UNIVERSITY OF PISA

DEPARTMENT OF COMPUTER SCIENCE

Master Degree in Artificial Intelligence

Notes of Computational Neuroscience

Based on lecture of professor C.Gallicchio

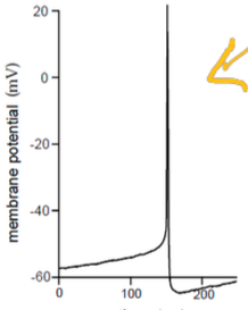
Written by: Aliprandi Francesco

Academic year 2024/25

(Part 1 Lecture 1) INTRO

Computational neuroscience: Discover and study the properties that characterize the mechanisms of data processing that take place in the brain. It **studies how networks of neurons can produce complex effects** (eg vision, learning, memory…)

Neurons:
- neurons are the focus of CNS.
- **Brains are aggregations of neurons**
- cells with the peculiar ability to communicate by means of voltage propagation, called "**spikes**"



spike: a spike is just a very brief (impulso) burst of electrical activity that a neuron sends out when it fires (quando si attiva). It's the basic way neurons communicate with one another.

CNS are interdisciplinary: neuroscience + computer science + psychology + philosophy

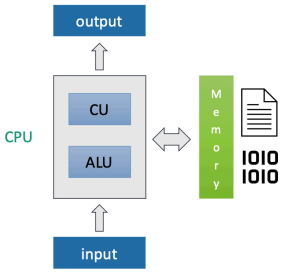
Neurobiological point of view: look at Artificial Neural Networks as a research tool to interpret neurobiological phenomena

Machine Learning point of view: look at neurobiology for new ideas to solve problems

Aim: Study biologically plausible mathematical models able to simulate neural dynamics

Brain as a special kind of computer:
- organically constituted, analog in representation, and parallel in he processing architecture
- not a general-purpose computer
- designed by natural evolution (not engineers)
- not only a cognitive device: thermoregulation, growth, reproduction, etc.
- temporal and spatial limitations
- efficient [!!]

Von Neumann architecture
- Memory is de-localized causing Von Neumann bottleneck
- **Moore's law**: the number of transistors on a chip doubles every 18 to 24 months, exponentially increasing processing power.
- **Koomey's law**: energy efficiency of computers, i.e. the number of operations per joule of energy consumed, doubles every approximately 1.5-2 years



Idee: both Moore's and Koomey's laws are slowing down due to physical limitations and rising costs

Energy consumption matters in AI application: DL training times doubles every 3 months while Moore's law stands that CPU computation power doubles every 2 years!

Nowadays energy consecution related to AI is an hot theme

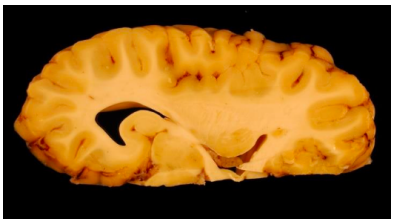
Brain vs calculators: brain is very much efficient in its computations. It's able of providing the corresponding of 30 PFlops with 20W of energy consumed, while calculators needs 10MW

Neuromorphic Computing: trying to mimic biological-neural architectures of the human nervous system to create more efficient and functional systems. **Nota:** in brain calculation and storage are not completely relocated

NERVOUS SYSTEM

It spreads into **central nervous system** and **peripheral nervous system**

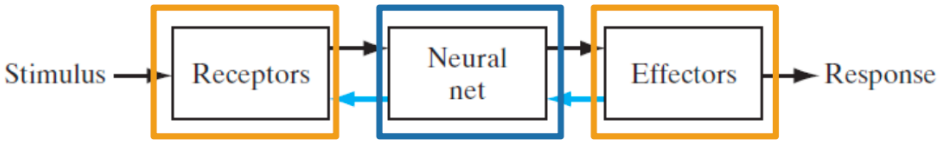
Central nervous system - neural net



- composed by Brain and Spinal cord (midollo spinale)
- **Function:** integrate and control centers. It continually receives and processes information
- **Grey Matter**: Neurons' body cells
- **White Matter**: Neurons' axons
- **Cerebral cortex**: Outer layer of the neural tissue in the brain

Peripheral Nervous System - Receptors/Effectors

- Converts external stimuli into electrical pulses
- Convert electrical pulses into discernible responses

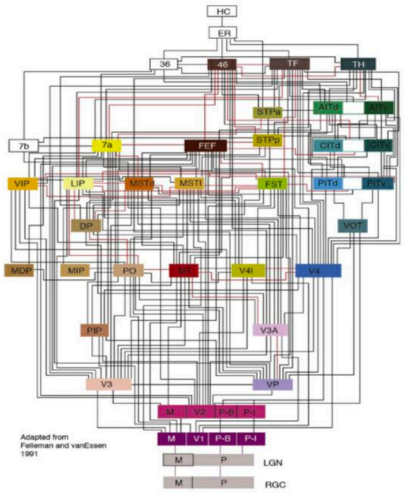


receptors collect the stimulus and the effectors allow to respond effectively

Deepness and recurrence in neural architecture

Deep (i.e., hierarchical) architecture: the greater the distance from cells responding to sensory inputs, the higher is the degree of information processing

Recurrent architecture: many forward projections are matched by a backward projection



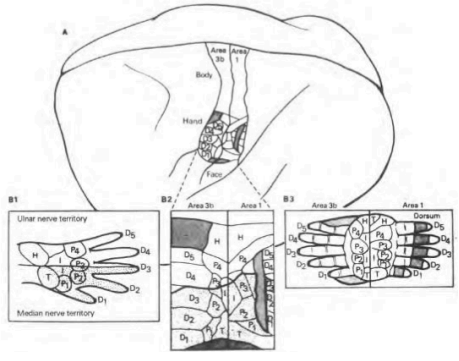
Brain Structure

neurons are organized in **systems**: distributed computational assemblies with many neurons that contribute to functional characteristics (eg visual system, autonomic system…

Eg: a schema of the hierarchy of the visual areas in the monkey brain. input enter at the bottom and is dislocated in many of the subsystems. This system contains a Lot of recursive connections

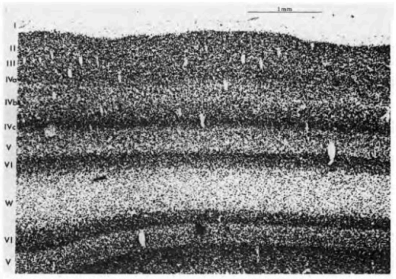
Topographic Maps: we can state that **adjacent neurons have adjacent** (e.g., visual) **receptive fields** and

Idea: In sensory and motor systems, neighbouring neurons process information from nearby areas, creating a spatial representation of the body or environment. This facilitates the processing and coordination of responses.

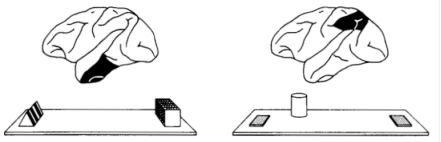


Layers and columns:

Layers: coherent layers are stacked one over the others (the grey layers in the image. Neurons in the layers can be different for this the colour difference)
columns: high commonality between cells in vertical columns. The commonality is in the way that similar neurons are processing informations



cross section through the striate cortex

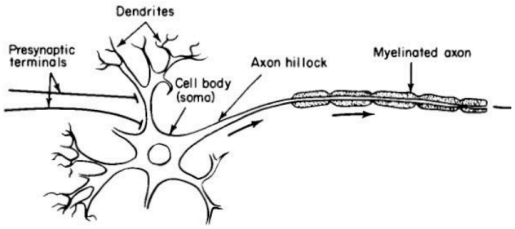


inferior temporal (IT) cortex posterior parietal (PP) cortex

Specialization of function:
different regions of the nervous system specialize to different tasks

Numbers:
- 10^{12} neurons in the human nervous system
- 10^{15} synapses in the human nervous system
- In 1 $mm3$ of cortical tissue: 10^{5} neurons, 10^{9} synapses (≈1 synapse/$\mu m3$)
- Each cortical neuron is connected to 3% of the neurons in the surrounding $mm3$

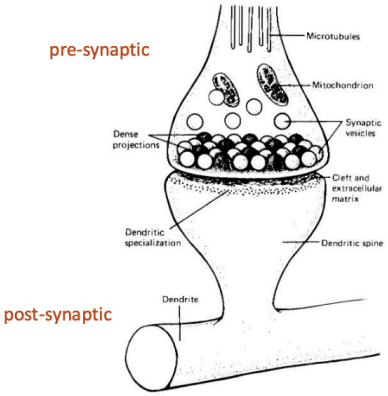
Neurons: elementary units of processing in the nervous system



Several types of neurons:
- **Anatomically:** several hundreds of different neurons
- **Different effects:** excitatory vs inhibitory neurons
- **Projections:** local vs long-range projections

Synapses: synapses are the **primary gateway** for inter-neurons communication

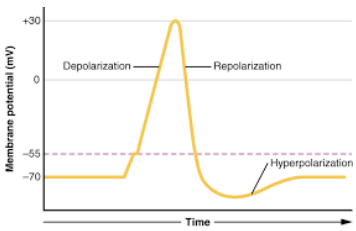
1. the voltage transient of a spike leads to the release of a neurotransmitter
2. the neurotransmitter binds to receptors at the post-synaptic side of the synapse
3. this causes ion-conducting channels to open, modifying the polarity of the post-synaptic cell membrane (**excitatory**: depolarization; **inhibitory**–hyperpolarization)



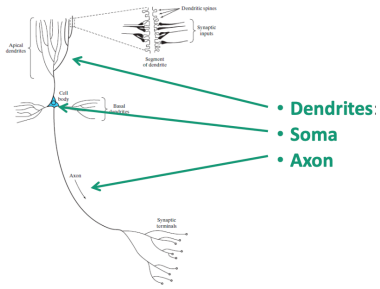
Ion Channels
Proteins in the neuron's membrane that act like **gates**, letting ions (like Na, K, Cl, Ca) to get in or out the cell membrane. Neurotransmitters have the effect of changing the membrane's ion conductance.

Depolarization: The postsynaptic neuron membrane potential becomes more positive. If it exceeds the threshold, it triggers a spike.

Hyperpolarization: The postsynaptic neuron membrane potential becomes more negative, making the neuron less likely to generate a spike.

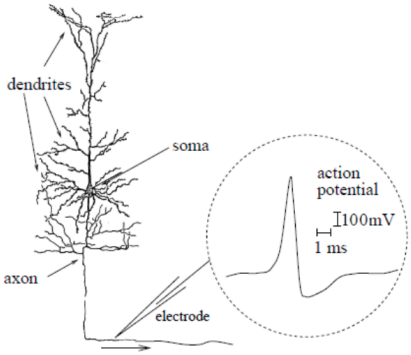


NEURAL MODELING



Three functionally distinct parts:
- **Dendrites**: input devices
- **Soma**: central processing unit
- **Axon**: output device

Action Potentials or Spikes:



- **Spikes**: elementary units of neuronal signal transmission.When the neuron spikes, its potential goes above the normal and then decreases it below the normal.
- Electrical pulses:
  - 100 mV of amplitude
  - 1-2 ms of duration
- **Spike train:** chain of spikes emitted by a single neuron
- **Absolute refractory period:** minimum distance between two spikes

Synapse

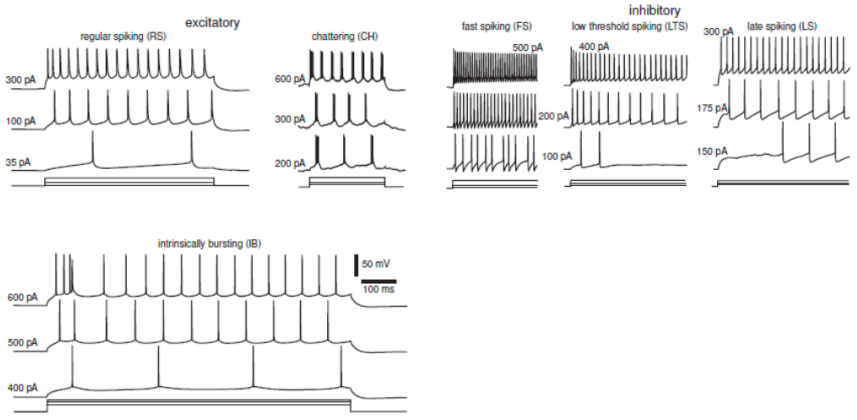
Def: a junction between **a pre-synaptic neuron** and a **post-synaptic neuron**

How it works:
- a presynaptic action potential triggers the release of neurotransmitters
- the postsynaptic cell membrane detects the neurotransmitters
- the permeability of the postsynaptic membrane to ions changes, leading to a change in membrane potential

Post Synaptic Potential (PSP): the voltage response of the postsynaptic neuron to a presynaptic spike

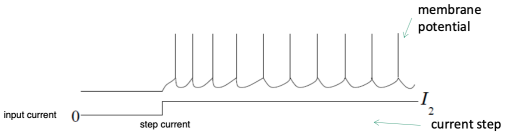
Spike Response Model:

There are only 6 fundamental classes of firing patterns:



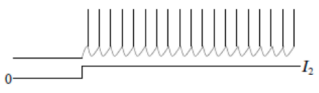
Why useful: if we want to model neuron we need to understand the dynamical functioning of them (wrt to only observing the structure)

Neuronal Dynamics: adaptation

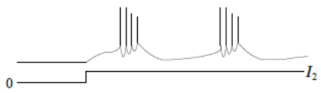


- We impose a stimulating external current I to the neuron (apply a current step)
- **Neuronal response:** a spike train
- Intervals between spikes increases until the dynamics reach a steady state of periodic firing
- **Adaptation**: a slow process that requires several spikes to emerge

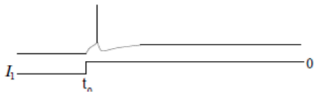
Fast spiking: no adaptation



Bursting: a sequence of spikes with long periodic intervals



Inhibitory rebound: the release of an inhibitory input can trigger a spike in the post-synaptic neuron



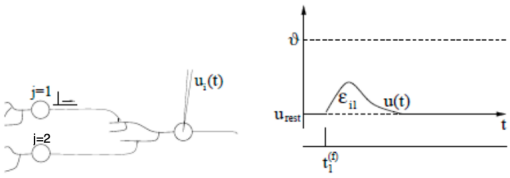
Spike Response Model intro :

Membrane potential $u(t)$: potential difference between the interior and the exterior of the cell

Constant value at rest $u(t) = u_{rest} \approx -65mV$ is our zero, meaning the rest value of the potential

Let's make an example:

Let's suppose a neuron with a potential $u(t)$ connected with two neurons in input. Let's suppose that neuron $j = 1$ fires, the PSP induced in neuron i is:



$$PSP_{ij} = \epsilon_{ij}(t) = u_i(t) - u_{rest}$$

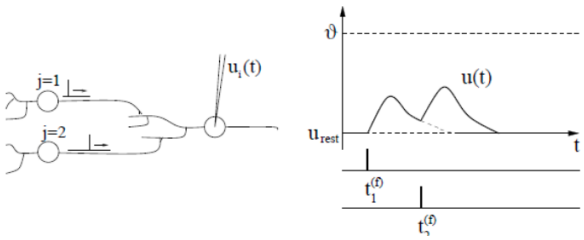
If PSP:

< 0 we have an inhibitory PSP (IPSP) we have **hyperpolarization**: the membrane become more negative

> 0 excitatory PSP (ESPS) we have **depolarization**: the membrane become more positive/less negative

When there are only a few presynaptic spikes the membrane potential can be approximated by a **linear combination of the individual PSP**

If the also $j = 2$ you can linear combine the contribution of that neuron



membrane potential of the post-synaptic neuron i at time t

post synaptic potential

$$u_i(t) = \sum_j \sum_f \epsilon_{ij}(t - t_j^{(f)}) + u_{rest}$$

sum over the pre-synaptic neurons

sum over the firing times

resting potential

Spike Response Model:

When the membrane potential **exceeds a threshold** the dynamics changes:

- **spike or action potential**: sudden depolarization (100 mV excursion) of the membrane potential
- **spike-afterpotential**: after the spike there is a phase of hyperpolarization below the resting value (where it's more difficult to re-depolarize it)

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

models the spike and the spike-afterpotential

PSPs

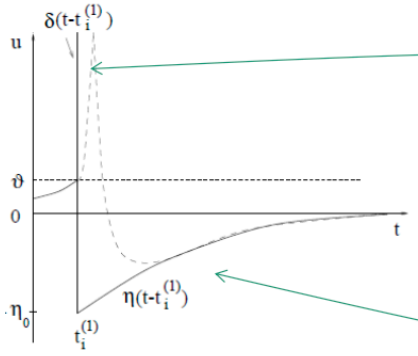
post synaptic potential

resting potential

$\hat{t}_i = \max\{t_i^{(f)} | t > t_i^{(f)}\}$ time of last spike of neuron i

If the membrane potential reaches the threshold from below then neuron i fires a spike

$$u_i(t) = \vartheta \text{ and } \frac{du_i(t)}{dt} > 0 \Rightarrow t_i^{(f)} = t$$



theta is our threshold and derivative positive means that u is increasing

Formal pulses

square pulse

$$\eta(t - t_i^{(f)}) = \begin{cases} 1/\Delta t & \text{for } 0 < t - t_i^{(f)} < \Delta t \\ -\eta_0 \exp\left(-\frac{t - t_i^{(f)}}{\tau}\right) & \text{for } \Delta t < t - t_i^{(f)} \end{cases}$$

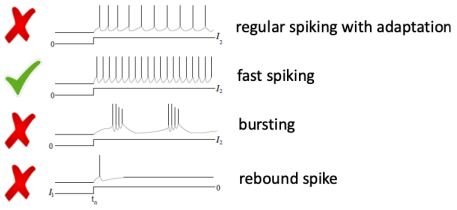
spike-afterpotential (negative)

Limitations of the spike response model

- Highly simplified model
- PSP always with the same shape
  - Does not depend on the state of the post-synaptic neuron (momentary level of depolarization, timing of previous spikes)
  - Does not depend on the spatial structure of synapses
- PSPs are linearly summed up
  - Does not consider non-linear interactions between PSPs
- Dynamics of the neuron depends only on the last firing time

No able of simulating many dynamical behaviours:

Not able to simulate many dynamical behaviors



NEURAL CODING

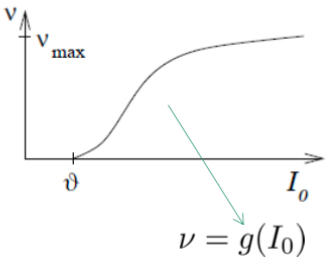
For neural coding we refers to **the way in which neurons communicates**. So it refers to the way the brain represents, processes and transmits information through the activity of neurons.

Rate codes: the code expressed by means of **firing rate**, computed as spiking count divided by the time elapsed

$$v = \frac{n_{sp}(T)}{T}$$

- Pros: Spikes are a convenient way to transmit a real value: just two spikes at 1/ v interval would suffice to encode the value v
- Cons: Unlikely that neurons can wait to perform a temporal average

Frequency – Current (FC) curve



on x the current (input of the neuron)

We can notice that the rate is a function of the current in input to the neuron and it acts like a sigmoid function

Rate as spike density: $\rho(T) = \frac{1}{\Delta t} \frac{n_k(t; t + \Delta t)}{K}$

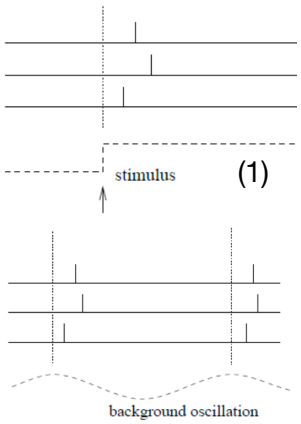
idea: instead of instead of averaging over time we average over k "experiments"

Rate as population activity: $\rho(T) = \frac{1}{\Delta t} \frac{n_{act}(t; t + \Delta t)}{N}$

idea: focus on communication of a population of neuron on the same time
nota: population of N identical neurons is not realistic/idealized

Spike codes

Neurobiological evidences say that spiking time has a role, so let's consider spikes instead of rates.

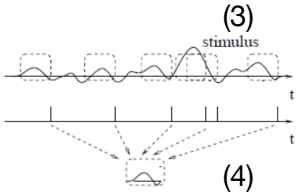


(1) Time to first spike: The information is encoded in the temporal distance of the neuron's response to the input

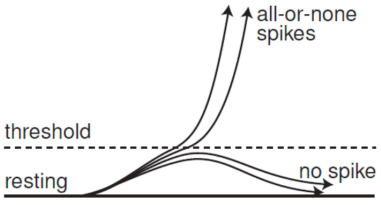
(2) Phase: The information is encoded in the phase of the spiking time with respect to a background oscillation (for example a neural wave like theta o gamma waves)

(3) Synchrony: The information is encoded in the pattern of firing synchrony within a population of neurons in response to a stimulus. How a population of neuron respond to external simulation?

(4) Reverse Correlation: used to understand the external stimulus that determines a spike in neurons. It consists in averaging the input under condition of an identical response, understanding the form of the input for which the neuron fires, "read the neural code"



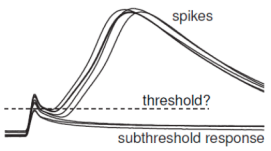
INTRODUCTION TO DYNAMICAL SYSTEM PERSPECTIVE



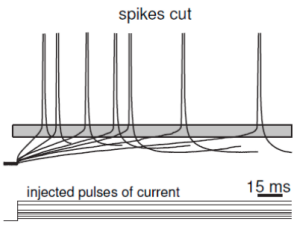
What make neurons fire: Often neurons are described as integrators (as they integrates signals coming from different neurons, either excitatory or inhibitory) with a threshold. Integrate the PSPs (sum the inputs incoming) and If the integrated value is above a threshold: fire

[!!] The existence of a threshold is questionable:

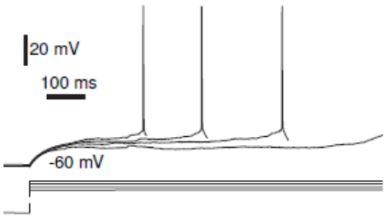
Injecting a **short input pulse** of current in the post synaptic neuron the threshold is not clear at all



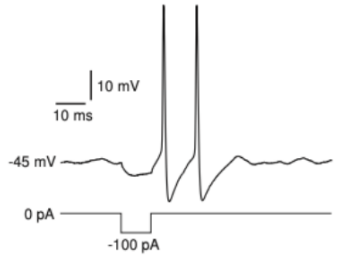
Injecting a **long but weak pulse** of current of various amplitude we can observe a slow depolarization and, eventually a spike. The firing threshold – if any – should be somewhere in the shaded grey region



rheobase: a threshold on the input current, i.e. the minimal amplitude of an injected current needed to fire a neuron.

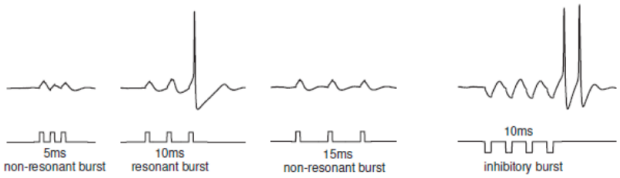


We can find the **rheobase** We begin by gradually decreasing the input current until we reach the minimum current required to elicit a spike. In the image we can observe that no clear rheobase is visible, so we can't tell if at the end the spike is not occurring because we found the rheobase or because the latency is longer than the duration of the experiment



Rebound spikes: specific type of neurons can spike when a negative input is eliminated: give a negative input and when we eliminate this negative current the neuron spikes. So where's the threshold here?

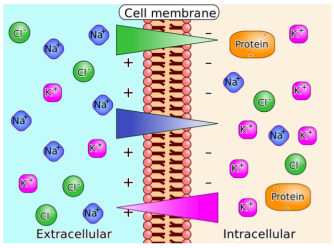
Response to specific resonant frequencies: we pass three inputs, if we change the frequency of this input we can have different responses, but the input pulse is still the same. What we can assume in this case about the threshold?



So what we wanna do? We aim at looking at a neuron also (and mainly) as a non-linear input-driven dynamical system a system that looks at its input through the prism of its own intrinsic dynamics

NOTA: from now on,  means very important, mandatory to know,  means just a detail. From part 3, they are omitted so follow what's on the slide

(Part 1 Lecture 2) NEURAL MODELING AND CNS



Conductance-based Models

Changes in the membrane potential $u(t)$ are due to ion currents (correnti ioniche), ions flows from outside to inside or vice versa, causing membrane potential changing. Main ions that take part into this process: Sodium Na^+, Potassium k^+, Calcium Ca^{2+}, Chloride Cl^- and others A^-

The **difference of ions concentration** between inside and outside the cell is responsible for the **generation of an electrical potential:**

Outside the neuron: higher concentration of *Na*, *Cl*, *Ca*
Inside the neuron: higher concentration of *K* (and other anions *A*, i.e., negatively charged ions)

Ion gates divides into pumps and channels. These are responsible for difference of ione concentration as they allow ions to flow from inside to outside and vice-versa. The difference in ions concentrations determines an electric potential, this because ions are positively or negatively charged.

Ion pumps: active transportation of ions. The result of their works is that ion concentration in the intracellular liquid differs from that in the extracellular liquid.

