral radius ρ_h: $W_h = \tilde{W}_h \cdot \frac{\rho_h}{\tilde{\rho}_h}$.
- Now, $\rho(W_h) = \rho_h$.

Initializing W_x

- Generate a random matrix $\tilde{W}_x$, for example in $[-1, 1]$.
- Scale by the desired input scaling factor ω_{in}. Either by range: $W_x = \omega_{in}\tilde{W}_x$; or by norm: $W_x = \tilde{W}_x \cdot \frac{\omega_{in}}{\|\tilde{W}_x\|}$

In this phase the most important hyperparameters are the spectral radius ρ_h and the input scaling ω_x.

Contracting RNNs are imposed a markovian bias: the final output will mostly depend on a suffix of the whole time series.

In particular, if the system is globally asymptotically stable, all the trajectories driven by the same inputs will synchronize, after an initial transient phase. For this reason it is important to discard a certain number of initial hidden states, that could be affected by the initialization of the network. This technique is called **washout**.

However, if the system is unstable, the initial conditions will determine the dynamic, and the trajectories could never synchronize.

3.5.3 ESN Training

Typical echo state networks consist of a tanh pool of neurons, that are left untrained, and a linear readout.

However, the readout can really be any machine learning model for regression or classification. Let's see how to train a linear readout for a transduction task:

Suppose we have a training set $\{(x_t, y_t)\}_{t=1,..,T}$, and our objective is to learn the map $x_t \to y_t$

1. We run the network (3.14) on the input sequence (x_t), and obtain a sequence of reservoir states $[h_1, ..., h_T]$.

2. We washout the states, discarding the first w state, where w is a user determined hyperparameter. Let us call $H = [h_{w+1}, ..., h_N] \in \mathbb{R}^{N_h \times (N-w)}$.

3. Let $Y = [y_{w+1}, y_N] \in \mathbb{R}^{N_y \times (N-w)}$. Then, the optimal linear predictor $W_o \in \mathbb{R}^{N_y \times N_h}$ solves the least square problem

$$\min_{W_o} \|W_o H - Y\|$$

We can solve this optimization problem via Moore-Penrose pseudoinverse:

$$W_o = YH^+ = YH^T(HH^T)^{-1}.$$

If the problem is ill-posed, we can use Tichonov regularization, imposing

$$W_o = YH^T(HH^T - \lambda I)^{-1},$$

where λ is the regularization strength and I is the identity matrix.

3.5.4 The Echo State Property

Now, we will prove some condition for the ESP to hold. We assume a ESN of the form (3.14).

Definition 8. The reservoir has contractive dynamics whenever its state transition function $F(x, \cdot)$ is Lipschitz-continuous with constant $L < 1$, i.e. whenever it exists a constant $0 < L < 1$ such that for all $x \in \mathbb{R}^{N_x}$, for all $h, h' \in \mathbb{R}^{N_h}$,

$$\|F(x, h) - F(x, h')\| \leq L\|h - h'\|$$

This condition is sufficient for the ESP to hold:

Theorem 6. *If an ESN has a contractive state transition function F (in any norm), and bounded state space, then it satisfies the ESP (for any input)*

Proof. Let $s \in \mathbb{R}^{(N_x)N}$ be the input sequence, and let h_0, z_0 be two initial reservoir states. Let $h_i = F(x_i, h_{i-1})$, $z_i = F(x_i, z_{i-1})$ for $0 < i \leq N$. Then,

$$\begin{aligned}
\|\hat{F}(s, h_0) - \hat{F}(s, z_0)\| = \|h_N - z_N\| &= \|F(x_N, h_{N-1}) - F(x_N, z_{N-1})\| \\
(\textit{lipsch.}) &\leq L\|h_{N-1} - z_{N-1}\| \\
&= L\|F(x_{N-1}, h_{N-2}) - F(x_{N-1}, z_{N-2})\| \\
(\textit{lipsch.}) &\leq L^2\|h_{N-2} + z_{N-2}\| \\
&\leq ... \\
&\leq L^N\|h_0 - z_0\|
\end{aligned}$$

Thus, if s has length N, $\|h_N - z_N\| \leq L^N\|h_0 - z_0\|$. Because $L < 1$, the right hand side of the equation vanishes as $N \to \infty$, proving the ESP. □

From Theorem 6, it is easy to prove Theorem 4 for tanh reservoirs with $\|W_h\|_2 < 1$.