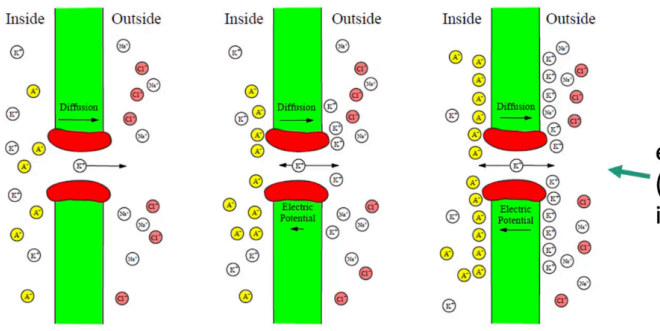
Active transport: transport ions *against the concentration gradient* (from low to high concentration) using energy (ATP)**.**

Ion channels: passive redistribution of ions. Result is reaching equilibrium (Nernst potential) of the specific ion current

Passive transport: allow ions to cross the membrane spontaneously following the concentration gradient (from high to low concentration) without using energy.

Two main forces drive the occurrence of ionic current across the membrane:
- **Concentration gradients** (diffusion): Ions move to regions where they are less concentrated
- **Electric potential gradients**: Ions are attracted by electrical charges of the opposite sign

Idea: if we have higher concentration of K^+ inside the membrane these tends to move outside. As K^+ ions moving outside, the A^- ions concentrate near the membrane trying to push back K^+ ions that are trying to go outside. **At the equilibrium the current is null** as K^+ ions are pushed outside by gradient difference but also attracted inside by electric potential gradient generated by A^- ions.



Nernst/Reversal Potential

$E_{[ion]}$ is the **potential equilibrium** of each specific ion channel is called its Nernst potential. The **Nernst potential** is the voltage at which the net flow of a specific ion across the membrane is zero, balancing the concentration gradient and the electrical force.

- if $\Delta_u < E_{[ion]}$ (potential lower then ion equilibrium)→ ions flow **into** the cell
- if $\Delta_u > E_{[ion]}$ (potential higher then ion equilibrium)→ ions flow **out** of the cell

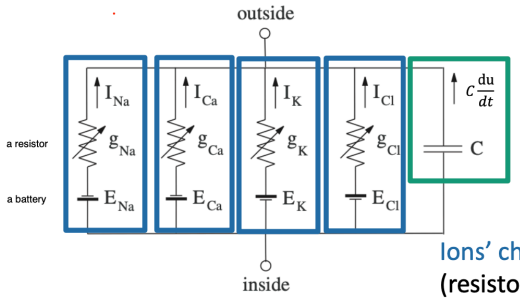
The **reversal potential** is the membrane voltage where the net ion flow through a channel is zero. It equals the Nernst potential for channels selective to one ion, but for channels carrying multiple ions, it depends on both the concentration and permeability of all involved ions.

When the membrane is above the reversal potential, the flow of ions reverses, changing direction.

Reversal Potential of [ion] — Boltzmann constant — temperature — ion density outside — ion density inside — charge of the ion

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

Equivalent circuit



Membrane (capacitor)

Ions' channels (resistors + battery)

This **equivalent circuit** represents the electrical properties of a neuron's membrane using circuit elements

Ion channels act as **resistors + batteries**:
- Each ion (Na^+, Ca^{2+}, K^+, Cl^-) has a conductance (g) represented by a **resistor**.
- The **battery** (E) represents the Nernst potential of the ion, which represents the driving force on the ion through that pathway (because of active ion transport through the cell membrane, the ion concentration inside the cell is different from that in the extracellular liquid. **Nota extra:** the polarity of the K battery is inside-negative because, at E_K, the voltage across the membrane is negative inside with respect to outside.

The **cell membrane acts as a capacitor (C)** as it separates the interior of the cell from the extracellular liquid :
- The term C (du/dt) represents the change in membrane potential over time. It has a key role for action potentials.

Now given $E_{[ion]}$ the reversal/Nernst potential, $(u - E_{[ion]})$ is the driving force that determines the flow of ions, the associate ion current from Ohms law is:

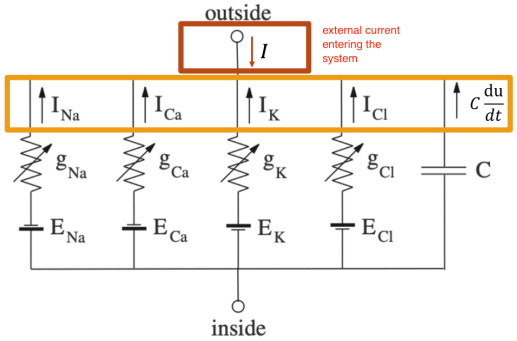
$$I_{[ion]} = g_{[ion]}(u - E_{[ion]})$$

conductance of the ion channel — reversal potential

with $g_{[ion]} = 1/R$ is the conductance (**conduttanza**)

Then Capacitante is: $C = \frac{q}{u} \rightarrow C\frac{du}{dt} = I_C$

Where:
- q the cumulative charge
- u the voltage across the capacitor and
- I_C the current that charges the capacitor



Thus considering also the **external current** entering the system and using Kirchhoff's Current Law:

$$C\frac{du}{dt} = I - I_{Na} - I_{Ca} - I_K - I_{Cl} = I - g_{Na}(u - E_{Na}) - g_K(u - E_K) - g_{Ca}(u - E_{Ca}) - g_{Cl}(u - E_{Cl})$$

Intuition: this comes trough the fact that applying the conservation of electric charge on a piece of membrane implies that the incoming current I(t) may be split in a capacitive current I_C which charges the capacitor C and further components I_k which pass through the ion channels:

So that: $I(t) = I_C(t) + \sum_k I_k(t)$ where the sum runs over all ion channels.

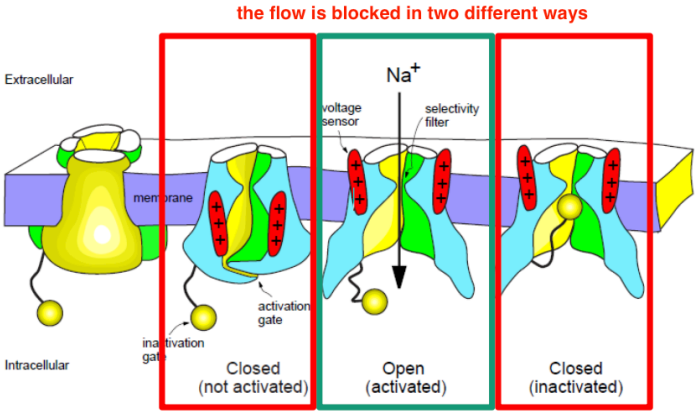
Thus reversing this relationship we can obtain our formula for the conductance

Limitations of this representation: the **conductances** of the ion channels can change **non-Ohmic** and **non constant** as it may change over time reacting to the membrane potential changing

Conductances of Ion Channels

Conductance is controlled by the gates or gating particles. We can assume, in a simplifying manner, that ion channels are composed by many gates. The main idea is that **to enabling ion currents to flow, all this gates must be open**.

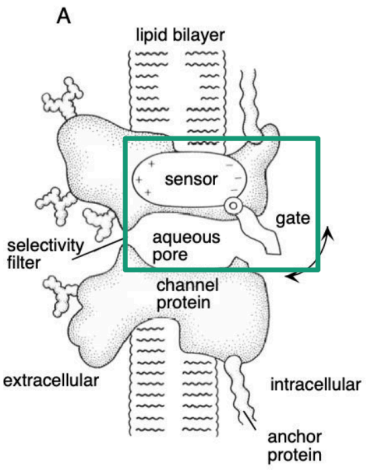
Voltage-dependent conductances are influenced by the membrane potential, as gates are sensitive to it.



Persistant conductances

A **voltage sensor** is connected to a (activation) gate that can open or close the pore:

- **gate opening**: activation of the conductance
- **gate closing**: de-activation of the conductance



Why persistent: because when the gate is open is persistently open, when closed is persistently closed

Probability of the channel to be opened: $p = n^k$

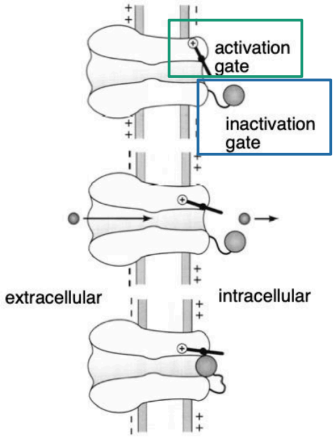
This depends on the fact that a gate has k components, each of which has a probability n of being open. n is called **gating variable** and is channel dependent.

[!!] Voltage dependency: depolarization of the membrane (when voltage is more positive) leads to increasing n

Transient conductances

Now the channel is regulated by two gates:

- the **activation gate** is open with a probability m^k (same intuition as persistent case)
- The **inactivation gate** (the ball) does not block the channel with probability h



So overall, the channel is opened with probability $m^k h$

[!!] Voltage dependency:

- Depolarization: m increases, h decreases

- Hyper-polarization: m decreases, h increases

The channel opens transiently while the membrane is depolarized

So considering this models, we can consider our **gating variables** being:

- **Activation variables:** n (persistent), m (transient)
- **Inactivation variable:** h (transient)

Dynamic of conductances

Given $x \in \{n, m, h\}$

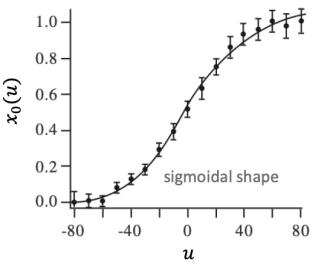
We can describe the change of this probabilities in time as: $$\frac{dx}{dt} = \frac{1}{\tau_x(u)}(x_0(u) - x)$$

Where:

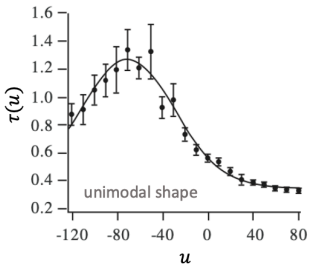
- $\tau_x(u)$ is the **time constant**: at which speed i'm approaching the asymptotic value. The larger the constant, the slower the approach to the asymptotic value; conversely, the smaller the constant, the faster the approach to the asymptotic value.
- x_0 is the **asymptotic value**: the value to which x approaches for t tending to infinity

Observing asymptotic value plot we can state that n and m increasing for increasing values of u, while observing time constant plot, we can state that h will tend to decrease while u increases

Nota: u is the membrane potential

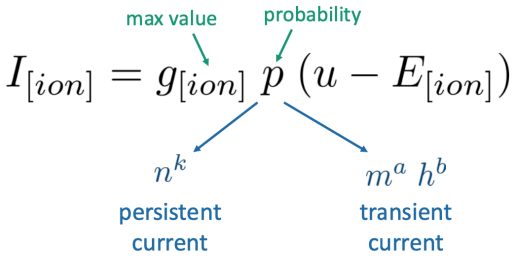


asymptotic value

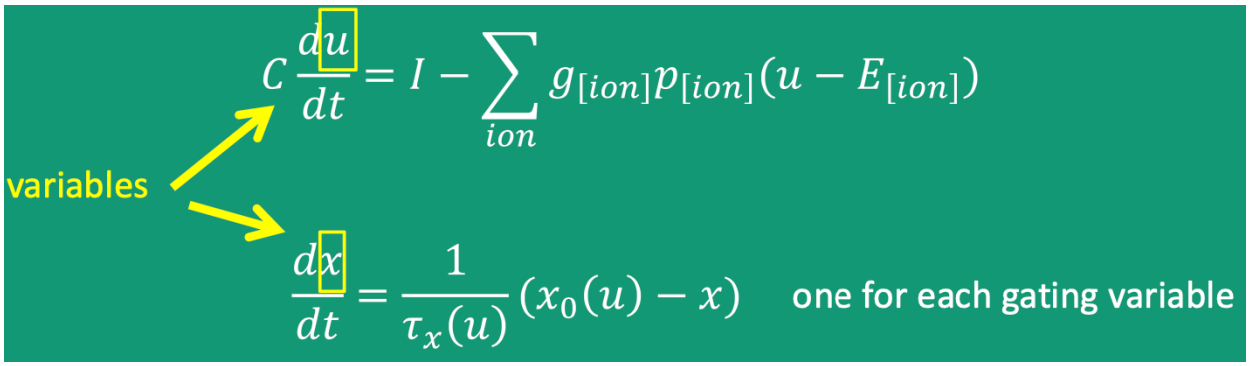


time constant

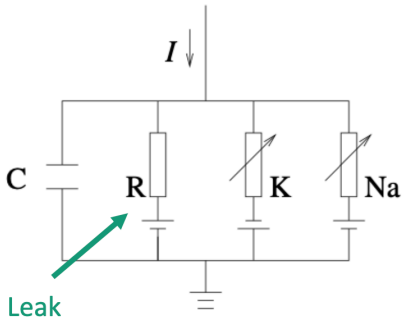
Considering these we can reformulate our Dynamic of Conductances as:



Thus our **conductance-based model** has 4 variables:



THE HODGKIN-HUXLEY MODEL



It's a Conductance-based model with 3 currents:

- Voltage-gated **persistent** K^+ with 4 activation gates
  → $I_k = g_K n^4 (u - E_k)$
- Voltage-gated **transient** Na^+ with 3 activation gates and 1 inactivation gate → $I_{Na} = g_{Na} m^3 h (u - E_{Na})$
- **Ohmic Leak current** (for all other ions) → $I_R = g_R (u - E_R)$

it's described as a set of 4 differential equations:

$$C\frac{du}{dt} = I - g_K n^4 (u - E_K) - g_{Na} m^3 h (u - E_{Na}) - g_R (u - E_R)$$

$$\frac{dn}{dt} = \frac{1}{\tau_n(u)}(n_0(u) - n) \qquad \text{activation gate of K}$$

$$\frac{dm}{dt} = \frac{1}{\tau_m(u)}(m_0(u) - m) \qquad \text{activation gate of Na}$$

$$\frac{dh}{dt} = \frac{1}{\tau_h(u)}(h_0(u) - h) \qquad \text{inactivation gate of Na}$$

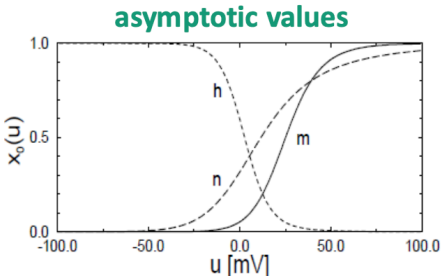
Original formulation of the gates in the model (dots on top means derivative):

$$\dot{n} = \alpha_n(u)(1 - n) - \beta_n(u)n$$
$$\dot{m} = \alpha_m(u)(1 - m) - \beta_m(u)m$$
$$\dot{h} = \alpha_h(u)(1 - h) - \beta_h(u)h$$

- u – membrane potential variable
- m, n, h - gating variables
- α, β – empirical functions adjusted by Hodgkin and Huxley

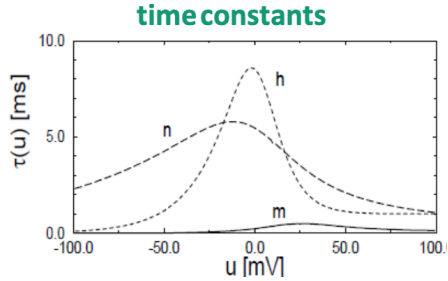
Hodgkin-Huxley Model – Dynamics

How gating variables influence the dynamic of the neuron (the behaviour of the membrane potential that determines the spiking activity)



Sodium (Na^+) inward current (m, h)

- The sodium moves **from outside to inside**
- Activation variable (m) increases for increasing membrane potential
- Inactivation increases (h becomes smaller) for increasing membrane potential
- **BUT**: activation is faster than inactivation (observable from smaller time constant value for m, ie it faster reaches the asympt. value)



Potassium (K^+) outward current (n)

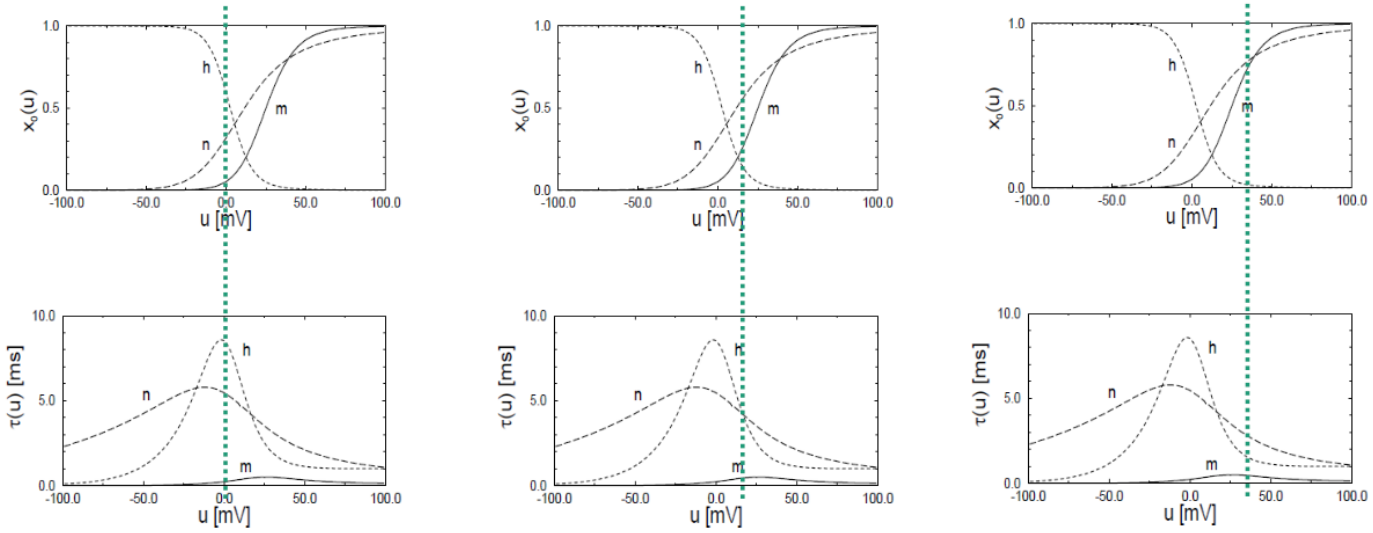
- The potassium moves from inside to outside
- Activation (n) increases for increasing membrane potential
- **BUT**: activation is relatively slow (slower than activation of Na^+, observable from higher time constant value)

Note from asymptotic values:

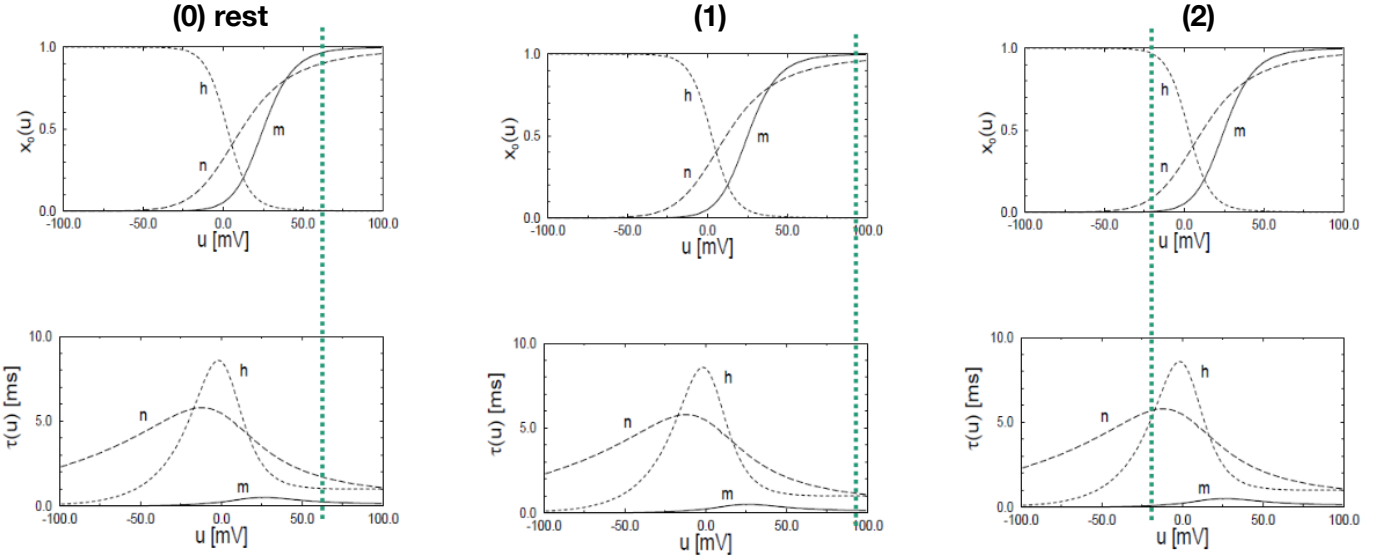
- Activation of Na (m) and of K (n) follow a **similar trend** but m is faster in growing (time constant is very small)
- Inactivation of Na (h) follows an **opposite trend**
- Both activation of K (n) and inactivation of Na (h) are much slower (higher time constant)
- close to the rest both m and n tend to be small

Hodgkin-Huxley Model – Spike Generation

1. An external input (e.g. an Excitatory PSP) leads to a depolarization (u increases)
2. Conductance of Na increases rapidly: Na ions flow in the cell and *u* increases even further (because *m* tend to grow fast)
3. if the feedback is strong enough the action potential is initiated (spike process is started)
4. At high values of depolarization, the Na current is stopped by the inactivation gate ($h \rightarrow 0$), conductance of K increases, hence K ions flow outside the cell
5. The membrane is re-polarized, with a negative overshoot (refractoriness)

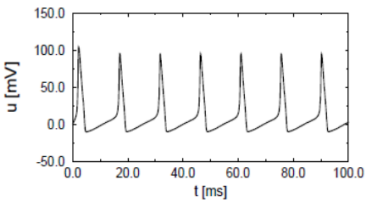


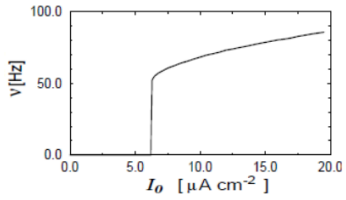
(0) rest **(1)** **(2)**



(3) **(4)** **(5)**

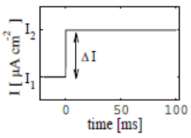
Regular spiking: apply a constant input I_0, if it's strong enough we observe a regular spiking





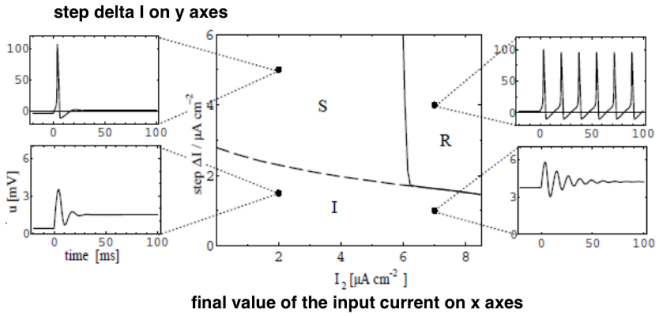
Step current input

If the input is in the form $I(t) = I_1 + \Delta I \theta(t)$ where $\Theta(t) = \begin{cases} 1 & \text{for } t > 0 \\ 0 & \text{otherwise} \end{cases}$



We can observe that:

- A large step **facilitates spike generation**
- Spikes are possible also for $I_2 = 0$ (rebound spikes)
- Repetitive firing is possible only for large values of the final current I_2
- **The action potential mechanism depends on both I_2 and the step current**



TWO - DIMENSIONAL NEURAL MODELS

The Hodgkin-Huxley model has 4 dimensional non-linear differential equations, so it's difficult to be analyzed and visualized. We need a **simpler 2 dimensional model** that enable **visualization** of the state space trajectories enabling the understanding of the dynamical behavior. This is obtained by approximating the neuron dynamics **using a set of differential equations with fewer variables**

Hodgkin-Huxley (4D)
u: membrane potential
n: K activation (slow)
m: Na activation (fast)
h: Na inactivation (slow)

→

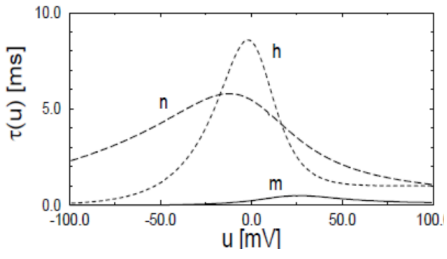
2d NEURAL MODEL
u: membrane potential
w: relaxation (recovery) variable (slow)

Approximations:

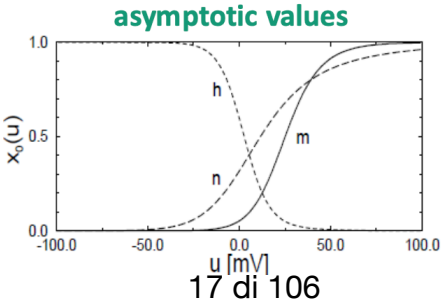
1. We can observe that **m** **has very fast dynamics** (because of its very small time constant value), so **we can treat it as an instantaneous function**: $m(t) \rightarrow m_o(u(t))$. So we can consider only the asymptotic value, without modeling its dynamics. **m** is no longer a variable of the differential equation, when it fires, it reaches instantaneously m_0
2. Time constants of **n** and **h** **are roughly the same** and plots of asymptotic values for **n** and **1-h** are very similar. So we can model **n** and **h** as a single variable **w**:

$$w = b - h = an \Rightarrow \begin{aligned} h &= b - w \\ n &= w/a \end{aligned}$$

time constants



asymptotic values



Reduction to 2 dimensions

$$C\frac{du}{dt} = I - \overbrace{g_k\, n^4\,(u - E_K)}^{I_K} - \overbrace{g_{Na}\, m^3 h\,(u - E_{Na})}^{I_{Na}} - \overbrace{g_L\,(u - E_L)}^{I_L} \qquad \left| \begin{array}{l} m(t) \to m_0(u(t)) \\ h = b - w \\ n = w/a \end{array} \right.$$

$$\frac{dn}{dt} = \frac{(n_0(u) - n)}{\tau_n(u)} \quad \frac{dm}{dt} = \frac{(m_0(u) - m)}{\tau_m(u)} \quad \frac{dh}{dt} = \frac{(h_0(u) - h)}{\tau_h(u)}$$

simplified equation

$$C\dot{u} = I - g_{Na}(m_0(u))^3(b - w)(u - E_{Na}) - g_K(w/a)^4(u - E_K) - g_L(u - E_L)$$

$\tau = RC$
Multiply both sides by $R = g_L^{-1}$

applying some math:

$$\begin{cases} \dot{u} = \frac{1}{\tau}\left(F(u,w) + R\,I\right) \\ \dot{w} = \frac{1}{\tau_w} G(u,w) \end{cases}$$

F and G are functions that determines the model, we can choose different functions obtaining different level of approximation

Morrise-Lecar Model

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

$$m_0(u) = \frac{1}{2}\left[1 + \tanh\left(\frac{u - u_1}{u_2}\right)\right] \qquad \tau(u) = \frac{\tau_w}{\cosh\left(\frac{u - u_3}{u_4}\right)}$$

$$w_0(u) = \frac{1}{2}\left[1 + \tanh\left(\frac{u - u_3}{u_4}\right)\right]$$

A first equation describes the evolution of the **membrane potential** u, the second equation the evolution of a slow **recovery variable** w. The **voltage** has been scaled so that one of the reversal potentials is unity

FitzHugh-Nagumo Model

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

$$\dot{u} = \frac{1}{\tau}\left(F(u,w) + R\,I\right) \qquad \dot{w} = \frac{1}{\tau_w} G(u,w)$$

where:

$$F(u,w) = u - \frac{1}{3}u^3 - w$$

$$G(u,w) = b_0 + b_1 u - w$$

(Part 1 Lecture 3) DYNAMICAL SYSTEMS

Dynamical systems theory attempts to describe systems in motion.

Def: a **dynamical system** is a mathematical object that describes a system (either artificial or real) that evolves over time (eg: Animal population sizes, Biological and artificial neural networks, Neuron activity, Covid spread…)

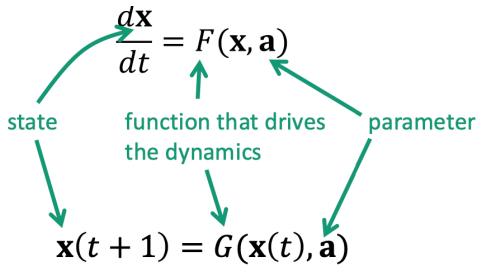
It's composed of:
- a **state** that describes the **current condition of the system** as a vector (of observable quantities). The **set of possible states** is called the **state space** (or the phase space) of the system
- a **dynamic** that describes how the system's state evolves over time. Dynamics are specified in terms of an **evolution rule** (or law, or function) that **governs the change of the state**

Considering the dimension of time, dynamical systems could be either **continuous** and **discrete.**

Continuous-time dynamical systems are ruled by **differential equations:** $\frac{d\mathbf{x}}{dt} = F(\mathbf{x}, \mathbf{a})$

Discrete-time dynamical systems are ruled by **iterated maps** (in a recursive fashion next state is a function of value at previous time step)**:** $\mathbf{x}(t+1) = G(\mathbf{x}(t), \mathbf{a})$

Notation:



Trajectory: the sequence of states that are exhibited by the dynamical system during its evolution

Equilibrium state or Stationary state:

A system is in an equilibrium state, or stationary state, $\mathbf{x}^*$ whenever the system in $\mathbf{x}^*$ is at rest. This results in:

- $d\mathbf{x}/dt = F(\mathbf{x}^*, \mathbf{a}) = 0$ for continuous systems. The idea is that equilibrium points represent **states where the system remains constant over time**, so rate of change of the system's state is zero in continuous case.
- $G(\mathbf{x}^*, \mathbf{a}) = \mathbf{x}^*$ so intuitively $x_t = x_{t-1}$, ie the state doesn't change

Limit cycle: an equilibrium trajectory, where the system continuously follows the same trajectory over time.

Property 1: when the system is in an equilibrium point it will never get out of it
Property 2: the behaviour of the system in a small neighbourhood of the equilibrium can vary **stable (attracting)** vs **unstable (repelling)**

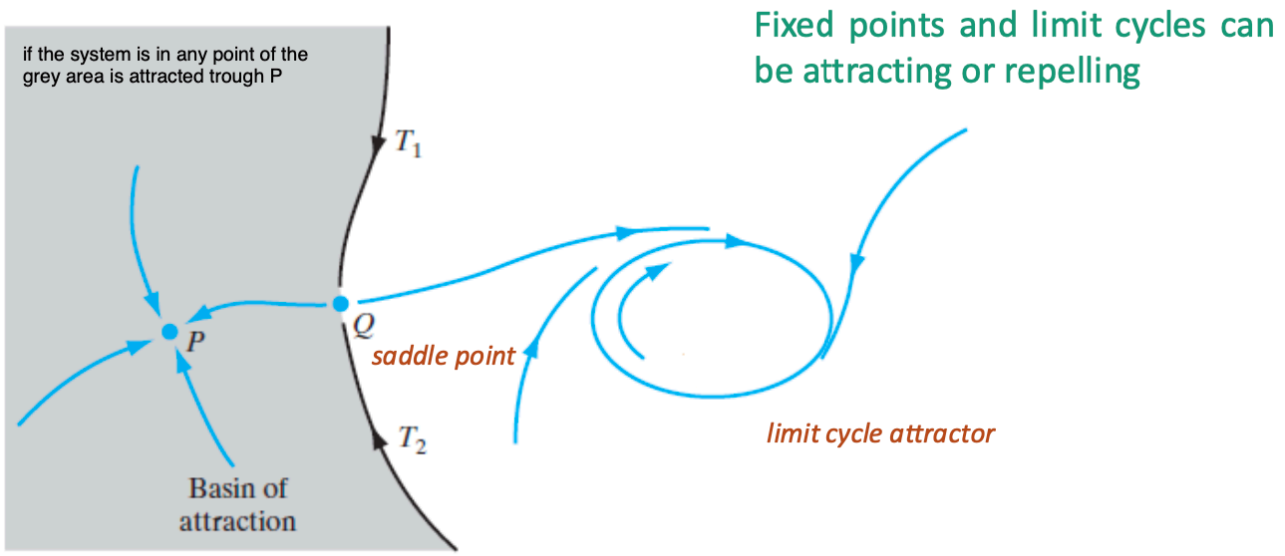
Attractor: a region in the state space where all trajectories converge from a larger subset of the state space (can be either a point or an orbit)
Repeller: points in a neighbourhood of the equilibrium move away from the equilibrium

Basin of attraction (bacino di attrazione)**:** the region of the state space for which trajectories will approach a specific attractor

Type of attractors:

- **Fixed point**: a point in which the system's evolution stops
- **Limit cycle**: a period orbit in which the system is stuck, not sopped but it cannot get out from this periodic behaviour
- **Quasiperiodic**: a limit cycle behavior that is not restricted to a single periodicity (multiple periodicities combined in an irrational fashion)
- **Chaos**: aperiodic, unpredictable behavior (the system is sensible to initial conditions) – no point in the state space is ever visited twice

Attractors: example

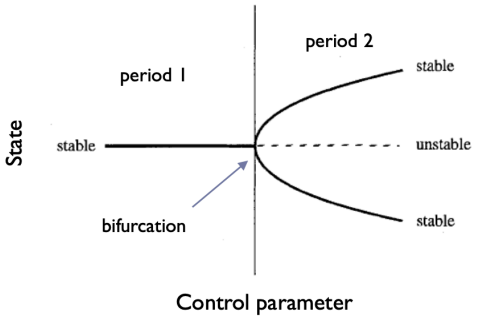


fixed point attractor

Idea: if I am in the white point i will approach the cycle and I will continue iterating trough it
Q is a **saddle point**: it attracts in some directions (T1, T2) and repelling in other directions (trough P or cycle attractor). This because they rely in between the two regions of the space with attractors

Bifurcations:

A dynamical system might be sensible to adjustments of the values of its parameters **a** (eg in neurons can be the input current to the neuron). So **the nature of attractors might change as a consequence of a change in the parameters' values.**



Def: in this context a **bifurcation** is a a qualitative change in the phase portrait of the system, so in the system's behaviour

a bifurcation occurs when a small smooth change made to the parameter values (the bifurcation parameters) of a system causes a sudden "qualitative" or topological change in its behavior

NEURONS AS DYNAMICAL SYSTEMS

Neurons are dynamical systems and their behavior can be modeled by resorting to dynamical systems theory

Variables of the state of the neurons:
- **Membrane potential**
- **Excitation variables** (e.g., activation of Na+): responsible for the upstroke of the spike
- **Recovery variables** (e.g., inactivation of Na+): responsible for re-polarization after the spike (downstroke)
- Adaptation variables: can affect excitability on the long run (we will ignore it)

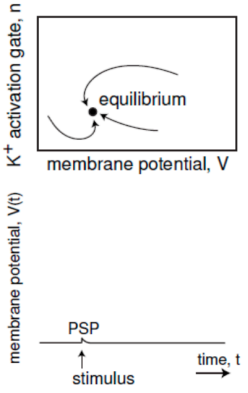
Phase portraits

Def: it's a graphical representation of the system's behavior over time. It allows the visualization of the trajectories that the system's state can follow over time under various conditions

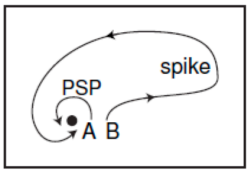
[!!] We consider a **2D state** composed by **membrane potential** and **recovery variable** (slow activation of K or slow inactivation of Na). Our **parameter** is the **current I injected.** At **rest** the state is considered in **equilibrium**

Equilibrium at rest

The equilibrium is an **attractor**. Suppose the system is stimulated with a small PSP (a small input current): the state will slightly deviate from the equilibrium… and eventually come back to rest (equilibrium)

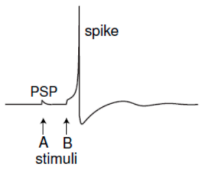


Rest



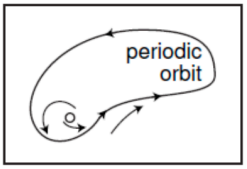
Spike

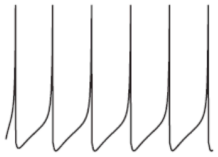
Suppose we stimulate the system with a larger input (perturbation B in the figure): the system deviates from the equilibrium substantially, a spike occur and finally the system comes back to the equilibrium (still an attractor)



Periodic Spiking

If a sufficiently strong input is injected, the neuron exhibits periodic spiking activity. The state of the neuron has now a **stable limit cycle** (periodic orbit), here we got a **bifurcation**: the external input has shifted the operation modality of the neuron's dynamics, stable fixed point disappears, stable limit cycle appears




Periodic

Observations:

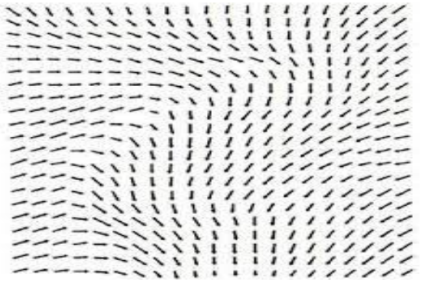
- **multiple stabilities could coexists together** (as seen in the neuron) so we can move the system from a stable situation to another. **Fixed points and limit cycles can coexists:** the neuron can be switched from one mode to another by transient input (in our case the current)
- **Neurons are excitable** (can generate spikes) **because they are near bifurcations from resting to spiking activity** (i.e. a little change in parameters can generate a bifurcation)
- Small perturbations in the input parameters can lead to large changes in the dynamics
- This sensitivity to the perturbation can correspond to the neuron's responsiveness to synaptic inputs

Phase plane analysis

The evolution of the system (trajectory) can be visualized in a phase plane $u(t),\ w(t)$. The **phase plane** contains **the direction of the flow of our dynamical system in each point**

After Δt, the system will move by $\begin{pmatrix} \Delta u \\ \Delta w \end{pmatrix} \approx \begin{pmatrix} \dot{u} \\ \dot{w} \end{pmatrix} \Delta t$

The flow field can be plotted in a phase portrait



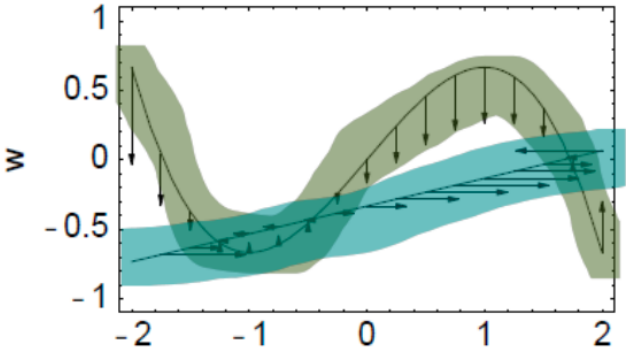
How to read: at each Δt the system move from one point to another (one arrow to another in the plot). The *arrows* indicate the *direction* and *speed* with which the system would move from any given state.

Nullclines

Def: Nullclines identify the regions in the phase plane where the flow is stationary along one direction

In our model we have 2 nullclines as we model the neuron in 2d system so we have one for each variable:

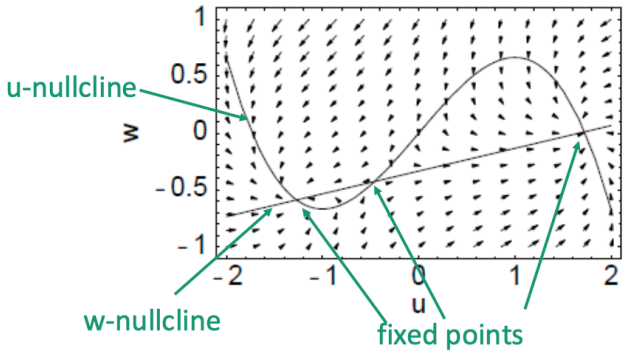
- **u-nullcline:** arrows are vertical, $\dot{u} = 0$
- **w-nullcline:** arrows are horizontal $\dot{w} = 0$



The **neuron as a dynamical system** is ruled by:

$\begin{pmatrix} \dot{u} \\ \dot{w} \end{pmatrix} = \begin{pmatrix} F(u,w) \\ G(u,w) \end{pmatrix}$, so a fixed point satisfies:

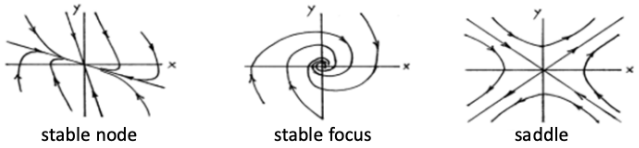
$\begin{pmatrix} F(u,w) = 0 \\ G(u,w) = 0 \end{pmatrix}$



[!!!] Fixed points of the system are given by the intersection between the nullclines

Stability of fixed points

- **Stable fixed point:** nearby states are attracted to it. They point straightforward to stable fixed point
- **Unstable fixed point:** nearby states are repelled from it
- **Saddle point:** some trajectories are attracted to the saddle point, others are repelled



the difference between stable node and stable focus is that in stable focus i'm approaching oscillating around the stable point

Example of stability on echo state network

Intuition: if my system is stable it means that all trajectory are attracted trough a fixed point converging to the same point. Depending on the parameters of the state transition function the origin can be an **attractor or a repeller,** making the system converging to the same stable point or to different points. In the case of the repeller, the ESN cannot generalize well, if we perturbate a little a state it will reach a different solution

Property : the stability depends on the slope of the flow near the fixed point.

In **1D case**: $\frac{dF}{du} < 0 \Rightarrow \text{stable} \quad \frac{dF}{du} > 0 \Rightarrow \text{unstable}$

Intuition: If the derivative is negative, it means that $f(x)$ is decreasing near the fixed point. When the system moves slightly away from x^*, the force acting on the system will bring it back toward x^*.

In **2d case** we can use **linearization** (Taylor expansion of order 1)

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{x} = \begin{pmatrix} F_u & F_w \\ G_u & G_w \end{pmatrix} \boldsymbol{x}$$

$\boldsymbol{x}$ → $\boldsymbol{x} = (u - u_{FP}, w - w_{FP})^T$

Jacobian matrix

$F_u = \partial F/\partial u$

$F_w = \partial F/\partial w$

Recall F is the function ruling u and G ruling w

stability depends on the **eigenvalues of the Jacobian** on the complex plane

Eigenvalue of Jacobian Matrix:
To find the eigenvalues we solve the characteristic equation:

$$J = \begin{pmatrix} F_u & F_w \\ G_u & G_w \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$det \begin{pmatrix} a-\lambda & b \\ c & d-\lambda \end{pmatrix} = 0 \quad \Rightarrow \quad (a-\lambda)(d-\lambda) - bc = 0$$

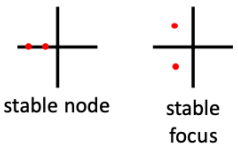
$$\lambda = \frac{(a+d)}{2} \pm \frac{1}{2}\sqrt{(a-d)^2 + 4bc}$$

$$\lambda = \frac{(F_u+G_w)}{2} \pm \frac{1}{2}\sqrt{(F_u-G_w)^2 + 4F_wG_u}$$

There're **5 different possible situations** depending on the real and complex part of the eigenvalues:

Stable fixed point: the real part of the eigenvalues of the linearized system are negative and the complex parte is null

- **Stable node:** both eigenvalues has no imaginary part. The **trajectories** in the phase plane **converge directly to the stable node without oscillating**

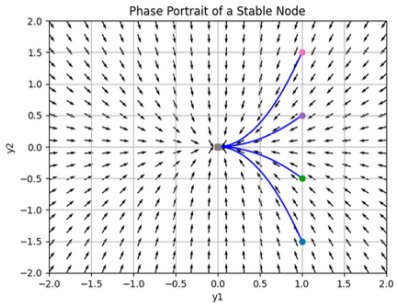


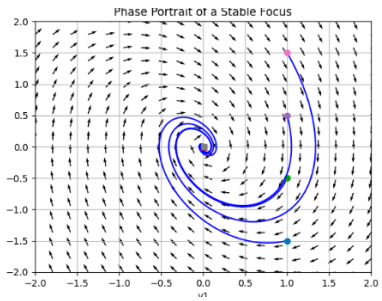
- Stable focus: eigenvalues are complex with negative real part. The **complex part** of the eigenvalues gives rise to a **spiral motion** while the real negative part ensure stability. The **trajectories oscillate around the fixed point** as they converge

$$\lambda_1 + \lambda_2 < 0 \qquad \lambda_1\lambda_2 > 0$$

$$F_u + G_w < 0 \qquad F_uG_w - F_wG_u > 0$$

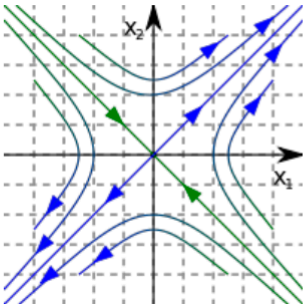
$$\lambda = \frac{(F_u+G_w)}{2} \pm \frac{1}{2}\sqrt{(F_u-G_w)^2 + 4F_wG_u}$$





Unstable fixed point: same as stable but with positive real part:
- **unstable node:** positive real part and no complex part
- **Unstable focus:** positive real part and complex part diverse from 0.

Saddle point: one eigenvalue is positive and the other one is negative, indicating opposite behaviours in the system's dynamics. **Paths approach the saddle point along the stable manifold and recede along the unstable manifold**.



While locally unstable, the saddle point **acts as a critical junction that separates different dynamical regimes of the system.**

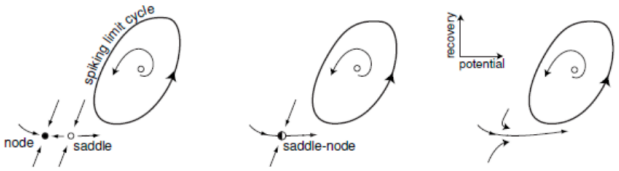
$$\lambda_1\lambda_2 < 0$$

$$F_u G_w - F_w G_u < 0$$

Bifurcations in neurons

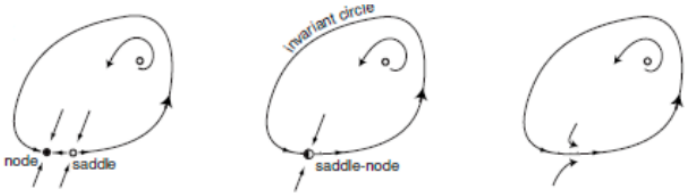
The **type of bifurcation determines the excitable properties of neurons**. Despite we have thousands of different types of neurons there exists only **4 different type of bifurcations**. Given that bifurcations happens when we change the parameters of the systems, we can observe them by changing the value of the injected input current I.

1. Saddle node bifurcations



At rest we can observe a stable fixed point (black) a saddle point white) and a limit circle equilibria that coexists. While the input current increases the black node approaches the white saddle point, and the equilibria annihilate each other and resting no longer exists (the two points will cancel each other out). System dynamics jumps to a limit cycle attractor (tonic spiking).

2. Saddle node on invariant circle



In this case an invariant circle becomes a limit circle attractor. The main difference with the previous bifurcation is that the limit cycle does not exists at rest, it creates when the saddle node annichilite the stable fixed node: one stability at a time in this case (mono-stability).

3. Subcritical Andronov-Hopf bifurcation

Also in this case we have 2 stability at the beginning: a fixed point unstable and a piking limit cycle. The inner unstable attractor become smaller and smaller and then the trajectory jumps to the outer big cycle attractor and the system start iterating on it.

4. Supercritical Andronov-Hopf bifurcation

a stable equilibrium looses stability, giving rise to a limit cycle attractor.

Definitions

- **Bistability:** coexistence of two equilibrium in the system, respectively resting and (tonic) spiking. It is observable in **saddle node bifurcations** and **subcritical Andronov-Hopf bifurcation**
- **Monostability:** no coexistence of resting and spiking

- **Resonators:** neuron exhibits a **damped oscillation** (like in focus stable points) of membrane potential around the resting state. These membrane potential oscillation make resonation response to input possible. It's observable in **Andronov-Hopf bifurcation**.
- **Integrators:** no damped oscillation of the membrane potential

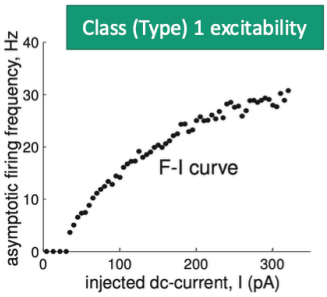
Idea integrators vs resonators: in integrators there's no oscillation around the rest, if the input current is strong enough it fires otherwise no.

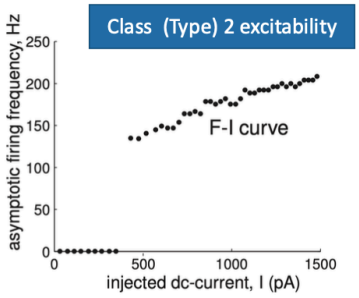
co-existance of resting and spiking states

subthreshold oscillations	YES (bistable)	NO (monostable)
NO (integrator)	saddle-node	saddle-node on invariant circle
YES (resonator)	subcritical Andronov-Hopf	supercritical Andronov-Hopf

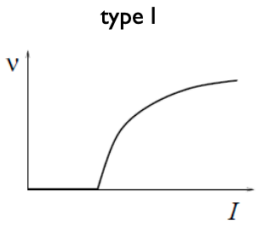
From this we can derive two different **class type excitability** of neurons:

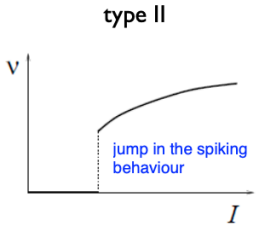
- **class type 1 excitability**: as observable in the current-frequency plot we can observe that the neuron that has this type of excitability fires also for lower input current, with a low spiking frequency. As the input current increases also the firing frequency increases.
- **class type 2 excitability**: in this case the neuron fires only for higher current input but with an high frequency. When the input current is strong enough it shows high frequency spikes, whenever the input current it's. The frequency of firing increase with the input current, but not so much as in class type 1.





We can state that frequency response in **type 1** models is continuous and the oscillation begins with a frequency higher then 0.
Frequency response in **type 2** models has a discontinuity, oscillation begins with non zero frequency and the limit cycle oscillation starts with frequency > 0.