Proof (Theorem 4). Let $x \in \mathbb{R}^{N_x}$, $h, h' \in \mathbb{R}^{N_h}$. Then

$$\begin{aligned}\|F(x,h) - F(x,h')\|_2 &= \|\tanh(W_x x + W_h h) - \tanh(W_x x + W_h h')\|_2 \\ \text{(tanh is 1-lipschitz)} &\leq \|W_x x + W_h h - W_x x - W h h'\|_2 \\ &= \|W_h(h - h')\|_2 \\ &\leq \|W_h\|_2 \|h - h'\|_2.\end{aligned}$$

Thus, because $\|W_h\|_2 < 1$, the conditions for Theorem 6 are satisfied, and the ESP holds. □

Proof (Theorem 5). Now, we will prove the necessary condition. Let us assume that are the inputs are 0 (in Definition 7, the condition must hold for any input, so we can assume any input we want). In this case 0 is a fixed point of the system, because $F(0,0) = \tanh(W_x 0 + W_h 0) = \tanh(0) = 0$.

Linearizing in 0, we can study the eigenvalues of the Jacobian, to see if the equilibrium is stable.

$$h_t = F(0, h_{t-1}) \approx J_F(x_t, 0)(h_{t-1} - 0) + F(0,0) = J_F(0,0) h_{t-1} \tag{3.16}$$

In this case, the Jacobian $J(0,0)$ is exactly W_h. Therefore, if $\rho(W_h) \geq 1$, the fixed point is not stable: there exists an initial state h_0 for which the trajectory will not converge to 0: we just take $h_0 = v_{max}$, where v_{max} is the eigenvector corresponding to the maximum eigenvalue in module λ_{max}.

Doing so,

$$\|h_t\| = |\lambda_{max}|^t \cdot \|v_{max}\| \xrightarrow{t \to \infty} \infty.$$

□

In a real application, the inputs are not all null, so the condition $\rho(W_h) > 0$ could be not necessary nor sufficient for the specific set of inputs, because the steady state will be not equal to 0. In this, case the non-linearity introduces bifurcations, that make the ESP not hold in general.

3.6 Advanced RNN Models

3.6.1 Extensions of the basic RNN

Vanilla RNNs rely on the causal assumption

$$\mathbb{P}(x_t | x_1, .., x_{t-1}, x_{t+1}, x_T) = \mathbb{P}(x_t | x_1, .., x_{t-1}).$$

If we drop this assumption, we can define a bidirectional RNN: in this model we scan the sequence from start to end, and also from end to start, in parallel.

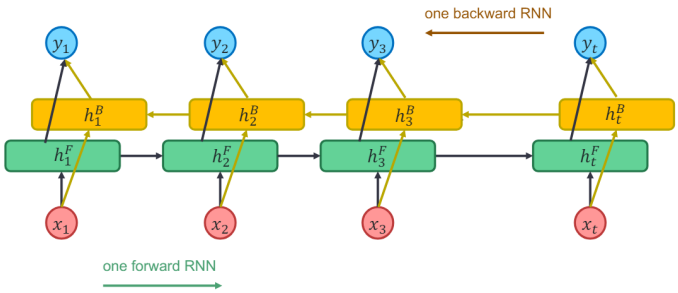


Figure 3.9: Bidirectional RNN architecture

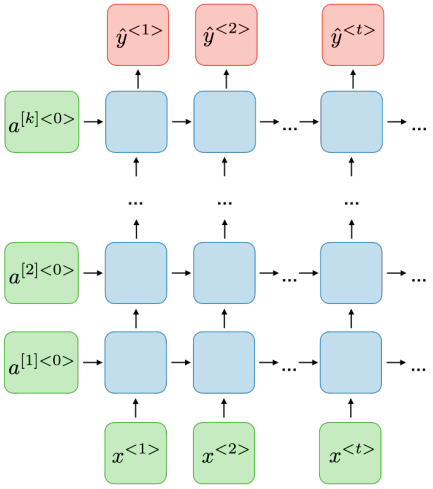


Figure 3.10: Deep RNN structure.