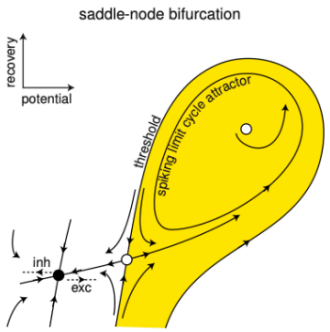


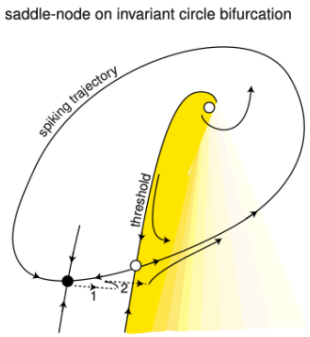
Bifurcations: Geometry of Phase Portraits

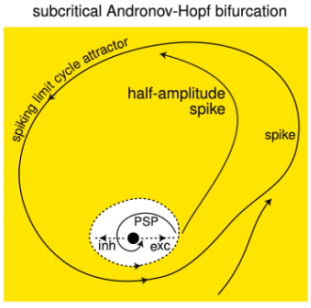
Black circles: stable equilibria (rest)
White region: attraction basin of the equilibrium state
Shaded yellow region: attraction basin of the limit cycle

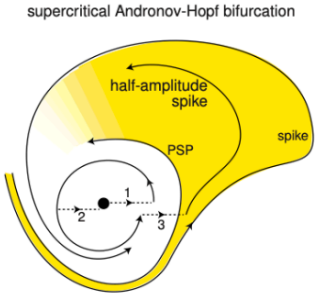
Monostability: right situation (no limit cycle). One stability at the rest, until the bifurcation happens the cycle does not exists

Integrators (top) **vs Resonators** (down):
- Integrators have all or none spikes, resonators have more complex behavior
- Integrators have well defined thresholds, resonators don't

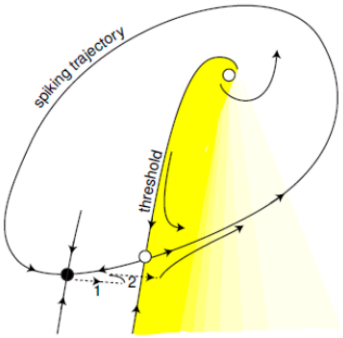




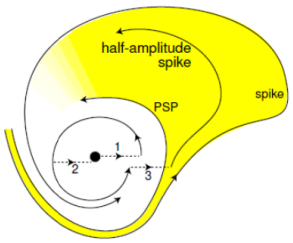




Integrators vs Resonators behaviour

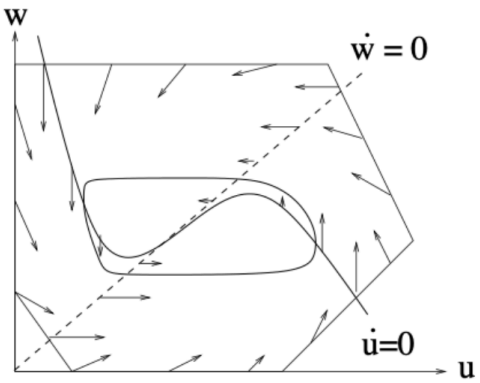


In the **integrator** we can observe that a subsequent input spikes push the state closer to the threshold and eventually make the neuron fire (when stable equilibria and saddle point annihilate each other). The stable manifold of the saddle identifies the **threshold** like behaviour: neuron at rest is in the equilibria black state, as more input current arrives, the neuron moves from equilibria states, coming back to it. If the input current overcome the threshold the system ends un in the yellow attractor region of th cycle and then comes back to the stable state.



In the **resonator** membrane potential has an **intrinsic damped oscillation** (the state spirals toward the stable equilibrium). Response to input depends on the frequency content of the input. In the image we can observe that the neuron receives the first current input (1), initiating a spiral movement around the rest point. Then, it receives a second input (2) while the system is oscillating, causing the trajectory to be attracted toward the black rest point due to the oscillation within the period. However, if the second input is received at position (3), the system — influenced by the damped oscillation — is instead attracted toward the limit cycle, resulting in the neuron firing a spike. There's **no clear threshold in this case**. timing is crucial to observe a spike but also the intensity must be big enough to cause the spike

How to determine if we have a limit cycle in the phase state?



Recall a limit cycle is a periodic attractor or repeller. The analysis of the phase plane can reveal the existence of limit cycles:

- **No-intersection theorem:** trajectories cannot cross one another
- **Poincaré-Bendixson theorem:** bounded solutions of a differential equation can either remain fixed or oscillate periodically

How to find them (*Poincaré-Bendixson theorem)***:** if we can construct a **bounded box around a fixed point**, then if the fixed point is unstable (so directions point toward outside this point) and the directions on the bounded surface points toward the inside of the surface, then we have a stable limit cycle between the unstable fixed point and the bounding surface

Example: FitzHugh-Nagumo model

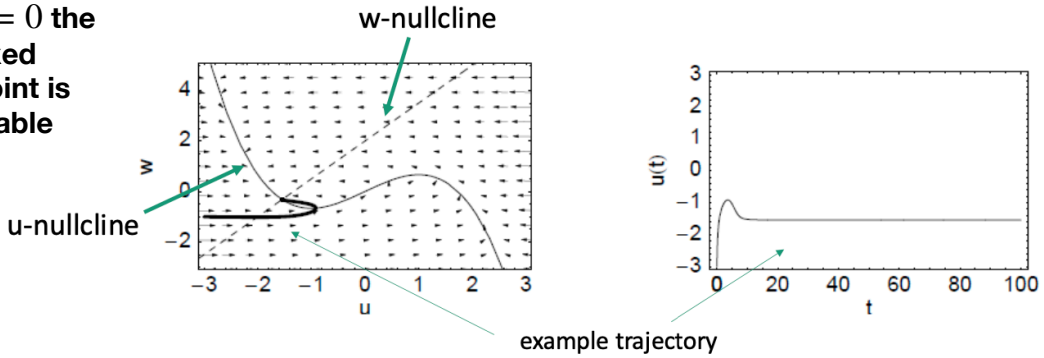
Recall:

$$\frac{du}{dt} = u - \frac{1}{3}u^3 - w + I$$
$$\frac{dw}{dt} = \epsilon(b_0 + b_1 u - w)$$

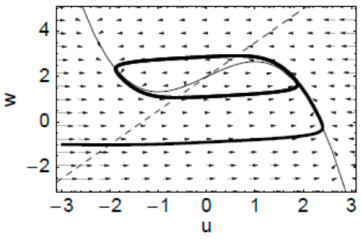
We can observe that: if we give a small perturbation to my state, te trajectory will be attracted trough the stable fixed point (intersection of the two nullclines).

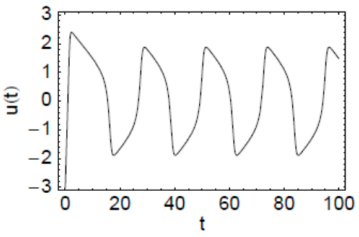
If I is increased a lot, the U nullcline is shifted upwards (see the formula) and for this reason the fixed point can become unstable

$I = 0$ the fixed point is stable



For $I > 0$ the fixed point is shifted upwards, loss stability and by the Poincaré-Bendixson theorem **there exists a limit cycle**



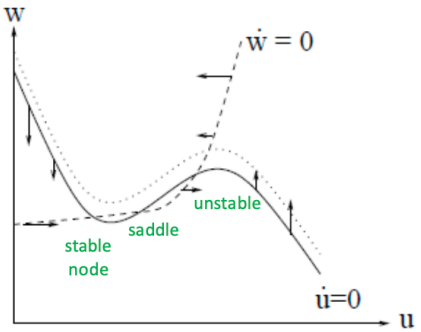


For increasing value of I bifurcation occurs (Hopf bifurcation). There is only 1 fixed point, but it looses stability the real part of at least one eigenvalue changes sign (**type 2 model**)

Bifurcation type 1 models

With $I = 0$ there are 3 fixed points: stable (node), saddle, unstable.

Increasing the I the u-nullcline moves upward: the node and the saddle get closer and closer until they disappear as u nullclines will no more intersect w nullclines in that part of the space. In this case we can observe an unstable fixed point with bounded flow: limit cycle (oscillation has increasing frequency)
(**Nota:** fixed and saddle point becomes closer and close untill they annihilate each other as previously seen)



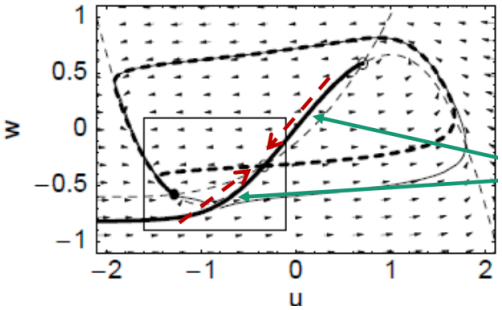
Note: the w-nullcline has a curvature

Excitability

Suppose to apply an external input of impulse type to a resting neuron. This input **shifts the system dynamics horizontally to the right:** $\dot{u} = \frac{1}{\tau}(F(u,w) + RI)$ and $\dot{w} = \frac{1}{\tau_w}(G(u,w))$**.** How does the system return to equilibrium? Is there a threshold behavior?

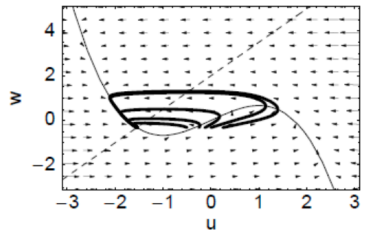
Type 1 case:

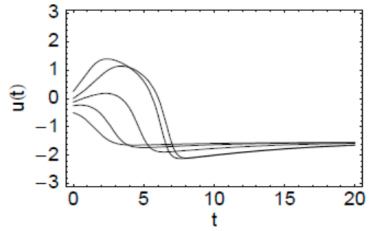
Recalling that trajectories cannot cross one another and that the eigenvectors of the saddle identify a manifold of stability. The stable manifold is a "**forbidden line**": when the input stimulation is sufficient to push the state to the right of the stable manifold (**threshold behaviour**) the state is forced to a long excursion to pass over the unstable fixed point (**all-or-none spikes**)



Type 2 case:

No stable manifold, no "forbidden" line. There is a continuum of trajectories (**no threshold**) and each small perturbation can generate different trajectories.
BUT: if u dynamics are much faster than those of w a threshold-like behavior can emerge.





THEORY: Numerical solution of differential equations

This section contains an overview of formal methods for solving systems of **linear ordinary** differential equations (ODEs). Concerning **non-linear** ODEs, analytical solutions are very difficult to find or do not exist.

Numerical integration: find an iterated map that approximates the flow (meaning approximate the continuous system with its discrete counterpart)

Let's consider a 2-dimensional flow:

$$\frac{dx}{dt} = f(x, y) \qquad \frac{dy}{dt} = g(x, y) \qquad \text{with initial condition given by } (x_0, y_0)$$

Our aim is to find an **iterative map**: $X_{n+1} = F(X_n, Y_n), \quad Y_{n+1} = G(X_n, Y_n)$
Such that $x(t) \approx X_n$ and $y(t) \approx Y_n$

With $t = nh$ and h being a **small** time step for integration. We want that the solution of the map approaches the solution of the flow for h sufficiently small

Euler Method
The simplest numerical method for solving ODEs, it directly follows from the definition of a derivative:

$$\frac{dx}{dt} = \lim_{h \to 0} \frac{x(t+h) - x(t)}{h}$$

We hence obtain:

$$X_{n+1} = X_n + hf(X_n, Y_n) \qquad Y_{n+1} = Y_n + hg(X_n, Y_n)$$

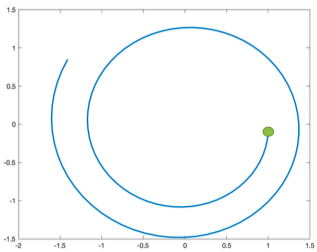
Limitations:
- The method **assumes that** f **is approximately linear over small intervals** h
- The accuracy depends on h: smaller h improves accuracy but requires more steps
- **Global error:** this method is explicit and first-order, meaning errors accumulate as $O(h)$
- **Local truncation error**: the error on each specific time step is quadratic in h, $O(h^2)$

Example: circular motion:

We want to approximate a circular motion. The motion is described by: $\frac{dx}{dt} = y$ and $\frac{dy}{dt} = -x$.

The radius depends on the initial condition.

Solving with Euler method ($X_0 = 1, Y_0 = 0, h = 0.1$)*:* on successive iterations the radius continues to grow and the orbit of the map **spirals outward**. This approximation is to enough good (h is too high).



Leap-frog method:

Instead of X_n use directly the new value of X (i.e. X_{n+1}) when iterating Y:

$$X_{n+1} = X_n + hf(X_n, Y_n) \qquad Y_{n+1} = Y_n + hg(\mathbf{X}_{n+1}, Y_n)$$

The result is accurate only to the first order, but in some cases errors cancel out. It's **typically more accurate than the Euler method**.

(Part 1 Lecture 4) FORMAL MODELS & IZHIKEVICH MODEL

So far, we have studied **conductance-based models**. In these models, we do not explicitly state that a neuron spikes when its potential exceeds a threshold. Instead, the neuron's membrane potential dynamics are represented by differential equations that describe both the potential and the behavior of membrane ion gates.

In this section we're gonna **simplifying neuron models** eliminating parts of the biophysical model in which spiking is not explicitly modeled, but we have to enforce it.

FORMAL THRESHOLD MODELS

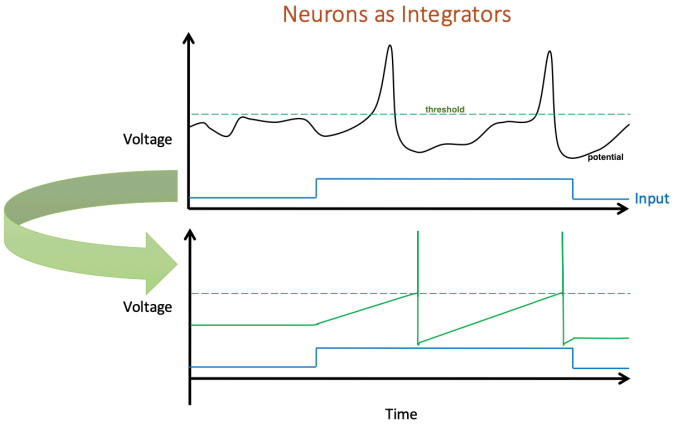
In this models, **spikes are stereotyped events** (every spikes are considered qual) that occur when the membrane potential **crosses the threshold** from below:

This means: $t(f) : u(t^{(f)}) = \theta$ and $\left.\frac{du(t)}{dt}\right|_{t=t^{(f)}} > 0$

So we want that the the membrane reach the threshold and the derivative of the membrane at the firing time is positive (so the membrane potential is increasing)

Spikes are fully characterized by their **firing time.** These models describe only the **sub-threshold dynamics**, focusing on what happens before a spike while ignoring spike and post-spike behavior.

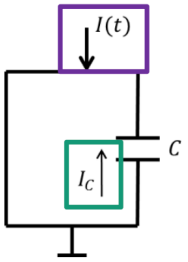
In the image we can observe a comparison between **conductance-based models (above)** in which the model focuses on neuron potential dynamics and the **formal threshold models (below)** where the model focuses on sub threshold behaviours, and when the potential reaches the threshold the neuron fires. In both cases the neuron is an **integrator** as it integrates input from pre-synaptic neurons



Integrate and Fire Model

It's the most simple case: **all membrane conductances are ignored.**

The equivalent circuit only contains a capacitor. The **input current is a parameter**: external aspect that we don't model in the dynamic of the system but it has an effect on the system itself. The only **variable** of the system is the membrane potential u



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

only the current input is considered in the equivalent circuit of the neuron

- Spikes are formal events characterized by the firing time $t(f) : u(t^{(f)}) = \theta$
- After the spike the potential is reset to u_r: $\lim_{t \to t^{(f)+}} u(t) = u_r$
- **Absolute refractory period:** after the spike, the integration is suspended for Δ^{abs}. The neuron cannot immediately fire again, mimicking biological neurons and preventing excessively high firing rates.

Summary:

- $\frac{du(t)}{dt} = \frac{I(t)}{C}$ — 1D ODE of the membrane potential
- $t^{(f)} : u(t^{(f)}) = \theta$ — annotates time of spikes (we know when the model triggers spike)
- $\lim_{t \to t^{(f)+}} u(t) = u_r$ — reset condition: reset potential to resting value ($u_r = 0$ often)

We can **numerically solve** the system above:
Suppose a constant input current I_0 is applied (e.g. an EPSP, positive excitatory), and the last spike occurred at time $t^{(1)}$, the time course (andamento temporale) of the membrane potential can be obtained by integration in the time interval $t^{(1)}$; t

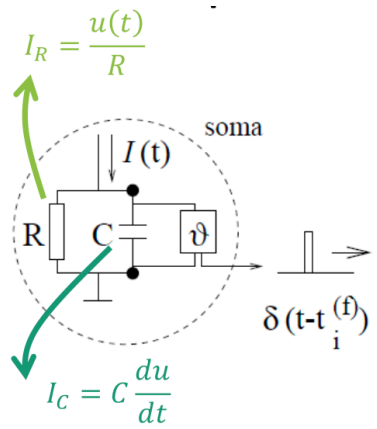
$$u(t) = \int_{t^{(1)}}^{t} \frac{I_0}{C} ds = \frac{I_0}{C}(t - t^{(1)})$$

This represents the amount of increase in membrane potential over time (i.e. the linear function that increases between two spikes in the previous image)

Properties:
- all or none spikes model with a well defined threshold
- Relative refractory period Δ_{abs}
- **Excitation vs Inhibition:** clear distinction, excitatory PSP push the membrane potential closer to the threshold (inhibitory PSP do the opposite)
- **Type I / Class 1 excitability:** the firing rate encodes continuously the strength of the input (the stronger the input, the larger the firing rate)
- Good model for **integrators**
- **Limitation:** many neuro-computational features cannot be simulated
- Good for analysis
- Fixed point is a **node**: unique stable equilibrium at rest
- Too simple for large scale simulations

Leaky Integrate-and-Fire Model

The membrane conductance is modeled as a single leakage term (termine di perdita). This works under the **assumption** that **the conductances are all constant** (they not depends on the membrane potential value). This is true for small fluctuations around the resting membrane potential.



The **equivalent circuit** consists in a **capacitor in parallel with a resistor.**

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

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

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

$\tau = RC$ is the **membrane time constant**

First equality is derived from the fact that sum of external current must be = to sum of internal currents.

Notice also in this case that the input current is a parameter (external aspect that i don't model in the dynamic of the system but it has an effect on the system itself).

Also as in previous case we consider both:

- $t(f) : u(t^{(f)}) = \theta$ annotates time of spikes (we know when the model triggers spike)
- $\lim_{t \to t^{(f)+}} u(t) = u_r$ reset condition: reset potential to resting value ($u_r = 0$ often)

As previous and with the same assumptions, it's possible to **integrate** over time to know the potential value over time:

$$u(t) = RI_0 \left(1 - e^{-\frac{t - t^{(1)}}{\tau_m}}\right)$$

First-Order Linear Differential Equation

Recalling (for deriving previous)

$$\begin{cases} y'(t) + a(t)y(t) = f(t) \\ y(t_0) = y_0 \end{cases}$$

solution: $y(t) = y_0\, e^{-A(t)} + e^{-A(t)} \int_{t_0}^{t} f(s) e^{A(s)} ds$

where: $A(t) = \int_{t_0}^{t} a(s) ds$

Also useful to remember:

$$\int e^{f(t)} f'(t) dt = e^{f(t)} + c$$

Computing the firing rate: when will next spike occur?

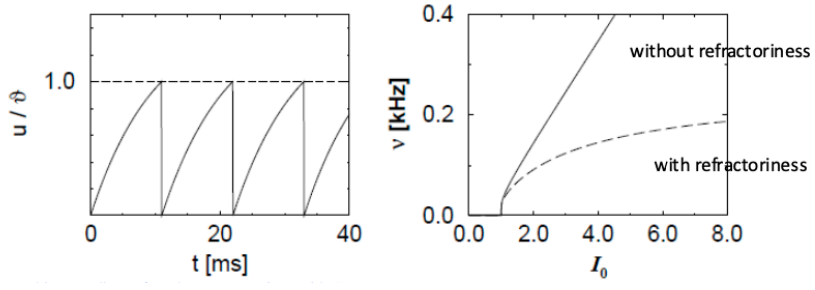
Hint consider that if the first spike is at $t^{(1)}$, you want to compute it for $t = t^{(2)}$, so the second time in which the potential reaches the threshold theta. Computing it, considers the absolute refractory period Δ^{abs}.

$$u(t^{(2)}) = \vartheta = RI_0 \left(1 - e^{-\frac{T}{\tau_m}}\right) \qquad T = t^{(2)} - t^{(1)}$$

$$T = \Delta^{abs} + \tau_m \ln\left(\frac{RI_0}{RI_0 - \vartheta}\right)$$

$$\nu = \left[\Delta^{abs} + \tau_m \ln\left(\frac{RI_0}{RI_0 - \vartheta}\right)\right]^{-1}$$

Firing rate (with refractory period)



Why simpler models?

Conductance-based models, e.g. Hodgkin-Huxley, are biologically accurate:
- each variable and parameter has a well defined biophysical meaning
- variables (u, m, h) and parameters (reverse potential, max conductance, external current in input) can be experimentally measured

However, **measurements may not be accurate**, and the biologically observed neuronal behavior could differ. A small measurement error can lead to significant discrepancies in the system.

To address this, we can **relax the requirement for strict biological plausibility**, allowing variables and parameters to not directly reflect biophysical entities. With careful parameter selection, a simple model can still generate a wide range of neuro-computational features from a dynamical systems perspective.

Simpler models are faster to be computed/simulated while biologically detailed models, like the Hodgkin-Huxley model, are expensive and large simulations are infeasible. In this scenario, simpler model allows **larger simulations become possible**.

We need a **trade-off between biological plausibility and simulation efficiency**

Resonate and fire model

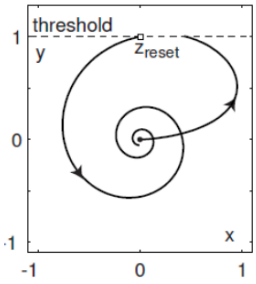
it's a **two-dimensional extension** of the integrate-and-fire model. Unlike the traditional Integrate-and-Fire model, which focuses on the threshold to determine spikes, R&F models focus on the oscillatory behaviour of neuronal membranes.

Added variable w: recovery variable representing slow *K* activation (*n*) or *Na* inactivation (*h*)

$$C\dot{u} = I - g_L(u - E_L) - w$$
$$C\dot{w} = (u - V_{1/2})/k - w$$

When u reaches a threshold the **neuron fires** (spike) **and** then **resets**:

$$\text{if } u \geq E_{thresh}$$
$$spike$$
$$u \leftarrow u_{reset}$$
$$w \leftarrow w_{reset}$$



damped oscillation around the resting state

$$\dot{z} = (b + i\omega)z + I$$

In this model we can observe **damped sub-threshold oscillation around the resting state** that are modeled using complex numbers

This allows to model **rebound spikes:** when a neuron receives a negative input current, and when the negative input is "removed" the neuron spikes

The unique fixed point is a **stable focus** around which the model oscillates

This model is good for describing **resonators** and **type II / Class 2 excitability.**

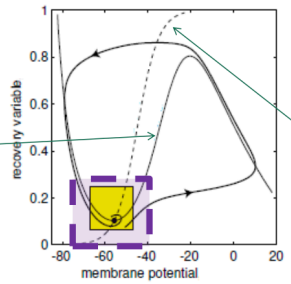
Properties:
- Linear differential equations
- Fixed-value firing threshold
- **Reset:** after the spike membrane potential and recovery variable are reset to the reset state
- the **equilibrium is a stable focus** for resonate-and-fire (a node for integrate-and fire)
- Good for analytical purposes thanks to their **efficency**
- Not good for large-scale simulations as simulations can be hampered by the limitations of the models

Izhikevich General Model

Izhikevich Model is a two dimensional spiking models with:

- a **fast voltage variable**
- a **slow recovery variable** (activation of K or inactivation of Na)

the fast variable has an N-shaped nullcline



the slower variable has a sigmoid nullcline

intersection point in the yellow region. here we got a fixed point. Behaviour of the w is proxy linear

model equations:

$$\begin{cases} \dot{u} = I + u^2 - w \\ \dot{w} = a(bu - w) \\ \text{if} \quad u \geq 1 \\ \quad u \leftarrow c, w \leftarrow w + d \end{cases}$$

reset condition: not a threshold

Where:

u is the membrane potential (*quadratic equation*)

w is the recovery variable (linear *equation*)

Note: the reset condition is NOT a spiking condition (spike is encoded in the ODEs)

[!!] nota: it's not a FORMAL threshold model, threshold is not well defined in system equations

Properties:

- Based on (*a, b,* I) it can be an **integrator** or a **resonator**
- Parameters (*c, d*) affect the **after-spike transient behavior**
- The value of *a, b, c, d* are neuron-type dependent

Fitting the spike initiation dynamics in cortical neurons:
(Cortical neurons are neurons located in the neural cortex that is, the outermost part of the brain)

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

Hint: the recovery variable will simulate the variable *n* and *1-h* of initial Hodgkin-Huxley model

Let's deep into parameters "semantic" intuition:

- **Parameter *a***: represents the **time scale** of the recovery variable w (0,02)
- **Parameter *b***: represents the **sensitivity** of the recovery variable to fluctuations of the membrane potential (larger b the more w is influenced by the value of u) (0,2)
- **Parameter *c***: after-spike reset value of the membrane potential (-65)
- **Parameter *d***: after-spike reset of the recovery variable (2)
- **Parameter** I: the input current

[!!] a, b, c, d has **not** a well defined biophysical meaning!

With Izhikevich Model is **it possible to model the most fundamental classes of firing patterns**.

The most fundamental classes of **firing patterns observed in the mammalian neocortex** are just 6 (respectively 3 excitatory and 3 inhibitory)

(1) Regular Spiking (RS)

The most common dynamics in the cortex, when stimulated by a constant input the neuron fires a few spikes with short period, then the period increases

(2) Intrinsically Bursting (IB)

The neuron fires a stereotypical **burst** followed by repetitive single spikes. A **burst** consist of a train of spikes with a very low period. This class of spikes is typically used to reduce neuronal noise

(3) Chattering (CH)

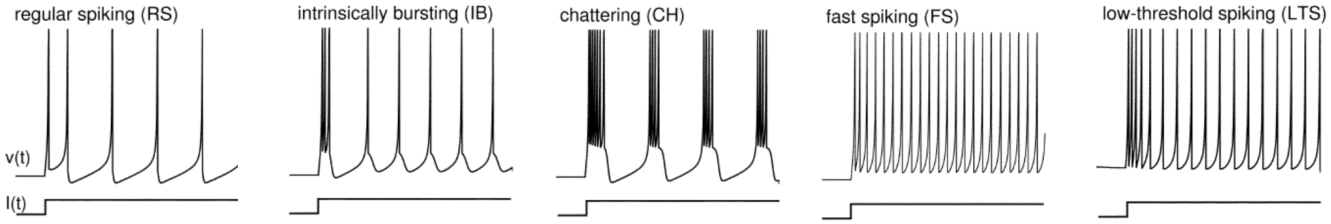
The neuron fires a train of stereotypical bursts

(4) Fast Spiking (FS)

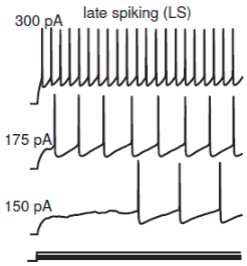
The neuron fires periodic trains of spikes at high frequency without adaptation (no adaptation means with no transient behaviour before this frequency of spikes)

(5) Low-threshold Spiking (LTS)

The neuron fires periodic trains of spikes at high frequency. Strong spike frequency adaptation and low firing thresholds



(6) Late Spiking (LS)



Voltage ramp near the threshold (neuron's membrane potential increases slowly and gradually before reaching the threshold needed to generate a spike) with **delayed spiking activity** (meaning they don't fire immediately when stimulated) and **sub-threshold oscillation** (where the membrane potential fluctuates without necessarily triggering a spike)

Note on adaptation: it refers to a neuron's tendency to change its firing rate over time in response to a constant stimulus. Typically, it means a decrease in firing rate after an initial high-frequency response.

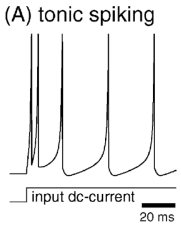
Firing regimes of the Thalamo-cortical neurons

- **Tonic firing**: the neuron is depolarized at rest and then when it receives an input it spikes
- **Rebound burst**: when a negative input current is removed the neuron fires and we observe a burst

IMPORTANT PROPERTY: The Izhikevich model can simulate all of them

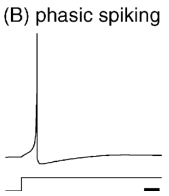
Neuro-computational Features

(A) Tonic Spiking



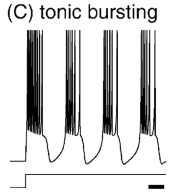
Normally neurons are typically quiescent but can fire a spike when stimulated (they're excitable). While the input is on the neuron continues to fire a train of spikes-

(B) Phasic spiking



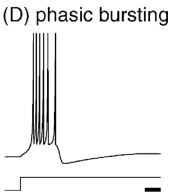
In this feature the neuron spikes one single time at the beginning of the input stimulation and after a, hyper-polarization it return to rest.

(C) Tonic bursting



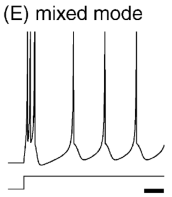
Like the tonic spiking but in this case, when stimulated, the neuron responds with a train of periodic bursts

(D) Phasic bursting



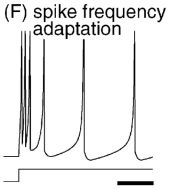
The beginning of stimulation is signalled by a single burst (like phasic spiking but with a burst)

(E) Mixed mode



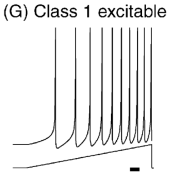
In this case the reaction to the input step current is a burst, followed by a spiking until the stimulation is positive.

(F) Spike Frequency Adaptation



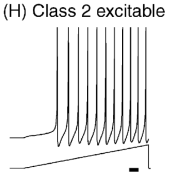
The neuron fires periodic spikes with **decreasing frequency.** So few spikes with very high frequency and then adaptation to a frequency

(G) Class 1 excitability



Encode the strength of the input into the firing rate: firing rate increases as the input current increases

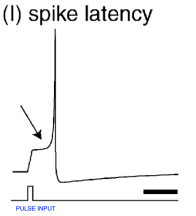
(H) Class 2 excitability



Neurons are either quiescent or fire high frequency spike trains. In this case the firing rate is a poor predictor of the strength of stimulation

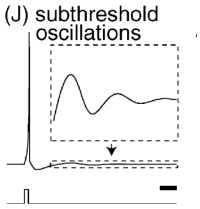
Idea: firing rate at the beginning is very high, it inreases as the input increases but not so much

(I) Spike latency



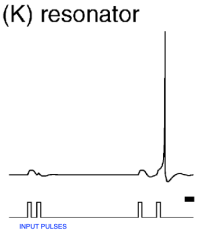
The neuron fires with a delay that depends on the strength of the input signal. In this case the input is a **pulse input**.

(J) Subthreshold oscillation



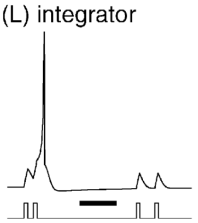
After the spike we can observe oscillations around the fixed points in the membrane potential. In this case the input is a **pulse input**.

(K) Resonator



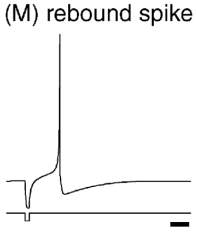
Neurons having oscillatory potentials can respond selectively
to inputs having a frequency content similar to the frequency of subthreshold oscillations. The neuron spikes depends on the frequency in between the input pulses: the farer the pulses the more likely it fires

(L) Integrator



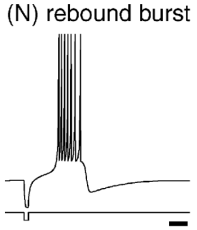
Neurons without oscillatory potentials act as integrators. In this case the frequency (distance) between pulses can cause a spike or not (nearest pulse fires the neuron, not near pulse not), so it does not strictly depend on the input current value

(M) Rebound spike



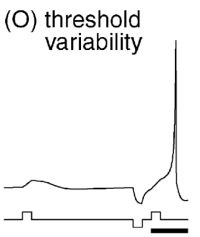
The neuron receives and then is released from an inhibitory input. When the negative inhibitory input is released the neuron spikes

(N) Rebound burst



After the release of inhibitory input the neuron can fire a rebound burst

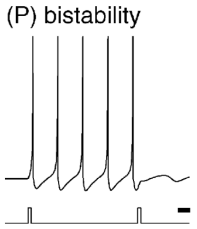
(O) Threshold Variability



Biological neurons have a variable threshold: the value of the threshold depends on the previous activity of the neuron. A brief inhibitory input can lower the value of the threshold

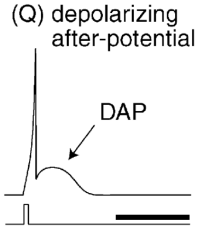
In this case: neuron with an input pulse will not determine a spike, instead if i give a negative input but then this input is followed by the same pulse but positive, it spikes because of the threshold variation

(P) Bistability



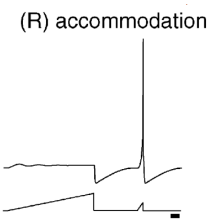
Since in this case we have bistability, both resting and spiking co-exists. A PSP can shift the neuron operational mode from one modality to the other states (**bifurcations**). The input pulse push the system dynamics in a limit cycle attractor and then another pulse will come back the behaviour of the system to rest situation

(Q) Depolarizing After-Potential



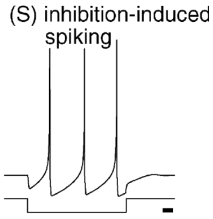
After the spike the membrane potential typically exhibits after-hyperpolarization (refractoriness), but in some cases a **depolarized after-potential is observed.** We say that the neuron is super-excitable

(R) Accommodation



Neurons are very sensitive to brief pulses but could not be sensitive to slowly increasing inputs: the slower longer ramp does not elicit a spike, while the brief sharp ramp does

(S) Inhibition-induced Spiking



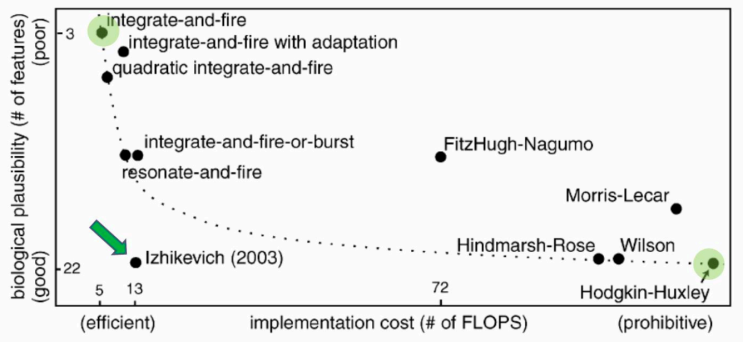
The neuron is resting when there is no input and fires periodic spikes when stimulated by an inhibitory input signal

(T) Inhibition-induced Bursting

The neuron responds to a prolonged inhibitory input with a train of burst (as previous but with bursting)

Neuron Models – Biological Plausibility vs Cost

This plots shows the comparisons of neuron models in terms of efficiency (on x axes represented as # of FLOPS) and in terms of biological plausibility (how close its behaviour is to the cortical behaviours in terms of a dynamical system perspective, it's not how well it approximate the neuron considering a biopshycal point of view of the neuron).



Izhikevich model is the **best model** in terms of both **plausibility** and **efficency** while integrate and fire is too poor in representations while Hodgkin-Huxley is too expensive.

(Part 1 Lecture 5) NETWORKS OF NEURONS

Extensive connectivity among neurons is a major characterization of the brain computation. We're gonna focus on **neocortical circuits:** layered recurrent circuits.

So far we've observed that:
- neurons lie in 6 layers
- Connectivity occurs among cortical columns structures

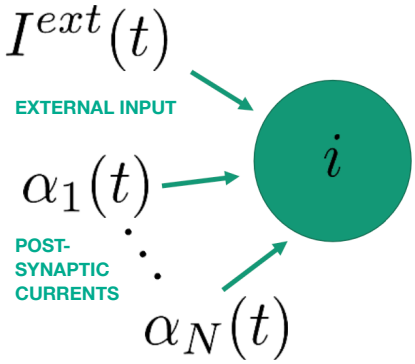
There're two type of connections:
- **feed-forward connections**: signal pathways to higher stages of computation
- **recurrent connections:**
  - signal feedbacks interconnecting neurons at the same stage of computation
  - top-down interconnections between areas in different stages of computation

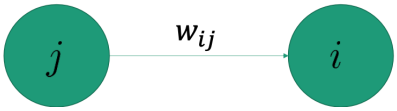
Our aim is to **interconnect spiking neurons** in a biologically plausible fashion. In order to do that, we can model neurons as we prefer using a mathematical model of spiking neurons (Hodgkin-Huxley, Integrate-and-fire, Leaky Integrate-and-Fire, Izhikevich…). Typically in MLP and DeepLearning firing-rate models are commonly used.

Networks of Spiking Neurons

Neurons are **organized in a pool of connected units** and are coupled through **synaptic connections**.

Each neuron receives in **input** two informations:
- the external input (I^{ext})
- The post-synaptic current generated by the pre-synaptic neurons to which it is coupled





The efficacy of each synaptic connection (synaptic strength) is modeled by a **synaptic weight** W_{ij}

Synaptic current is modulated by the **synaptic weight** —> $w_{ij}\alpha(t - t_j^{(f)})$

Where t is the current time and $t_j^{(f)}$ is the last time firing of pre-synaptic neuron j

The amplitude and sign of the current generated are determined by W_{ij} :
- $W_{ij} > 0$ for **excitatory synapses** (EPSP)
- $W_{ij} < 0$ for **inhibitory synapses** (IPSP)

Patterns of connectivity

1. Fully connected homogeneous networks

In this type of network, all neurons are identical (eg resonator, integrators..) and receive the same external input. The strength of synaptic interaction is uniform and is computed as $w_{ij} = \frac{J_0}{N}$, where J_0 is total synaptic strength and we divide this term by the number of neuron.

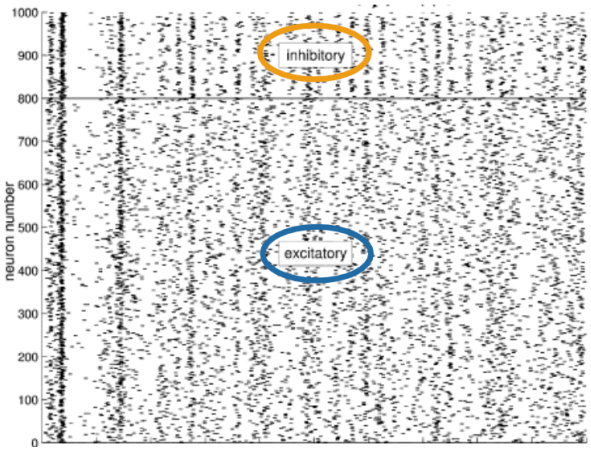
2. Sparsely connected neurons

There exists two types of sparsely connected networks:

- **Fixed coupling probability**: the probability of a connection between 2 neurons inside a cortical column is $p \approx 10\%$. This value is determined through biophysical experiments.
- **Fixed number of presynaptic partners:** each neuron is connected to *C* others. In this scenario neurons are connected more densely to nearest neurons and this density of connections tend to decrease as we consider more farer neurons

The networks are composed by **two populations**:

- A pool of **inhibitory units ≈ 20%** (e.g. modeled as fast spiking neurons). They' re connected with negative weights
- A pool of **excitatory units ≈ 80%** (e.g. modeled as regular spiking neurons). They' re connected with positive weights



The right plot is a picture of what happens in my network at each time step, how its behavior evolves over time:

- **x axes**: time
- **y axes**: size of the networks
- **One row in the plot**: the activity of a neuron over time. White dot means the neurons at that time not spikes, black dot means that the neuron spikes.

Synaptic Plasticity

As already said, each synapse is characterized by a strength w_{ij} that **modulates the response amplitude.** In biological neurons the **response amplitude are not constant**, but might change over time (this means learning, memory, …)

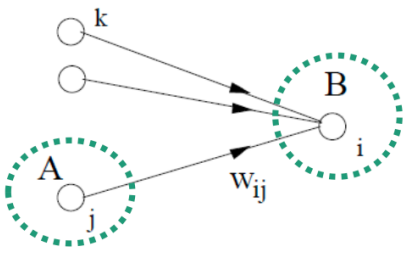
Based on the stimulation pattern we can observe:

- **Long-term Potentiation (LTP)**: a stimulation pattern leads to a persistent increase of the synaptic transmission efficacy (ie the strength/value of the weight increases)
- **Long-term Depression (LTD)**: a stimulation pattern leads to a persistent decrease of the synaptic transmission efficacy (ie the weight of the post synaptic neuron is decreased)

In conventional ML approaches **synaptic weights are considered as parameters**. Synaptic weights can be **adjusted to optimize the performance of the network for a given computational task.** In this context, we refers to **learning** as the process of parameter adaptation and to **learning rule** as the algorithmic procedure that is used to perform learning.

HEBBIAN LEARNING

what drives the weight adaptation is the changes in synaptic transmission efficacy that are due **to correlations** of pre- and post-synaptic activity

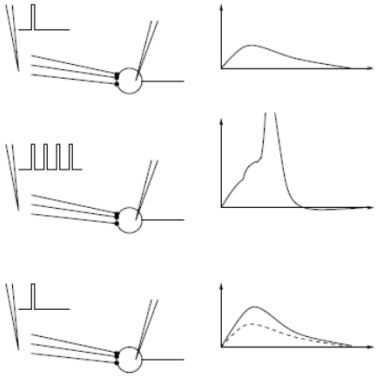


Idea: *"when an axon of cell A is near enough to excite cell B or repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both the cells such that A's efficiency, as one of the cells firing B, is increased"*

Hebbian learning paradigm: modifications in the synaptic **transmission efficacy are driven by** correlations in the firing activities of pre- and post-synaptic neurons

Long-term Potentiation

1st observation: In this experiment electrodes are placed in the post-synaptic neuron (intracellular) and on pre-synaptic fibers allowing to measure the input current and the response of the post-synaptic neuron:



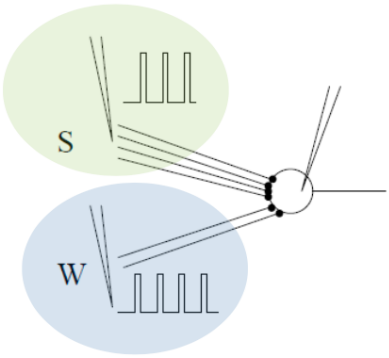
1. small pulses are applied to the presynaptic fibers and the postsynaptic response is measured
2. a stronger stimulation is applied to the presynaptic fibers to evoke a postsynaptic potential
3. finally the same small pulse input is applied to the presynaptic fibers and the measured postsynaptic response is increased

Note: integration depends on the dynamic of the neuron

2nd observation: Both pre-synaptic and post-synaptic spiking activity is required for LTP

Let's consider this scenario with this three different situations: we've two separate input pathways: S for strong stimulations and W for weak stimulation.

1. Stimulation along **S alone:**
   - postsynaptic spike
   - no LTP on the W pathway (W is not increased because they not participate to the spike of post-synaptic neuron)
2. Stimulation along **W alone:**
   - no postsynaptic spike
   - no LTP on the W pathway (the post-synaptic neuron does not fires, so W is not increased)
3. Stimulation along **both S and W:**
   - postsynaptic spike
   - LTP along the W pathway (W is now contributing to post-synaptic neuron spike so also W is increased)



Time of Spikes

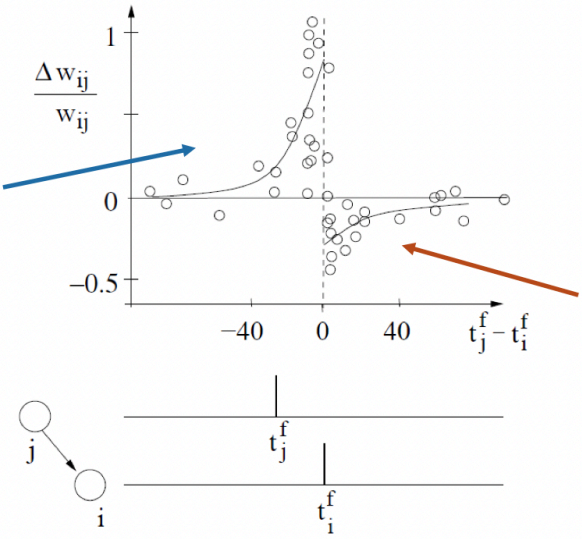
The timing of pre- and post-synaptic spikes plays a major role in weight adaptation. The change in the synaptic efficacy is a function of the spike-time differences

Given neurons i and j the spike time differences is computed as: $t_j^{(f)} - t_i^{(f)}$

Where $t_j^{(f)}$ is the time of last **pre-synaptic** spike and $t_i^{(f)}$ is the time of last **post-synaptic** spike

Spike-time-dependent plasticity

The synapse is strengthened if the presynaptic spike occurs shortly before the postsynaptic neuron



The synapse is weakened if the presynaptic spike occurs shortly after the postsynaptic spike

x axes: spike time difference $t_j^{(f)} - t_i^{(f)}$

y axes: amount of increase of the synaptic connection (derivative of weights wrt to ij)

In both cases of synapse weakening and strengthening, **the smaller the distance between the last spikes** of the presynaptic and postsynaptic neurons, **the greater the update effect** (whether it's more strengthened or more weakened).

Now, let's translate the previous observation into an **algorithm to model the strength of the connections between neurons**.

- Changes in synaptic weights are determined by:
  - **activity-independent decay** of the synaptic strengths trough time
  - **pre-** and **post-synaptic activities**

Effect of Pre-synaptic spikes

- **non-Hebbian presynaptic term:** each pre-synaptic spike can trigger a change in the synaptic efficacy even when no post-synaptic potentials are generated
- **Hebbian (correlation) presynaptic LTP:** increasing in the synaptic efficacy is related to the last spiking time of the post synaptic neuron

$$\frac{\mathrm{d}}{\mathrm{d}t} w_{ij}(t) = S_j(t) \left[a_1^{\mathrm{pre}} + \int_0^{\infty} a_2^{\mathrm{pre,post}}(s)\, S_i(t-s)\, \mathrm{d}s \right]$$

the amount of change when a presynaptic spike follows a postsynaptic fire after a delay s — can be also negative

$$S_j(t) = \sum_f \delta(t - t_j^{(f)})$$
$$S_i(t) = \sum_f \delta(t - t_i^{(f)})$$

spike train of pre and post synaptic (j is pre)

Effect of Post-synaptic spikes

- **non-Hebbian postsynaptic term:** each post-synaptic spike can trigger a change in the synaptic efficacy even when no pre-synaptic potentials are generated
- **Hebbian (correlation) postsynaptic term:** another change in the synaptic efficacy is related to the last spiking time of the presynaptic neuron

pre-synaptic term

$$\frac{\mathrm{d}}{\mathrm{d}t} w_{ij}(t) = a_0 + S_j(t) \left[a_1^{\mathrm{pre}} + \int_0^{\infty} a_2^{\mathrm{pre,post}}(s)\, S_i(t-s)\, \mathrm{d}s \right] + S_i(t) \left[a_1^{\mathrm{post}} + \int_0^{\infty} a_2^{\mathrm{post,pre}}(s)\, S_j(t-s)\, \mathrm{d}s \right]$$

activity independent term

amount of change when a presynaptic spike is followed by a postsynaptic fire with delay

post-synaptic term

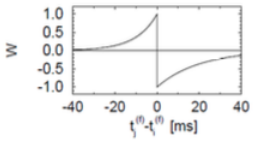
Nota: just one variable, we do not need leap frog method

The two Hebbian terms **define the extent of the learning window:**

- $a_2^{post,pre}$ gives the amount of change when a presynaptic spike is followed by a postsynaptic fire with delay s
- $a_2^{pre,post}$ gives the amount of change when a presynaptic spike follows a postsynaptic fire after a delay s

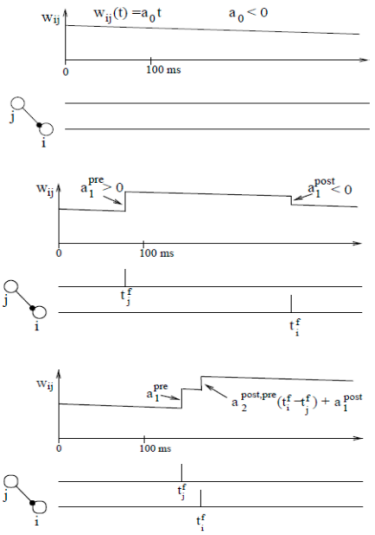
we can model this two terms with a function: $$W(s) = \begin{cases} a_2^{\text{post,pre}}(-s) & \text{if } s < 0\,, \\ a_2^{\text{pre,post}}(s) & \text{if } s > 0\,, \end{cases}$$

A typical choice is in exponential form: $$W(s) = \begin{cases} A_+ \exp[s/\tau_1] & \text{for } s < 0\,, \\ A_- \exp[-s/\tau_2] & \text{for } s > 0\,, \end{cases}$$



Where A and τ are parameters: A^+ is the maximum value of the increase (1 in the plot) and A^- is the maximum value of the decrease (-1 in the plot)

Following this model, three different situations can happens:



1. an **activity-independent term** causes a constant decrease in the synaptic weight (effect of a_0)
2. pre- and post-synaptic spikes that occur quite distant in time determine **non-Hebbian modifications** to the synaptic weight, first slight increase and then decrease due to post-synaptic neuron firing
3. pre- and post-synaptic spikes occurring close in time determine a Hebbian modification (weigh potentiation)

LIQUID COMPUTING

LSM is a **dynamic computational model** designed to process temporal information.