At each step, we have two hidden states, $h_t^F = \phi(x_t, h_{t-1}^T)$, from the forward scan, and $h_t^B = \phi(x_t, h_{t+1}^B)$ from the backward scan. The output y_t is then a function of the two states, $y_t = \phi_y(h_t^F, h_t^B)$.

Another natural extension of the vanilla RNN is stacking multiple RNN layers together. The hidden state at time t and layer l is used as input for the RNN at layer $l+1$: the general hidden state becomes $h_t^{l+1} = \phi_{l+1}(h_t^l, h_{t-1}^{l+1})$. Doing this, we are biasing the process towards modeling multiple scale dynamics.

3.6.2 Gated Architectures

The human brain is able to choose the important information to keep and the irrelevant information to discard. This is the idea behind implementing a gating mechanism in recurrent networks, that allow to selectively forget information

from the past hidden states. This gates can also help RNNs to deal with the loss of information due to gradient vanishing problems.

The first idea is to implement a leakage term α, such that

$$s_t = \phi(x_t, h_{t-1})$$
$$h_t = \alpha h_{t-1} + (1-\alpha)s_t$$

This leakage fundamentally corresponds to add a multiple identity matrix to the recurrency: there is a path on the backpropagation that is just multiplying α for each time step. The identity matrix obviously does not suffer from vanishing, but if $\alpha < 1$ (which we obviously impose, or else h_t would always be the same), there is still an exponential decay of the gradient.

The idea is then making the gates dependent on the input.

Long-Short-Term-Memory (LSTM) has two types of state information:

- the **hidden state** h_t is a short term state, that contains information on the previous time-step
- the **cell state** c_t is a long term state, that carries information that cross the time steps.

The model also contains three types of gates:

- **input gate** $i(t)$: determines which new information should be stored in the cell state
- **forget gate** $f(t)$: determines what information from the cell state should be discarded or forgotten
- **output gate** $o(t)$: controls which part of the cell state should be exposed to the output

The gates are implemented as a NN layer with a sigmoid activation, that take in input the previous hidden state and the current input. The equation for a generic gate $m(t)$ is

$$m(t) = \sigma(W_{hm}h(t) + W_{xm}x(t) + b_m) \tag{3.17}$$

$m(t)$ is then multiplied element-wise with the variables that we want to gate.

Let us analyze figure 3.11. The red rectangles are the gates, from left to right, forget, input and output gate.

- (lower left of the figure) Forget gate $f(t)$ is computed through Equation (3.17) from the hidden state and the input.
- Then (upper left) the long-term cell state is multiplied element-wise by $f(t)$:
$$\tilde{c}(t) = f(t) \otimes c(t-1)$$
.

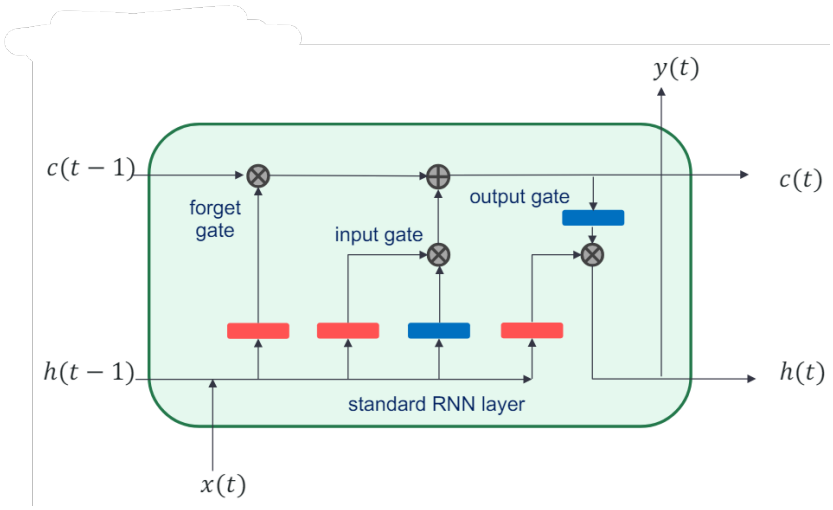


Figure 3.11: Architecture of a LSTM. The red rectangles are the gates, the blue ones are tanh layers. the $\otimes$ symbol represent the element-wise product.