3 generation of neuron models:

1. **First Generation**: McCulloch-Pitts neurons, based on perceptrons and threshold gates with digital output (0/1)
2. **Second Generation**: firing-rate models (the output can be interpreted as the firing rate of a biological neuron). Neuron are models based on activation functions (sigmoid, linear saturated, rectification, …) with continuous output
3. **Third Generation**: in this generation **timing of single action potential** is used to encode information. Those models uses spiking neurons (e.g. integrate-and-fire models) but with simplified models of action potential generation: they try to simulate dynamic behaviour of biological neurons (e.g. modeling fast activation/slow inactivation of Na^+ channels). These models are more complex and more difficult to train but more computational powerful

Neuroscience: it's a research tool to validate the models of brain functioning. It's useful to explain and do predictions on the way in which biological neural networks operate

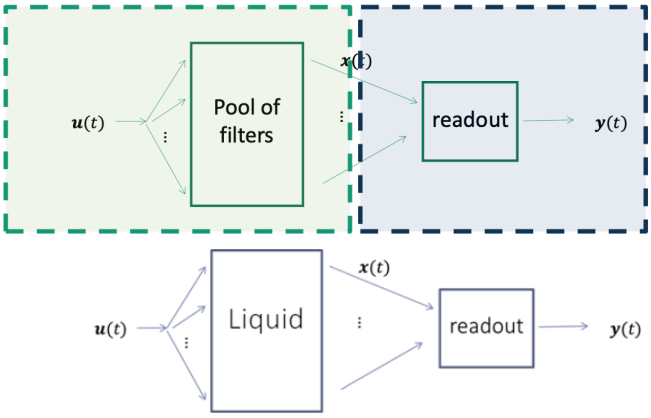
Machine Learning: use these computational models to solve problems. Concerning **temporal problems** (where input and output have temporal dimensions) we can say that learning is computationally intensive, and efficiency plays a key role

Dynamical Systems (Repetita): neurons implement input-driven non autonomous dynamical systems and are are excitable because their dynamical regime is **close to a bifurcation.**

Time and **randomness** play a crucial role in neurons; times dictates neural responses, while stochasticity in connections allows for adaptive and flexible neural processing, influencing learning and information flow in networks.

Notation:
- input $\boldsymbol{u}(t)$
- State $\boldsymbol{x}(t)$
- Output $\boldsymbol{y}(t)$

Objective: perform a temporal task in real-time i.e. given a time-series $u(t)$ in input predict/estimate/ forecast a target time-series $d(t)$ in output



Idea: encode the input history into a **pool of dynamical systems/filters** use such pool as input for the **output** computation

How to compute filters? Let's use the **liquid metaphor**

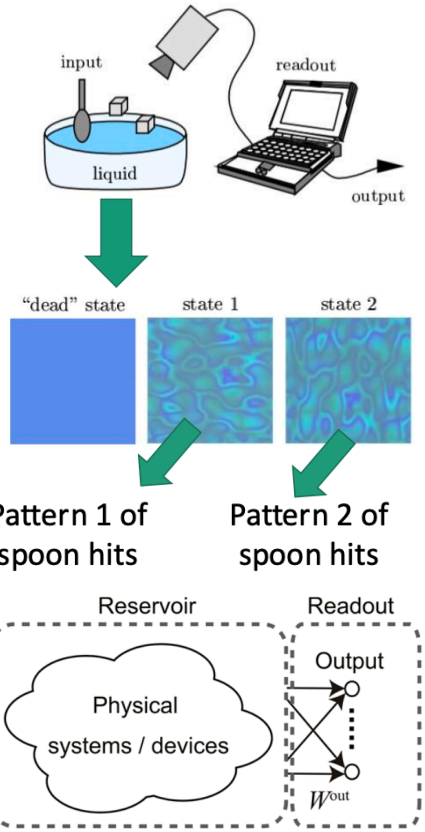
Real-time Computing with a Liquid Medium

Idea: Imagine throwing a stone into a pool of water. The waves and how they propagate can tell something on the stone stimulus to the water. Throwing other stones we can measure the interaction among the waves and these can tell us something on the history of thrown stones. **The state of the water can be useful to distinguish among different** (recent) histories of stones throwing stimuli

Input time series: Sequence of perturbations applied to the liquid, e.g. encoded by the pattern of spoon/sugar cubes hits

Liquid states: the surface of the liquid encodes the history of the spoon perturbations, like a state machine, but with a liquid state. Stable states are not of interest

Readout: it's a **memoryless** function that transform the liquid state of the machine in the desired output value (can be anything, a NN a convNN a simple regressor…)



Strong assumption needed for output computation: at each time, the liquid contains all the relevant information on the input history

Richness: the liquid should provide a rich reservoir of possibly diverse representations of the input history

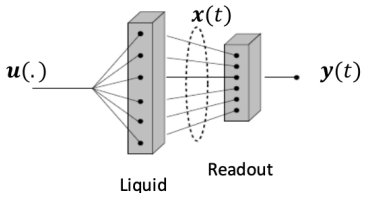
Randomness: random temporal filters are suitable to the purpose as long as they provide rich/diverse enough temporal dynamics

Reservoir: we can use a bucket of liquid or any other physical device with some dynamic observable physical properties (eq a photonic circuit or skin of an octopus)

Liquid State Machines (LSMs)

Leverage the inherent randomness and dynamic properties of spiking neural networks to process temporal signals. **Core idea** is to project an input signal into a *high-dimensional latent space* using a "liquid" that is a **randomly connected network** of spiking neurons and acts as a **dynamic temporal memory**

Input layer: feeds the external signal into the liquid (*E.g., sensory data, time-series information).*



Liquid $x(t) = F^L(u(\,.\,))$**:** a randomly connected network of spiking neurons that transforms the input signal into a higher-dimensional, temporally rich state, capturing both current and past inputs. It's a continuous state machine.

Readout $y(t) = F^R(x(t))$**:** it extracts useful information from the liquid state. It can be trained to interpret the liquid for specific tasks (prediction, classification, generation…)

Computation of the state x is a dynamical system while computation of readout is instantaneous. It's possible to use **two different readout on the same liquid if we have multiple task on the same input time series**

[!!] Temporal filters through the liquid have **two major properties:**

1. **Time-invariant**: a temporal shift of the input determines a temporal shift of the output of the filters (the state of the system) of the same amount
2. **Fading memory**: the output of the filters for an input sequence u1 can be approximated by the output of the filters for another input sequence u2, if u2 approximates well u1 over a long (recent) time interval. **For long input signals the output of the filters depends only on the most recent inputs (the suffix)**.

Pointwise separation property (Liquid)

Suppose there are 2 sequences s_u ands_v, which differ before a time step t_1 $(t < t_1 : s_u(t) \neq s_v(t))$ there exist a basis filter in the class of considered basis filters such that:

$$F^L(s_u(\ldots, t_1)) \neq F^L(s_v(\ldots, t_1)$$

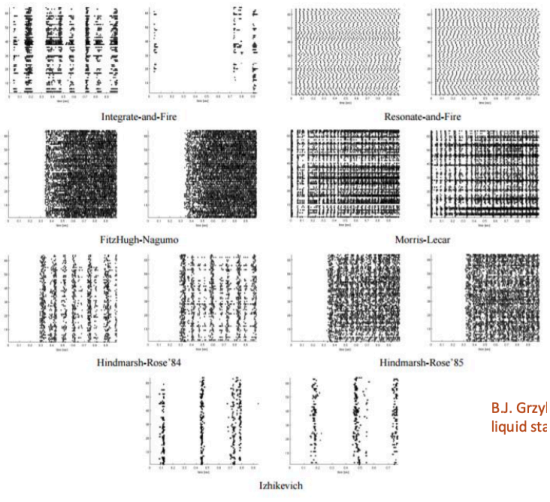
Universal approximation property (Readout): any continuous function on a compact domain can be uniformly approximated by our readout (eg a MLP..)

[Theorem] a Liquid State Machine can implement any time-invariant temporal filter with fading memory, provided that:

- the liquid satisfies the pointwise separation property
- the readout satisfies the universal approximation property

[!!] The liquid does not need to be trained/adapted, training can be restricted only to the readout

What to use for the readout: any classification or regression tool. Provided that the liquid gives a rich transformation of the temporal input stream a linear readout can be used. **Provided that the liquid gives a rich transformation of the temporal input stream a linear readout can be used**



Mathematical models of neural microcircuits are suitable to implement the liquid

We can use as **liquid** a layer of interconnected neurons

Implementation of Liquid State Machines

Liquid

- A layer of randomly interconnected spiking neurons (a microcircuit model): the **connectivity randomness** follows biologically plausible patterns
- Typically untrained or adapted through the STDP plasticity rule (or adapted with some unsupervised approach)

Readout

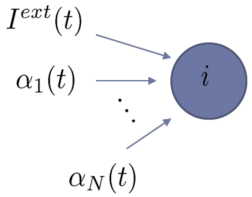
- Any classification/regression model (perceptron, spiking neuron, MLP, SVM, etc.)
- Training with: pseudo-inversion, Tikhonov regularization, delta rule, backpropagation, linear regression, etc….

Neural coding: the liquid state can be

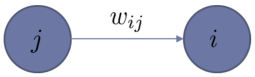
- Roughly, the spiking/non-spiking activity pattern of each neuron in the liquid
- Temporal coding: firing-rate

(Part2 Lecture1) Firing Rate Models and Hebbian Rules

This second part is focused on learning. In this lesson we will cover: synaptic plasticity, the Hebb rule and its variants, Firing-rate models, covariance rule, BCM rule, Synaptic normalization and Oja rule.



Recall: NN and synaptic plasticity
neurons are organized in a pool of connected units, **signal transmission** between two neurons is modulated by a weight value that represent the strength of the connection.



α_i are the signal comings from other neurons coming into the current neuron

Neurons are organized in **layers**. There're two functionally different types of connections:
- **feedforward connections** from lower layers to higher layers
- **recurrent** (**feedback**) **connections** from the same layer (or higher layers)

Recall: plasticity

Synaptic plasticity is the basic learning phenomenon:
- **Long Term Potentiation** (LTP): increase in the synaptic efficacy
- **Long Term Depression** (LTD): decrease in the synaptic efficacy

Hebbian learning: changes in synaptic transmission efficacy are due to correlations between pre- and post-synaptic activities (spiking)

Supervised learning: a teacher (supervisor) imposes a set of desired input-output (target value) responses. The **adaptation rule** adjusts synapses until the desired behavior emerges

Reinforcement Learning: supervision is given only in terms of **evaluative feedback**. Learn by doing, thanks of a function that measures the goodness of the behaviour of the model.

Unsupervised Learning: the network responds to the input based only on its intrinsic connections and dynamics. The network organizes based only on the synaptic plasticity rule

When we deal with learning as a dynamical process, we need to take into considerations several aspects like stability, competition…

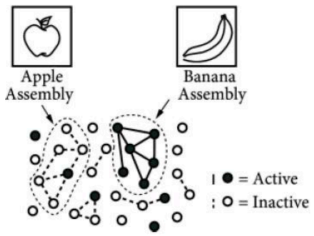
Stability
Increasing the synaptic strength in response to activity leads to a positive feedback process. Might determine an uncontrolled growth of the synaptic strengths: **weight values update must be somehow "bounded"**

Competition
If synaptic connection are modified independently: might determine lack of selectivity loosing efficacy in the learning process.
Synaptic competition: some synapses are forced to weaken when other become stronger

In our system we'd like **stability** and **competition** among synapses

Recall on Hebbian Learning
Hebbian learning is by nature **local**: we consider individual connection/synapses. Connections between neurons in a pool of units that fire together (e.g., in response to a similar stimulus) are strengthened

Hebbian creates an **associative memory**: the repeated, simultaneous activation of neurons leads to a *persistent strengthening* of the connections between them, forming a *neuronal assembly*. When a part of the assembly is activated by a stimulus, it can trigger the activity of the entire assembly, facilitating the recall of information.



Example: we have to type of images, banana and apple. When the neuron receives an image only a subset of neuron is activated

Idea: when the network receives a specific type of input a subset of the network activates.

FIRING-RATE MODELS

We're shifting from spiking neurons to firing-rate neurons. The latter consists of neuron models in which information is not encoded as discrete spikes but rather as a numerical value representing the **likelihood of a neuron firing**.

Def: firing-rate models are **networks of neuron-like** units whose output consists of firing rates (number of spikes averaged over time) rather than action potentials

Pros: **easier** to simulate, analyze, setup
Cons: temporal aspects cannot be accounted and **not so biological plausible**

How to encode the communication among neurons?

The **activity of a spiking neuron** can be described in terms of its **neural response function:**

$$\rho(t) = \sum_i \delta(t - t_i)$$

delta function spikes located at times when the neuron fired action potentials. It locates spiking trains of the neuron in time (delta permette di localizzare nel tempo gli spike del neurone).

The **neural response function** can be **approximated** by the firing rate (probability density of firing) → $r(t) = \frac{1}{\Delta t} \int_t^{t+\Delta t} < \rho(\tau) > d\tau$

Firing rate is the integral of the expected value of the function ρ over an interval $[t, t + \Delta t]$ averaged over the time interval size.

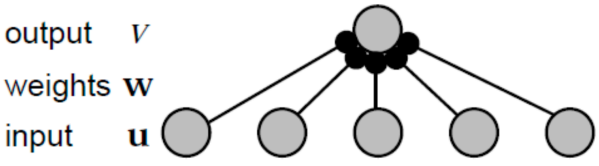
Hint: r(*t*) is the probability density of firing and is obtained from ρ(*t*) by averaging over trials

Firing-Rate model construction

Firing-Rate model two aspects of neural computation:
1. **Total synaptic input** is computed starting from firing rates of presynaptic neurons
2. **Postsynaptic firing rate** is then computed starting from total synaptic input

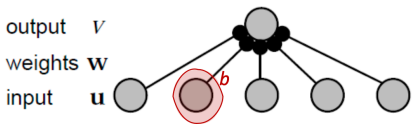
First, we calculate the **total synaptic input**, defined as the overall current reaching the soma due to synaptic conductance changes triggered by presynaptic action potentials. Then, in the second step, we determine the **postsynaptic firing rate** based on this total input received by the neuron.

Notation
- total synaptic input: I_s
- presynaptic firing rate: $u(t)$
- postsynaptic firing rate: $v(t)$
- N_u: number of input neurons
- $u = [u_1, \ldots u_{N_u}]$: vector of input rates



Considering a single output neuron and Nu input neurons. Each synapses has a weight vector

1. Computing total synaptic input to a postsynaptic neuron: an incoming spike from b-th input at time 0 that determines a postsynaptic current in our neuron $w_b K_s(t)$. This is called **synaptic kernel**



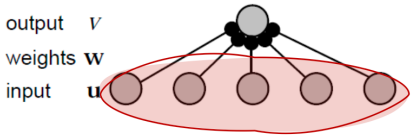
1st assumption: the effects of spikes at a single synapse sum linearly

So the contribution of b-th synaptic connection to the total synaptic input is:

$$w_b \sum_{t_i<t} K_s(t-t_i) = w_b \int_{-\infty}^{t} d\tau\, K_s(t-\tau)\rho_b(\tau)$$

exploiting basic properties of the Dirac delta function and the definition of the neural response function ρ we can write the

The synaptic weight w_b is multiplied by the sum over time of synaptic kernel, where t_i are firing times of the pre-synaptic neuron preceding b. ***K*** describes the temporal evolution of the synaptic current



2nd assumption: the effects of different synaptic currents (from different inputs) sum linearly

Total synaptic current from all the presynaptic inputs is obtained summing up over all neurons in the network::

$$I_s = \sum_{b=1}^{N_u} w_b \int_{-\infty}^{t} d\tau\, K_s(t-\tau)\rho_b(\tau)$$

$$I_s = \sum_{b=1}^{N_u} w_b \int_{-\infty}^{t} d\tau\, K_s(t-\tau)u_b(\tau)$$

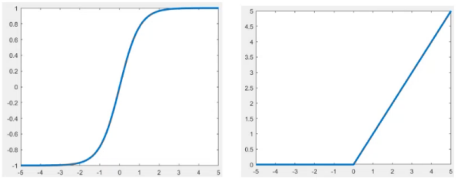
simplification:
the neural response function $\rho_b(t)$ is approximated by the firing rate $u_b(t)$

Using the exponential form for the synaptic kernel $K_s(t) = \frac{1}{\tau_s} e^{\frac{-t}{\tau_s}}$

$$\tau_s \frac{dI_s}{dt} = -I_s + \mathbf{w}\cdot\mathbf{u} \quad \Leftarrow \quad \tau_s \frac{dI_s}{dt} = -I_s + \sum_{b=1}^{N_u} w_b u_b = -I_s + \mathbf{w}\cdot\mathbf{u}$$

So the **total Synaptic Input to a neuron** results into the inner product between the input vector u and the weight vector w. The total synaptic input I_s approaches the value $u \cdot w$ with τ_s speed.

2. Computing Firing Rate of postsynaptic neuron
For constant input I_s the steady state value of the neuron's firing rate is expressed by an **activation function *F***: $v = F(I_s)$



F-I plot: frequency of firing in response to a given constant input current

Time-independent output:

$$\begin{cases} \tau_s \dfrac{dI_s}{dt} = -I_s + \boldsymbol{w}\cdot\boldsymbol{u} \\ v = F(I_s) \end{cases}$$

the output firing rate shows no dynamics w.r.t. the input (instantaneous firing rate)

Time-dependent output:

$$\begin{cases} \tau_s \frac{dI_s}{dt} = -I_s + \boldsymbol{w} \cdot \boldsymbol{u} \\ \tau_r \frac{dv}{dt} = -v + F(I_s(t)) \end{cases}$$

If also the postsynaptic firing rate depends on time. τ_r determines how rapidly the output firing rate approaches its steady state $F(I_s(t))$

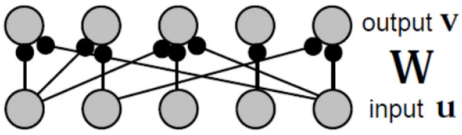
Simplifications:

- Input dynamics much faster than output dynamics $\tau_s << \tau_r \rightarrow$ we can discard the first dynamical system equation, and we've only dynamics in the output firing rate
- if output firing rate dynamics is much faster than the total synaptic input dynamics $\tau_r << \tau_s \rightarrow$ we can simplify using **time-independent output** formulation

Let's put this ideas together into a network of neuron

Feedforward Networks

A network of interconnected neuron with N_v output units (1+ output neuron), whose firing rates are collected into $\boldsymbol{v}$. Synaptic weights are collected into a matrix $\boldsymbol{W}$ (one row for one output neuron and one column for one input neuron).



Previous formulas stands for each output neuron in our network:

$$\tau_r \frac{dv_a}{dt} = -v_a + F(\sum_{b=1,\ldots,N_U} W_{ab} u_b)$$

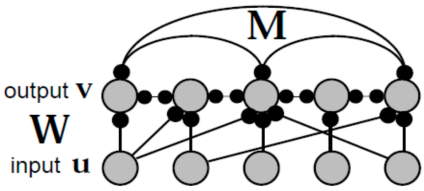
if considering the *ai-th* output neuron

$$\tau_r \frac{d\boldsymbol{v}}{dt} = -\boldsymbol{v} + F(\boldsymbol{W}\,\boldsymbol{u})$$

Nota: this is a continuous version of a NN. NN in ML are a discretized version of this

Recurrent Networks

Neurons in the output layer are interconnected by synapses whose weights are described by matrix $\boldsymbol{M}$.



$$\tau_r \frac{d\boldsymbol{v}}{dt} = -\boldsymbol{v} + F(\boldsymbol{W}\,\boldsymbol{u} + \boldsymbol{M}\,\boldsymbol{v})$$

Variant: pools of excitatory and inhibitory units $\rightarrow$

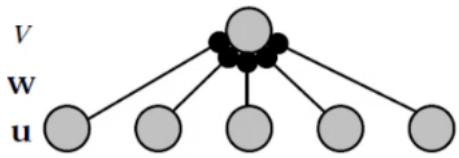
$$\tau_E \frac{d\mathbf{v}_E}{dt} = -\mathbf{v}_E + \mathbf{F}_E\,(\mathbf{h}_E + \mathbf{M}_{EE} \cdot \mathbf{v}_E + \mathbf{M}_{EI} \cdot \mathbf{v}_I)$$

$$\tau_I \frac{d\mathbf{v}_I}{dt} = -\mathbf{v}_I + \mathbf{F}_I\,(\mathbf{h}_I + \mathbf{M}_{IE} \cdot \mathbf{v}_E + \mathbf{M}_{II} \cdot \mathbf{v}_I)$$

Idea of firing-rate models: starting from the spiking neurons we can derivate a formulation that can describe a way in which over time we can calculate the firing rate of a pull of neurons.

MATHEMATICAL FORMULATION OF HEBBIAN RULES

Which model consider? We consider the simplest model with one single postsynaptic neurons (output) and N_u presynaptic inputs (inputs)



$$\tau_r \frac{dv}{dt} = -v + F(\boldsymbol{w} \cdot \boldsymbol{u})$$

Nota: the · means the dot product between vectors, ie $w \cdot u = w^T u$

Assumption:

- linear activation function → $\tau_r \frac{dv}{dt} = -v + \boldsymbol{w} \cdot \boldsymbol{u}$
- Dynamics of plasticity are much slower than the firing rate dynamics, the **output firing rate dynamics** are **instantaneous** at our time-scale of interest: $v = F(\mathbf{w} \cdot \mathbf{u}) = \mathbf{w} \cdot \mathbf{u}$

  so **instantaneously the output firing rate** goes to the value of the asymptotic firing rate output firing rate for the linear model

  Dynamics of weight changes: $\tau_w \frac{dw}{dt} = G(v, \boldsymbol{u}, \boldsymbol{w})$

  nota: time now is no longer time**, time represents differente input patterns** (different data given in input to the model that drives the learning of the model).

Basic Hebb Rule

Recall that a strength in the connection among two neurons happens if there is a simultaneous activation of the two

Online version: sum the weight changes caused by each input pattern $\tau_w \frac{dw}{dt} = v\,\boldsymbol{u}$

Averaged Hebb rule: compute the weight change due to the average of the changes caused by the input patterns (**considering a batch of data**). Can be computed in parallel.

$$\tau_w \frac{dw}{dt} = < v\,\boldsymbol{u} >$$

average over the input patterns

Correlation based plasticity rule:

Equation of the neurons:

$$\begin{cases} \tau_w \frac{d\mathbf{w}}{dt} = \langle v\mathbf{u} \rangle \\ v = \mathbf{w} \cdot \mathbf{u} \end{cases}$$

Nota: <..> is the average over the training set

We can expand first term: $\tau_w \frac{d\mathbf{w}}{dt} = < v\mathbf{u} > = < \mathbf{wuu} > = < \mathbf{uu} > w = Q\mathbf{w}$

With Q being the input correlation matrix: $Q = < \mathbf{uu} >$, $Q_{bb'} = < u_b u'_b >$

correlation based plasticity rule: $\tau_w \frac{d\mathbf{w}}{dt} = \mathbf{Q} \cdot \mathbf{w}$ or $\tau_w \frac{dw_b}{dt} = \sum_{b'=1}^{N_u} Q_{bb'} w_{b'}$

Note: this does not depend explicitly on the output because the output is a function of input and weights only

Stability proof

Considering the network from a dynamical system prospective, we don't want our weights to become too huge.

Considering $\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u}$ we can state that the **basic Hebb rule is <u>unstable</u>**:

Proof: consider the dynamics of the length of the weight vector

$$|\mathbf{w}|^2 = \mathbf{w} \cdot \mathbf{w} = \sum_b w_b^2$$

$$d|\mathbf{w}|^2/dt = 2\mathbf{w} \cdot d\mathbf{w}/dt$$

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

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

NOTE FOR THE EXAM: professor derived this one; he leaves the others as exercises but will ask about them at the oral

and $2v^2$ is an unbounded value so weights norm has an **unbounded growth**

This formulations tells that as input income in the system, weight values can only increase or remain constant, never decrease.

Problems:
- No competition among the synapses in this formulation
- **Saturation constraints** can be imposed to avoid the growth (but it will stop learning so can be not useful at all)

Solution: modify the rule adding a **Long Term Depression** term

Modifying Hebbian rule: Covariance Rule

Introduce also an LTD factor: in presence of presynaptic activity, now the synaptic strength is:
- **potentiated** → for high postsynaptic activity
- **depressed** → for low postsynaptic activity

Covariance rule formulations: $\tau_w \frac{d\mathbf{w}}{dt} = (v - \theta_v)\mathbf{u}$ $\tau_w \frac{d\mathbf{w}}{dt} = v(\mathbf{u} - \theta_u)$

Where θ_v is the **postsynaptic threshold** for which LTD switches to LTP:
- if the output is larger than the threshold we have a **potentiation** in the synaptic weight
- Otherwise **depression**

Alternative: we can apply the same idea on the input (**applying a threshold on the pre-synaptic rate**). In presence of postsynaptic activity, the synaptic strength is:
- **potentiated** → for high presynaptic activity
- **depressed** →for low presynaptic activity

Nota the threshold θ_u now is a vector

Why these are covariance rules?

Because thresholds are typically set to averages over the training sets: $\theta_v = < v >$, $\theta_u = < \mathbf{u} >$

Now using the the averaged form of the plasticity rule:

$$\tau_w \frac{d\mathbf{w}}{dt} = < (\mathbf{u} - < \mathbf{u} >)\mathbf{u} > \cdot \mathbf{w} = C \cdot \mathbf{w}$$

Where $C = \langle (\mathbf{u} - \langle \mathbf{u} \rangle)(\mathbf{u} - \langle \mathbf{u} \rangle) \rangle = \langle \mathbf{uu} \rangle - \langle \mathbf{u} \rangle \langle \mathbf{u} \rangle = \langle (\mathbf{u} - \langle \mathbf{u} \rangle)\mathbf{u} \rangle$ is the **covariance matrix computed over the input patterns**

Despite the transformation also the **covariance rule** is **unstable** and **non-competitive:**

$$\tau_w \frac{d\mathbf{w}}{dt} = \mathbf{C} \cdot \mathbf{w} \qquad \tau_w \frac{d\mathbf{w}}{dt} = (v - \theta_v)\mathbf{u} \qquad \tau_w \frac{d\mathbf{w}}{dt} = v(\mathbf{u} - \boldsymbol{\theta}_u)$$

$$d|\mathbf{w}|^2/dt = 2v(v - \langle v \rangle)$$

$$< 2v(v - < v >) > \quad \propto \quad < v(v - < v >) > \quad = \quad < v^2 > - < v >^2$$

BCM Rule

Idea: Both presynaptic and postsynaptic activities are required to trigger a weight change (either LTP or LTD)

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

θ_v is a threshold over the output firing rate:

- if θ_v it's **constant**: unstable/unbounded growth
- if the **threshold grows faster than the output firing rate** it's stable: $\tau_{\theta_v} \frac{d\theta_v}{dt} = v^2 - \theta_v$

[!!] With **varying threshold** the BCM rule is **stable** and **implements competition**: strengthening one synapse leads to raising the threshold)

Synaptic Normalization: direct approach to stabilize Hebbian plasticity rules consists in normalizing the weight vector. This can be done in two ways:

1. the sum of weight values is fixed
2. the sum of squared weight values is fixed

1. Hebb rule with Subtractive Normalization:

$$\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u} - \frac{v(\mathbf{n} \cdot \mathbf{u})\mathbf{n}}{N_u}$$

with $n = [1,1,1...,1]$ so that $\sum w_b = n \cdot w$

Weight normalization: the sum of the weights cannot change (observable trough dynamics of $\mathbf{nw}$)

$$\tau_w \frac{d\boldsymbol{n}\,\boldsymbol{w}}{dt} = v\,\boldsymbol{n} \cdot \boldsymbol{u}\left(1 - \frac{\boldsymbol{n} \cdot \boldsymbol{n}}{N_u}\right) = 0$$

Highly competitive rule

2. Oja Rule

Impose a dynamic constraint on the sum of the squares of synaptic weights (ie on the length of the weigh vector)

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

This constraint is an **asymptotic constraint** so its not fixed, this holds after a while (asymptotically)

This rule is **stable:** $\tau_w \frac{d|\mathbf{w}|^2}{dt} = 2v^2(1 - \alpha|\mathbf{w}|^2)$ and $|w|^2$ will converge towards $1/\alpha$

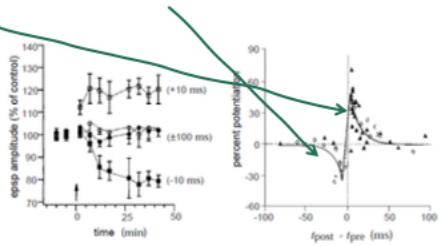
The rule **introduces competition between different weights** because if I change one weight I need to change all the others to preserve the rule

Timing-Based Rules

The effect of pre and postsynaptic timing can be included by including a temporal difference between times in which the firing rates are considered

$$\tau_w \frac{d\mathbf{w}}{dt} = \int_0^\infty d\tau \; (H(\tau)v(t)\mathbf{u}(t-\tau) + H(-\tau)v(t-\tau)\mathbf{u}(t))$$

$H(\tau)$ determines the rate of synaptic modification that occurs due to postsynaptic activity separated from presynaptic activity by a τ interval



Left side of the equation is as previous formulation while right side is an approximation of … (non si sentiva a lezione)

Hebbian rule revisited

Focus on the case of a single post-synaptic neuron: $\tau_w \frac{d\mathbf{w}}{dt} = Q\mathbf{w}$

Now **let's study the eigenvectors of** Q (**nota** correlation matrix always symmetric):

Eigenvectors are defined as: $Q \cdot e_\mu = \lambda_\mu e_\mu$ for $\mu = 1,2,...,N_u$

Properties of symmetric matrices:

1. the set of its eigenvectors e_μ forms an **orthonormal basis** of the input space $\mathbb{R}^{N_u} \rightarrow$ $e_\mu e_\nu = \delta_{\mu\nu}$ (all orthogonal to each other)
2. the eigenvalues are real and non-negative: $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_{N_u}$

The eigenvector e_1 is called the **principal eigenvector** as it corresponds to the higher eigenvalue

Given this properties we can state about w (nota w changes over time):

$$w(t) = \sum_{\mu=1}^{N_u} c_\mu(t)\mathbf{e}_\mu$$

I can rewrite w in terms of e_μ because the latter forms a basis of my vector space

where coefficients are equal to the dot products of **w** with the cor responding eigenvectors. For example, at time 0, $c_\mu(0) = w(0) \cdot \mathbf{e}_\mu$

Solving the differential equation $\tau_w \frac{d\mathbf{w}}{dt} = Q\mathbf{w}$, and from previous relation of $w(t)$ we obtain:

$c_\mu(t) = c_\mu(0)\exp(\lambda_\mu t/\tau_w)$ so that:

$$w(t) = \sum_{\mu=1}^{N_u} \exp(\frac{\lambda_\mu t}{\tau_w})(w(0) \cdot \mathbf{e}_\mu)\mathbf{e}_\mu$$

this shows how the weight vector evolves over time in the space that is identified by the eigenvector of the correlation matrix

The exponential factors grow over time (as eigenvalues are all nonnegative and sorted), but they are dominated by the term corresponding to the **largest** eigenvalue λ_1 (as it use the largest so it influences more the summation). The weight vector will be approximated with its projection on the first eigenvector of the correlation matrix (so it become parallel to the principal eigenvector because it becomes something like a number times the principal eigenvector.

So for large values of t (i.e. for large datasets) w is approximated by $\exp(\frac{\lambda_1 t}{\tau_w})(w(0) \cdot \mathbf{e}_1)\mathbf{e}_1$ and $w \propto e_1$ and the output of the neuron v will become proportional to the projection of the input u along the direction of e_1, i.e. $v \propto e_1 \cdot u$.

Idea: The response of the neuron is proportional to the projection of the input vector onto the principal eigenvector → essentially, **the Hebb rule corresponds to performing a PCA** compressing the whole dataset onto a single dimension

The **Oja rule** has a different shape but with the same steps we end up to a formulation in which the weight vector w becomes:

$w = e_1/(\alpha)^{1/2}$ as $t \to \infty$

It gives a **weight vector that is parallel to the principal eigenvector**, but normalized to a length of $\alpha^{1/2}$ rather than growing without bound (confirming stability of Oja rule)

In the case of **subtractive normalization**:

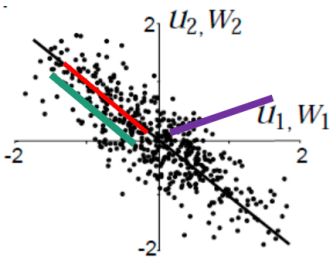
$$\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u} - \frac{v(\mathbf{n} \cdot \mathbf{u})\mathbf{n}}{N_u} \qquad \tau_w \frac{d\mathbf{w}}{dt} = \mathbf{Q} \cdot \mathbf{w} - \frac{(\mathbf{w} \cdot \mathbf{Q} \cdot \mathbf{n})\mathbf{n}}{N_u}$$

$$\mathbf{w}(t) = (\mathbf{w}(0) \cdot \mathbf{e}_1)\,\mathbf{e}_1 + \sum_{\mu=2}^{N_u} \exp\left(\frac{\lambda_\mu t}{\tau_w}\right)(\mathbf{w}(0) \cdot \mathbf{e}_\mu)\,\mathbf{e}_\mu$$

The weight vector approaches a configuration that is parallel to the eigenvector corresponding to the largest (or next largest) eigenvalue (of the correlation matrix)

Wrapping-up

Hebb and **Oja rules** generate weights that are parallel to the principal eigenvector of the correlation matrix.
Example:



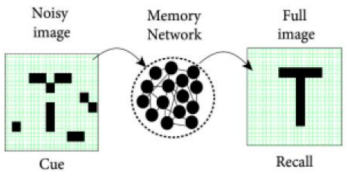
- Considering a two dimensional model with output $v = wu$
- The input sampled from two-dim Gaussian distr. with different std.
- The initial weight vector $\boldsymbol{w}$ 0 is random
- the final weight vector is proportional to the principal eigenvector of the input correlation matrix

Principal component projection - Advantages

- Setting the direction of w to be proportional to the principal eigenvector e_1
- When I project the input vector using my neuron I project it in the direction of maximum variability (over the training set) which is the optimal choice to encode the (represent and reconstruct) input through linear relations
- From the information theory perspective it's like maximizing the variance of the output v using a Hebbian rule maximizing the amount of information that v carries about u.

(Part 2 Lecture 2) Hopfield Networks

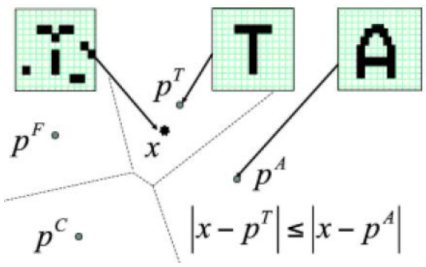
INTRODUCTION TO BRAIN AND MEMORY



Brains can realize **associative memories** that can be exploited to complete partial information: a noisy input triggers the recall of the most similar **prototype** (recall process)

noisy/partial image is recognized as a T

Recall in Information Theory



In information theory/information retrieval, **prototypes** are memorized as patterns p^μ for $\mu = 1...M$. Then, given an an input pattern x the **recalled prototype** is the one minimizing an error function (for example distance function):

$$|x - p^\alpha| \leq |x - p^\mu| \text{ for all } \mu$$

For our interests, we can **replace the explicit search algorithm by the dynamics of interacting neurons**. In this setting we can consider the cross-talk of neurons will find the prototype that best corresponds to a noisy input

In our brain, sub-networks of connected neurons that respond together in correspondence of a concept are called **neural assembles** and they are the biological background of neuronal associative memories.

HOPFIELD MODEL

The role of time
In the context of a (firing-rate based) neural network, time is incorporated into the operation of the model by means of feedback:
- **Local feedbacks** concerns a single neuron in the networks
- **Global feedback** are feedbacks that comes from other neurons or layers in the network

Analyzing Dynamical neural network models we can state that **stability plays a fundamental role in associative memories**.

Dynamical system recall

A dynamical system is a system whose state varies with time. The state is vector in the N-dimensional space $x(t) = [x_1(t), x_2(t), \ldots, x_N(t)]^T$ and the dynamics is expressed as differential equations in continuous time settings or as iterated map (thanks to Euler approximation or similar) in discrete time setting.

$$\frac{d}{dt}x_j(t) = F_j(x_j(t)), \qquad j = 1, 2, ..., N$$

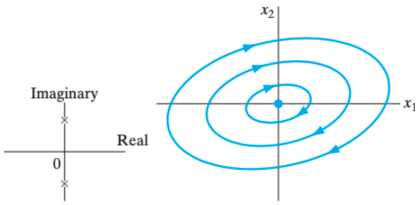
$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{F}(\mathbf{x}(t))$$

A vector $\bar{x}$ is an **equilibrium state** (or a **stationary state**, or a **fixed point**) of the dynamical system if $F(\bar{x}) = 0$ (the zero vector); so if the dynamics of the system stop at $\bar{x}$

Local stability of equilibria can be studied by **linearizing** the governing equations around the fixed points: $F(X) \approx \bar{x} + A\Delta x(t)$ where A is the jacobian matrix, $\bar{x}$ is our fixed point:
- $\bar{x}$ is a stable fixed point if attracting near points
- $\bar{x}$ is unstable if it repels near points

The equilibrium can be classified looking at the complex ad the real parts of the eigenvalues of the Jacobian (**see here**)

Let's just add the **center stability:** the eigenvalues has real part equal to 0 and complex part different from 0. So it's both **stable** and **non dissipative** in the sense of the dynamics: nearby trajectories oscillate around the equilibrium without converging to it, indicating neutral stability



Uniform stability
The equilibrium state $\bar{x}$ is said to be **uniformly stable** if, for any positive constant ϵ, there exists another positive constant $\delta = \delta(\epsilon)$ (so delta is a function of epsilon) , such that if:

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

Idea: if we want that our system stays confined in an epsilon tube around the fixed point in the future, we just need to initialize it in a sufficiently small neighborhood of the fixed point

Convergence
The equilibrium state $\bar{x}$ is said to be **convergent** if there exist a positive constant $\delta > 0$, such that if $\|x(0) - \bar{x}\| < \delta$ then $x(t) \to \bar{x}$ as $t \to \infty$

Idea: if the initial state is close enough to the equilibrium state, the trajectory will approach the equilibrium state with $t \to \infty$

Asymptotic stability

The equilibrium state $\bar{x}$ is said to be **asymptotically stable** if it is both:

i) uniformly stable
ii) convergent

Global Asymptotic stability

The equilibrium state $\bar{x}$ is said to be **global asymptotically stable** if it is both:

i) uniformly stable
ii) all the trajectories of the system converge to $\bar{x}$ as $t \to \infty$

The **difference** is that in global asymptotic case not all the trajectory that starts near $\bar{x}$ (convergence condition) but ALL possible trajectories for each possible initial conditions will moves toward the fixed point

Lyapunov's Theorems

Stability can be investigated using the direct method of Lyapunov. This method allows the stability of an equilibrium point to be assessed without explicitly solving the system's differential equations (so we don't need to linearize or to compute the Jacobian matrix)

Lyapunov function: a Lyapunov function is a **scalar**, **positive-definite** function, defined in a small neighborhood of the equilibrium such that:

1. The function $V(x)$ has continuous partial derivatives wrt the elements of the state x
2. $V(\bar{x}) = 0$
3. $V(\bar{x}) > 0$ if $x \in U - \bar{x}$ where U is a small neighbourhood around x

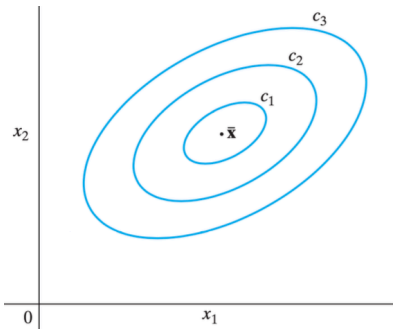
Theorem 1: The equilibrium state $\bar{x}$ **is stable** if, in a small neighborhood of $\bar{x}$, there exist a positive-definite function $V(x)$ such that its derivative wrt time is **negative semidefinite** in that region, ie if $\frac{d}{dt}V(x) \leq 0$ for $x \in U - \bar{x}$ then $\bar{x}$ is stable

Idea: instead of linearizing the system and studying the eigenvalues we use the function $V(x)$ that depends on the dynamic of the systems and we can study the derivative of this function instead of directly study the dynamic of the system that is more complex

Theorem 2: The equilibrium state $\bar{x}$ is **asymptotically stable** if, in a small neighborhood of $\bar{x}$, there exist a positive-definite function $V(\mathbf{x})$ such that its derivative wrt time is **negative definite** in that region, ie if $\frac{d}{dt}V(x) < 0$ for $x \in U - \bar{x}$ then $\bar{x}$ is **asymptotically** stable

Lets plot Lyapunov surfaces to visualize those properties. Lyapunov surfaces are obtained by cutting the function $V(x)$ with planes (ie $V(x) = c$)

Under theorem 1: when a trajectory crosses a Lyapunov surface for some positive constant c, the trajectory moves inside a set of points defined by $V(x) \leq c$ and **never comes out of that Lyapunov surface**



Under theorem 2: the **trajectory will move from one Lyapunov surface to an inner Lyapunov surface** with smaller constant c as the constant c decreases in value, the Lyapunov surface moves closer to the equilibrium state (dynamics collapse trough the fixed point)

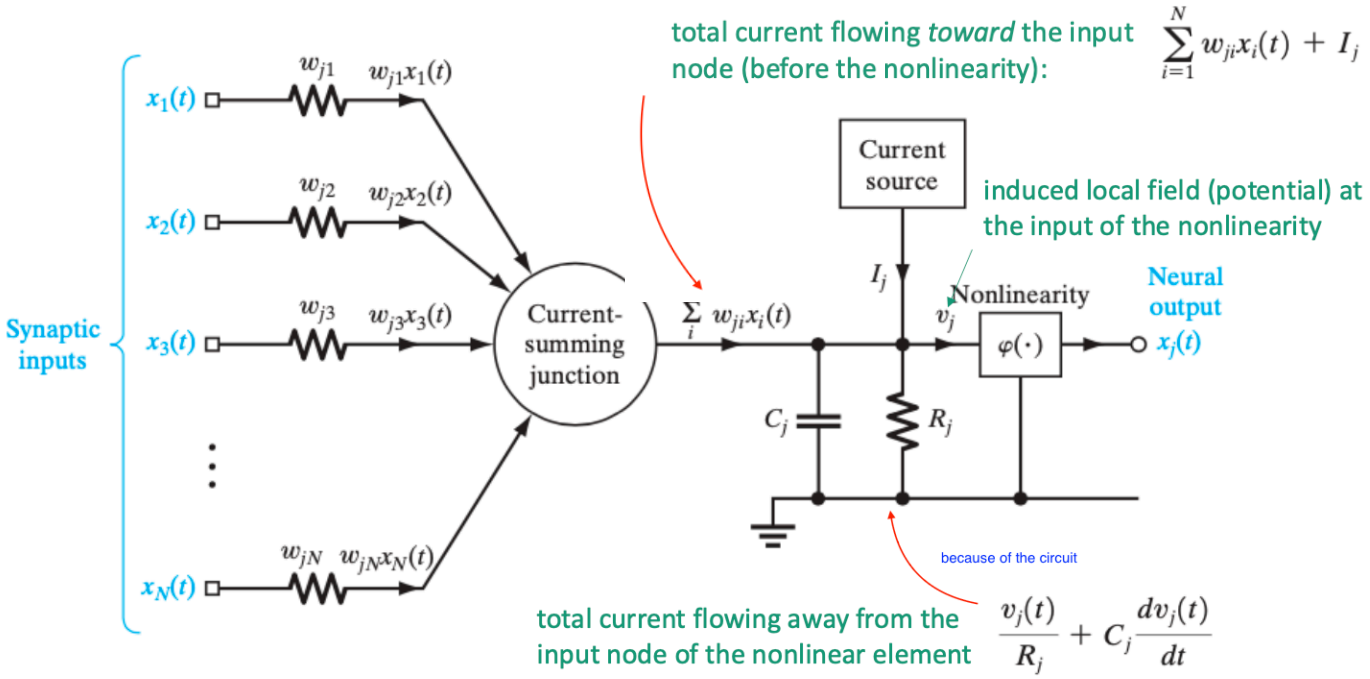
Lyapunov function $V(x)$ can be seen as a measure of "**system energy**" that is lowest at the equilibrium point.

Time derivative $V'(x)$ represents the rate of change of $V(x)$ along the system's trajectories. Its negative definiteness signifies that the system's "energy" decreases over time, guiding the system towards stability.

The existence of a suitable Lyapunov function $V(x)$, with a decreasing trend over time ($V'(x) < 0$), is **a powerful indicator of a system's stability.** The system inherently tends towards equilibrium without the need for explicit solutions to its dynamics

Neurodynamical models: the term **Neurodynamics** refers to the neural networks viewed as dynamical systems. The focus is on stability property. General characteristics are: a large number of degrees of freedom, non-linearity, dissipation (the state-space volume converges onto a manifold of lower dimensionality as time goes on)

Additive model



The additive model is an **integration model:** it's current flowing outside is computed as the weighted sum of its presynaptic input outputs. Then a bias is summed up as an external input source. Finally applying the Kirchhoff's law we obtain:

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

At the final output of the neuron, a **non linearity** is applied (tanh in continuous setting):

$$x_j(t) = \phi(v_j(t))$$

Recurrent Network

Consider a network consisting of the interconnection of N neurons, the dynamics of each neuron is described by an additive model. Then the dynamics of the network can be defined by a system of coupled first-order differential equations:

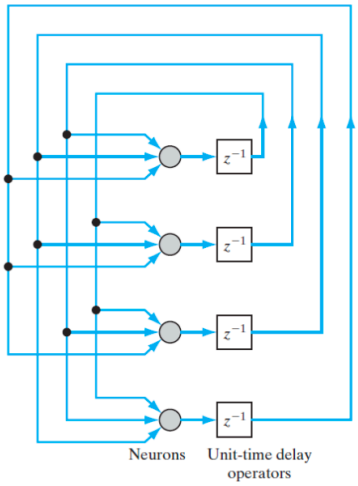
$$C_j \frac{dv_j(t)}{dt} = -\frac{v_j(t)}{R_j} + \sum_{i=1}^{N} w_{ji} x_i(t) + I_j, \qquad j = 1, 2, \ldots, N \qquad\qquad x_j(t) = \varphi(v_j(t))$$

Nota: we assume that the dynamics of the non linear computation of the output of a neuron is **instantaneous**.

Now we can **memorize some patterns in the activation of the neurons**. How? We need to train the weight vector in order to change the location of the attractor of the systems. So given this intuition, **learning**, consists in a non-linear dynamical manipulation of the location of the attractors that allows us to encode information or learn useful temporal structures **trough an energy minimization process**.

Hopfield Model definition

An Hopfield network is defined as a **set of neurons + time delays modules** (the z^{-1}) that delays the output from **one to four time steps ahed**. The output of each neuron is **fed back** to each of the other neurons in the network but there's **no self-feedback** (in the discrete formulation)



Continuous-time equations based on the additive neuronal model is:

$$C_j \frac{d}{dt} v_j(t) = -\frac{v_{j(t)}}{R_j} + \sum_{i=1}^{N} w_{ji} \phi\big(v_i(t)\big) + I_j,$$
$$j = 1, \ldots, N$$

Properties:

- **Asymptotic value** corresponds to the 0 value of the derivative so when v_j is equal to the sum of weighted input + the bias term (ie right-end side = 0)
- The **synaptic-weight matrix** is **symmetrical** $w_{ij} = w_{ji}$ for all i, j
- Neuron outputs is computed using a **non linear activation function** $x_j(t) = \phi(v_j(t))$

Energy function of Hopfield network:

The energy function of the Hopfield network is defined as follows:

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

component due the weight matrix — componet due to non linearity — bias term

The Energy function E is a **Lyapunov function** of the continuous Hopfield model. Moreover it is monotonically decreasing with time and its derivative (w.r.t. time) is null in the equilibria

The **time evolution of the continuous Hopfield model** represents a trajectory in the state space that seeks out the minima of the energy (Lyapunov) function E and stops at the fixed points of the Hopfield model

Idea: instead of studying the trajectory of the complex high dimensional system we can study the dynamics of the energy function

The Hopfield network may be operated in:
- **continuous model:** based on the additive model analyzed so far, neurons are permitted to have self-loops
- **discrete model:** based on a **McCulloch-Pitts model**, neurons **do not have self-loops** (i.e., $w_{jj} = 0$) and typically, neurons do not have a bias (i.e., $I_j = 0$)

Hopfield Model – Discrete Version

$$v_j = \sum_{i=1}^{N} w_{ji} x_i + I_j$$

Weight values

Local field

= affine transformation of the outputs of the other neurons

Bias (can be 0 for all neurons)

Output of neurons in the network

The discrete mode of operation of the Hopfield network is based on the **McCulloch-Pitts** model

The non linearity of the output is computed as the sign of the output current:

$$x_j = \phi(v_j) = \mathrm{sgn}(v_j) = \begin{cases} +1 & if\ v_j > 0 \\ -1 & if\ v_j < 0 \end{cases}$$

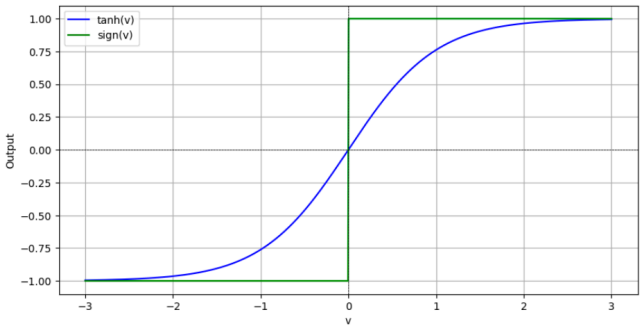
The network is composed by N neurons arranged in one single layer. According to the non linear activation function, each neuron is in one of two possible states:
- $+1$ the neuron is ON
- -1 the neuron is OFF (silent)

Non linear function:
A common choice for the nonlinear function $\phi(v)$ in continuous setting is instead is given by the **tanh function:**

$$\phi_i(v) = \frac{1 - \exp(-a_i v)}{1 + \exp(-a_i v)}$$

Where **the variable a is the steep variable**: larger the a, steepest the function near the origin

The **discrete counterpart** (sign) can be seen as a special case of the nonlinearity of the continuous model, for infinite gain neurons and the output is +1 for positive infinite v value and -1 for negative infinite v value.