- (down center) The contribute of the new input is computed by applying a tanh activation function to a linear combination of the previous hidden state $h(t-1)$ and the current input $x(t)$, resulting in $\tilde{g}(t)$ (blue square in the middle). This can be expressed as:

  $$\tilde{g}(t) = \tanh(W_{hg}h(t-1) + W_{xg}x(t) + b_g)$$

  The final result $g(t)$ is then computed by element-wise multiplying $\tilde{g}(t)$ with the input gate $i(t)$ (red square in the middle):

  $$g(t) = i(t) \odot \tilde{g}(t).$$

- (up-center) The cell state at time t is then computed by summing the discounted previous cell state, $\tilde{c}(t)$ with the new input term $g(t)$: this can be expressed as

  $$c(t) = \tilde{c}(t) + g(t)$$

  .

- (upper right) Finally, the new hidden state $h(t)$ is computed by the output gate multiplied by the cell state $c(t)$.

  $$h(t) = o(t) \otimes \tanh(c(t)).$$

- The hidden state $h(t)$ is also the value that the cell outputs at time t: $y(t) = h(t)$.

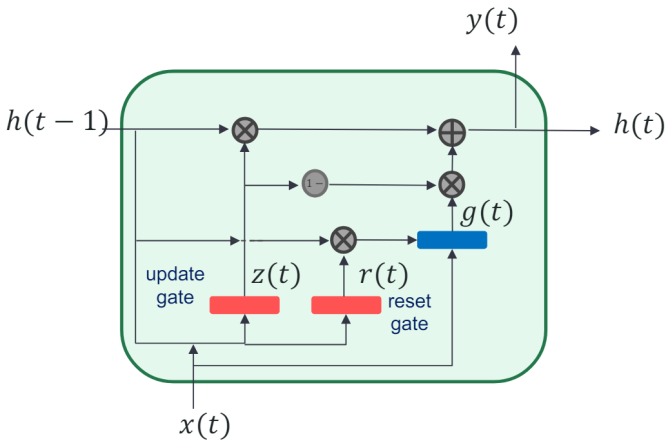


Figure 3.12: GRU architecture.

Overall, LSTM has eight weight matrices and four biases for each cell: two matrices and one bias for each gate, and the matrices and bias of the tanh layer. Vanilla RNN only has the last parameters mentioned. Due to this, the traning of LSTMs is much slower when compare dto vanilla RNNs.

The pro of the LSTM is that the gradient of $c(t)$ flows "uninterrupted": the only operations on c are point-wise multiplications and sums.

Gated Recurrent Unit (GRU) is similar to LSTM, but it only has just one hidden state $h(t)$, and two gates: a reset gate $r(t)$, and an update gate $z(t)$.

The reset gate $r(t)$ is applied to $h(t-1)$ before the recurrent tanh layer, and is used to decide what portion of the previous hidden state is used in the update of the new input.

The update gate $z(t)$ is used to decide how much of the previous state contributes to the new state. The Equations of GRU are as follows:

$$
\begin{aligned}
z(t) &= \sigma(W_{hz}h(t-1) + W_{xz}x(t) + b_z) \\
r(t) &= \sigma(W_{hr}h(t-1) + W_{xr}x(t) + b_r) \\
g(t) &= \tanh(W_{hq}(r(t) \odot h(t-1)) + W_{xq}x(t) + b_q) \\
h(t) &= z(t) \odot h(t-1) + (1 - z(t)) \odot g(t)
\end{aligned}
$$