Energy function of discrete counterpart:

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ (i\neq j)}}^{N} w_{ji}x_i x_j + \sum_{j=1}^{N}\frac{1}{R_j}\int_0^{x_j}\phi_j^{-1}(x)dx$$

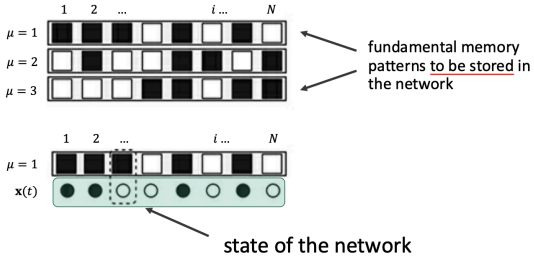
$$= -\frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ (i\neq j)}}^{N} w_{ji}x_i x_j + \sum_{j=1}^{N}\frac{1}{a_j R_j}\int_0^{x_j}\phi^{-1}(x)dx$$

In the discrete case, we are assuming that the gain is infinitely large

$$= -\frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ (i\neq j)}}^{N} w_{ji}x_i x_j$$

The network is used as a **content-addressable memory.** The task condissi in storing and recalling M patterns. Each **pattern** is defined as a desired configuration of the state of the neurons in the network (eg in the discrete case +1, -1, -1… while in continuous setting we could have real numbers). A **pattern is correctly recalled** if the state of the neurons corresponds to the configuration of the pattern

Network state update: The state of the network is updated in successive time-steps in an **asynchronous** manner (i.e., one random neuron per time-step):



Iterate until a fixed point is reached:

(1) at time t we choose randomly one neuron j to be updated
(2) Apply the state update equation for that neuron:

$$x_j(t+1) = \text{sgn}\left(\sum_{i=1}^{N} w_{ij}x_i(t)\right)$$

(Def) Overlap functions: given our patterns in our dataset to store $\{\epsilon_\mu \,|\, \mu = 1,2,...,M\}$ the overlap functions measures the similarity between patterns ϵ and states $x_i(t)$:

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

- max value = 1 in case of perfect recall
- min value = -1 (each neuron takes the opposite value of that one in the memory)

Nota: one overlap function for each pattern to be stored. **Normalized dot product** to measure the similarity

How to learn and recall in Hopefield networks:

The parameters w_{ij} are adjusted (learning rule) to locate the equilibria of the system, i.e., the minima of the Energy function, in the patterns to store:

- **Storage phase:** suitably modify the weight values to change the attractors
- **Retrieval phase**: follow the dynamics to relax the system in one of the attractors

Storage phase - Learning

The weights are set based on **correlations** between the fundamental memories (Hebbian learning). The main idea is to **consolidate the connection between neurons that has most of the time the same value** using the hebbian learning.

The weight matrix is **symmetric** and its updated in one shot (not learning trough epochs):

$$w_{ji} = \frac{1}{N}\sum_{\mu=1}^{M} \xi_{\mu,j}\xi_{\mu,i} \qquad \mathbf{W} = \frac{1}{N}\sum_{\mu=1}^{M} \xi_{\mu}\xi_{\mu}^{T}$$

neuron by neuron formulation — all weights together

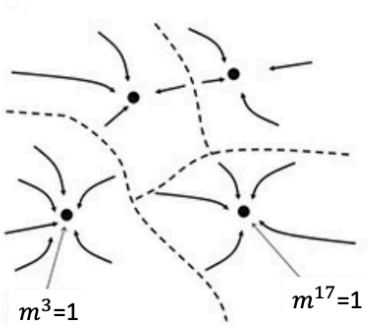
Important: in discrete formulation the **diagonal is filled with 0** (no self-feedback)

Retrieval Phase - Reconstructing from stored patterns

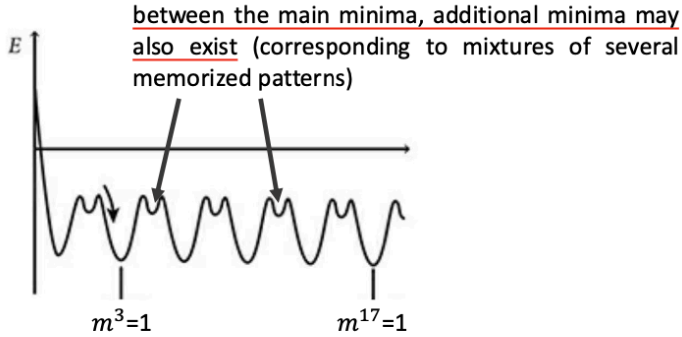
1. Provide an N-dimensional (probe) vector to the system. This can be an incomplete or noisy version of one of the stored patterns.
2. Initialize the states of the neurons using this probe vector $x(0) = \epsilon_{probe}$
3. For each epoch (steps over the same input data):
   3.1. Choose a random order of neuron update
   3.2. For each neuron following the order, update its state with $x_j(t+1) = \text{sgn}(\sum_{i=1}^{N} w_{ij}x_i(t))$
   3.3. Compute the set of overlap functions and the energy function
   3.4. Stop when a fixed point is reached (minimum of energy function reached or no more flips in the states of the neurons)

Attractor memories

The network dynamics is attracted toward a stable fixed point



The dynamics are attracted towards fixed points corresponding to memorized patterns (max value of overlap)



The energy function has multiple (equivalent) minima, each one corresponding to the retrieval of one pattern

- *left img:* prototypes are the fixed points of my dynamics:
- *right img:* correlation between patterns could complicate the learning creating suboptimal local minima (fake memories). This happens because patterns in our memory are NOT orthogonal

(Part 3 Lecture 1) MLP, Backprop, CNN

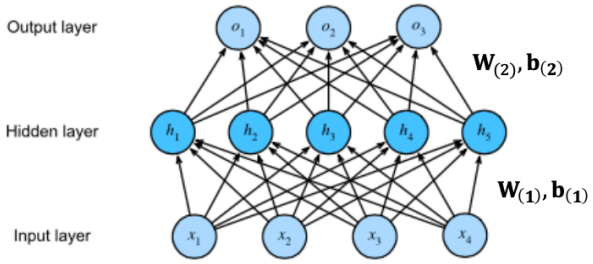
MULTILAYER PERCEPTRONS (MLP)

We can state that linear models are very limited because linearity in affine transformation is a strong assumption. For example it implies the weaker assumption of monotonicity in data. The **solution from DL** is to incorporate 1+ hidden layer.

A **Multilayer Perceptron** is a type of feedforward artificial neural network composed of an input layer, one or more hidden layers with nonlinear activation functions, and an output layer. It is fully connected and trained using backpropagation, allowing it to approximate complex nonlinear functions and solve a wide range of supervised learning tasks.

Input: $\mathbf{x} \in \mathbb{R}^{N_x}$
Hidden activation: $\mathbf{h} \in \mathbb{R}^{N_h}$,
Output: $\mathbf{o} \in \mathbb{R}^{N_o}$

input layer has same dimension of the input (ie if we need to classify a 32x32 image the input could be sized 1024)



Weights: $\mathbf{W}_{(1)} \in \mathbb{R}^{N_h \times N_x}$, $\mathbf{W}_{(2)} \in \mathbb{R}^{N_o \times N_h}$,
Bias: $\mathbf{b}_{(1)} \in \mathbb{R}^{N_h}$, $\mathbf{b}_{(2)} \in \mathbb{R}^{N_o}$

Weights and bias are the trainable parameters of the network.

Introducing NON-linearity

If we consider the MLP **<u>without</u> non-linear activation function** is the same as performing a huge linear transformation in 2 steps (passing trough h).

If $\mathbf{h} = \mathbf{W}_{(1)}\mathbf{x} + \mathbf{b}_{(1)}$ and $\mathbf{o} = \mathbf{W}_{(2)}\mathbf{h} + \mathbf{b}_{(2)}$ then we can compute directly $\mathbf{o}$ as:

$$\mathbf{o} = \mathbf{W}_{(2)}\mathbf{h} + \mathbf{b}_{(2)} = \mathbf{W}_{(2)}(\mathbf{W}_{(1)}\mathbf{x} + \mathbf{b}_{(1)}) + \mathbf{b}_{(2)} = \mathbf{W}_{(2)}\mathbf{W}_{(1)}\mathbf{x} + \mathbf{W}_{(2)}\mathbf{b}_{(1)} + \mathbf{b}_{(2)}$$

So we are not transforming anything in an hidden representation, **we're just making an affine transformation in 2 steps**.

Applying some non linearity after our hidden layers allows to obtain an **universal approximator:** this means that our model can approximate any non linear function in a compact domain.

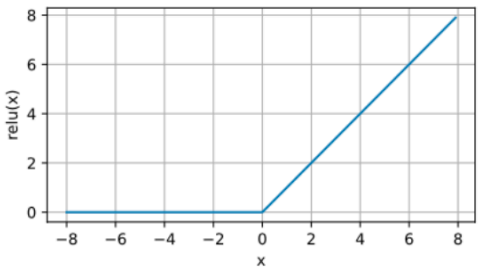
$$\mathbf{h} = \phi(\mathbf{W}_{(1)}\mathbf{x} + \mathbf{b}_{(1)})$$

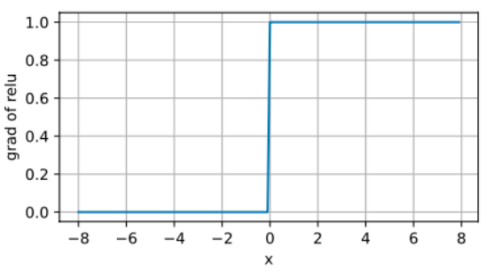
Non linear activation functions

This functions are inspired by the trend of the frequency current in the biological models

ReLU

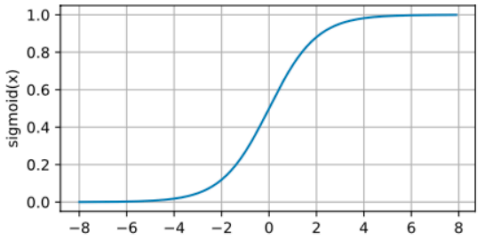
$$ReLU(x) = \max(0, x)$$

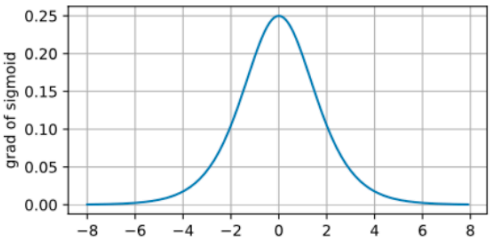




Sigmoid

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





Property:

$$\frac{d}{dx} sigmoid(x) = sigmoid(x)(1 - sigmoid(x))$$

Sigmoid function:

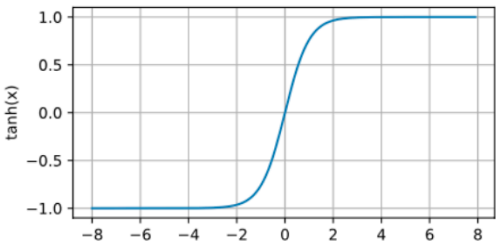
- **Linear regime near zero:** when the input is close to 0, the sigmoid function can be approximated by a straight line, as it changes almost linearly.
- **Non-linear intermediate regime:** between the linear region and saturation, the function exhibits a non-linear behavior, with more significant changes in output in response to input variations.
- **Saturation at extremes**: for very small (negative) or very large (positive) input values, the function enters saturation, **meaning changes in input have little to no effect on the output**.

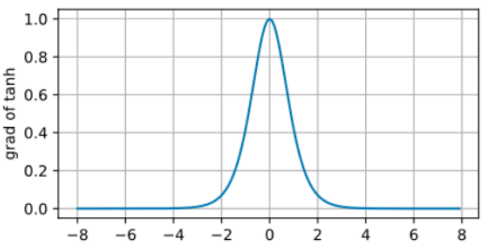
First order derivative:

- in **0** we have the **peak** of the derivative (highest modification in weights)
- If the regime is **non linear** the derivative is **small**
- if the regime is **saturated** the derivative is **0** (no weight update)

Hyperbolic Tangent

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$





Property:

$$\frac{d}{dx} tanhh(x) = 1 - tanh^{2}(x)$$

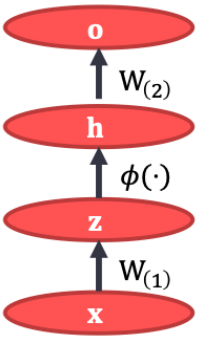
- similar to sigmoid rescaled to -1, +1 domain
- the peak of the first order derivative is 1 not in 0.25 (better for vanishing issues)

COMPUTATIONAL GRAPH

There're two type of propagations in NNs:

- **Forward propagation:** Calculation and storage of intermediate variables for a neural network, from the input layer to the output layer
- **Backward propagation:**
  - calculations involved in computing the adjustments to the parameters of the neural net architecture
  - The gradient of the loss wrt the components of the network is computed and is propagated backward in the NN architecture
  - Weigh update follows the rule: $W \leftarrow W - \eta \frac{\partial L}{\partial W}$ where the **loss** is the error measure, it measures the discrepancy between NN outputs and the target values

Forward Propagation →



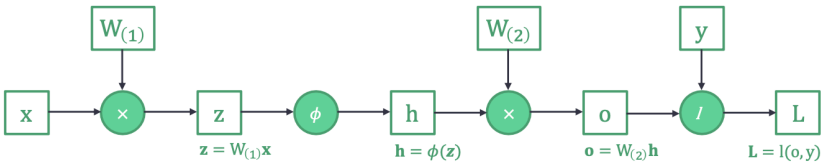
$\mathbf{o} = \mathbf{W_{(2)}}\mathbf{h}$ loss $\mathbf{L} = \mathrm{l}(\mathrm{o}, \mathrm{y})$

$\mathbf{h} = \phi(\boldsymbol{z})$

$\mathbf{z} = \mathbf{W_{(1)}}\,\mathbf{x}$

bias is ignored for simplicity

Computational graph: drawing a computational graph is helpful to visualize the dependencies of operators and variables involved in the computation performed by a neural network. Forward computation flows from left to right, backward computations from right to left



1 - hidden layer FFNN computational

There're two type of node in the graph:

- **circle nodes:** operations with the operation indicated
- **square nodes:** operators

Arrows are from operators to the operations they are involved to

Backpropagation

It's a method to calculate the gradient of neural nets parameters. The architecture, i.e., the computational graph, is traversed in **reverse order**, from the output to the input layer.

The gradients are computed according to the **chain rule:** $\frac{\partial Z}{\partial X} = \frac{\partial Y}{\partial X}\frac{\partial Z}{\partial Y}$

where in our case, Y is a squared node in between Z and X in our computational graph.

We need to store all the intermediate variables required while calculating the gradients

Gradients rules of thumb

1. $\frac{\partial \mathbf{W}\mathbf{x}}{\partial \mathbf{x}} = \mathbf{W}^T$ if you compute $\mathbf{W}\mathbf{x}$ then the result is a column vector $\mathbf{z}$ in which $z_1 = w[1] \cdot x_1 \ldots$ so if you derivate wrt to x you obtain the transpose of W

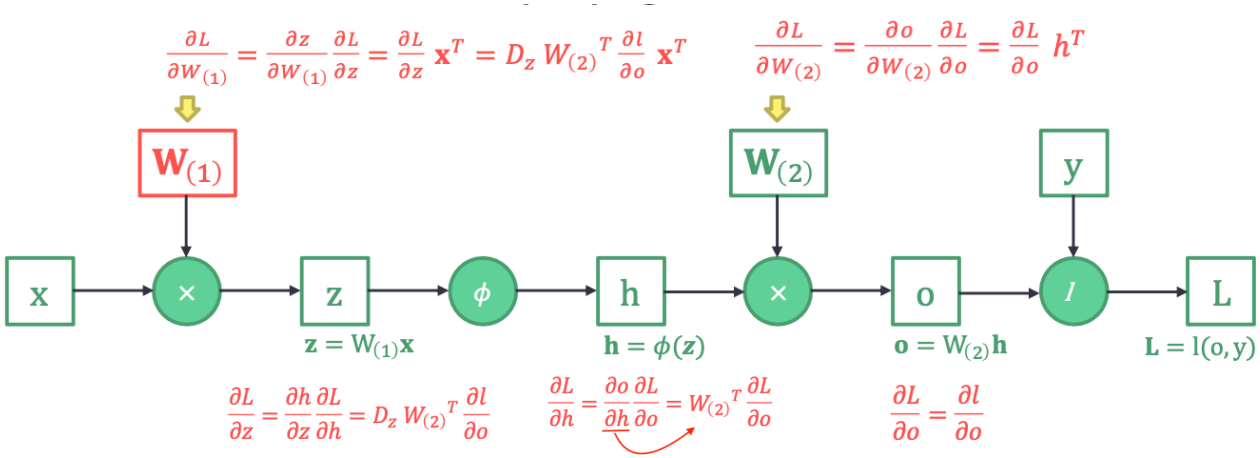
2. $\mathbf{h} = \phi(\mathbf{z})$ then $$\frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \frac{\partial \phi(\mathbf{z})}{\partial \mathbf{z}} = \begin{bmatrix} \phi'(z_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \phi'(z_{N_h}) \end{bmatrix} = D_z$$

This because $\phi(z)$ is a function that applies independently to each element of z $(h_i = \phi(z_i))$ so if we want to compute $\frac{dh_i}{dz_j}$ this derivative is $\phi'(z_i)$ if $i = j$ and 0 otherwise. So the jacobian, that is at the matrix of partial derivatives in 2D will be a diagonal matrix as other contributes are 0.

$$J = \begin{bmatrix} \frac{\partial h_1}{\partial z_1} & \cdots & \frac{\partial h_1}{\partial z_{N_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{N_h}}{\partial z_1} & \cdots & \frac{\partial h_{N_h}}{\partial z_{N_h}} \end{bmatrix} = \begin{bmatrix} \phi'(z_1) & 0 & \cdots & 0 \\ 0 & \phi'(z_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \phi'(z_{N_h}) \end{bmatrix}$$

3. If $\mathbf{z} = \mathbf{W}\mathbf{x}$ then $\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial \mathbf{z}}{\partial \mathbf{W}} \frac{\partial L}{\partial \mathbf{z}} = \frac{\partial L}{\partial \mathbf{z}} \mathbf{x}^T$

So now, given the value of the loss L we want to compute the gradient of the loss wrt our parameters ($W_{(1)}$, $W_{(2)}$) to update them.



Nota: the bias update is the same as the corresponding weight matrix but without the multiplication of the input term because the bias term multiplies a vector of ones instead of h

During training, forward and backward propagation depend on each other: during **forward propagation** the computational graph is traversed in the direction of the dependencies and all the variables values are computed along this path, **during backward propagation** the values computed in the forward pass hidden activations etc…) are used to propagate the gradients in the reversed order

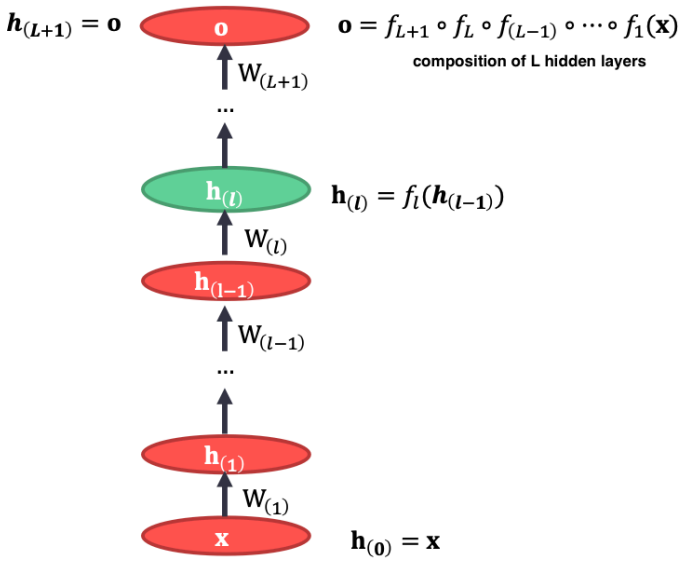
Training: after model parameters are initialized, we **alternate forward propagation** and **backpropagation.** Training requires more memory, because of storage of intermediate computations, than inference/prediction

Vanishing and Exploding gradients

If we consider a very deep architecture with L hidden layers, we can write the gradient of $\mathbf{o}$ wrt $\mathbf{W}_{(l)}$ as follows:

$$\frac{\partial \mathbf{o}}{\partial \mathbf{W}_{(l)}} = \frac{\partial \mathbf{h}_{(L+1)}}{\partial \mathbf{h}_{(L)}} \cdot \ldots \cdot \frac{\partial \mathbf{h}_{(l+1)}}{\partial \mathbf{h}_{(l)}} \cdot \frac{\partial \mathbf{h}_{(l)}}{\partial \mathbf{W}_{(l)}}$$

$$\mathbf{M}_{(L)} \cdot \ldots \cdot \mathbf{M}_{(l)} \cdot \mathbf{v}_{(l)}$$

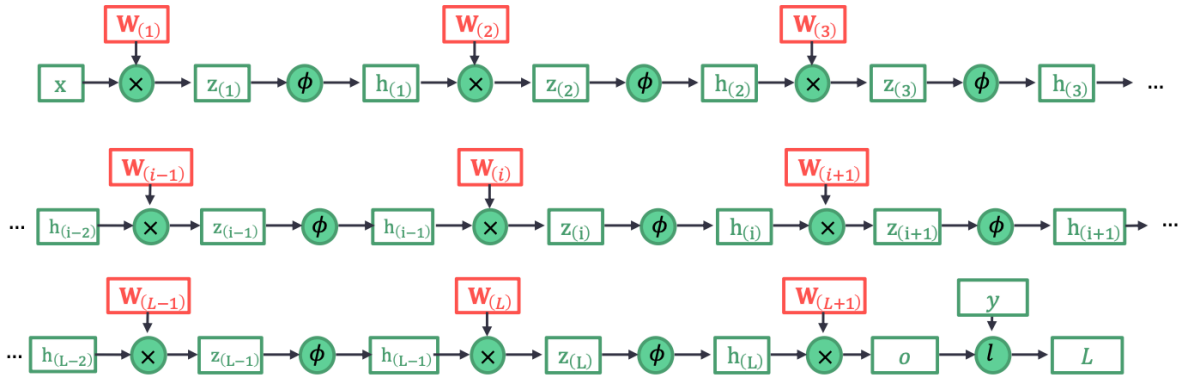


Where $\mathbf{M}$ are matrices and $\mathbf{v}$ is the input incoming in that layer. The resulting product can be very large or very small:

- **too large** (**exploding gradient**): the model is destroyed by the updates
- **too small** (**vanishing gradient**): learning becomes impossible as the parameters are essentially still the same

The **core idea** is that, starting from the first layer where the gradient becomes very small, the preceding layers stop learning effectively. This happens because the small gradient from the upper layer gets propagated backward, limiting the updates to the earlier layers.

Now let us consider a deep feed-forward neural network (MLP) with L hidden layers:



Lets compute the gradients:

output layer

$$\frac{\partial L}{\partial W_{(L+1)}} = \frac{\partial o}{\partial W_{(L+1)}} \frac{\partial L}{\partial o} = \frac{\partial l}{\partial o} h_{(L)}^T$$

$$\frac{\partial L}{\partial h_{(L)}} = \frac{\partial o}{\partial h_{(L)}} \frac{\partial L}{\partial o} = W_{(L+1)}^T \frac{\partial l}{\partial o} \qquad \frac{\partial L}{\partial z_{(L)}} = \frac{\partial o}{\partial z_{(L)}} \frac{\partial L}{\partial h_{(L)}} = D_{z_{(L)}} W_{(L+1)}^T \frac{\partial l}{\partial o}$$

$$\frac{\partial L}{\partial W_{(L)}} = \frac{\partial z_{(L)}}{\partial W_{(L)}} \frac{\partial L}{\partial z_{(L)}} = D_{z_{(L)}} W_{(L+1)}^T \frac{\partial l}{\partial o} h_{(L-1)}^T$$

layer L

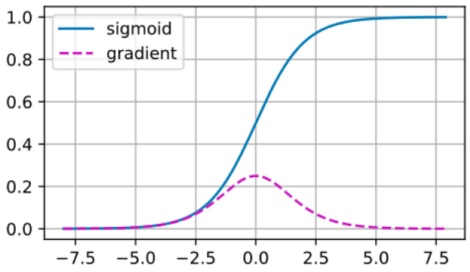
For a generic layer i:

$$\frac{\partial L}{\partial z_{(i)}} = D_{z_{(i)}} W_{(i+1)}^T D_{z_{(i+1)}} W_{(i+2)}^T \dots D_{z_{(L-1)}} W_{(L)}^T D_{z_{(L)}} W_{(L+1)}^T \frac{\partial l}{\partial o}$$

$$\frac{\partial L}{\partial z_{(i)}} = \prod_{k=i}^{L} \left(D_{z_{(k)}} W_{(k+1)}^T \right) \frac{\partial l}{\partial o}$$

$$\frac{\partial L}{\partial W_{(i)}} = \frac{\partial z_{(i)}}{\partial W_{(i)}} \frac{\partial L}{\partial z_{(i)}} = \frac{\partial L}{\partial z_{(i)}} h_{(i-1)}^T$$
$$= \prod_{k=i}^{L} \left(D_{z_{(k)}} W_{(k+1)}^T \right) \frac{\partial l}{\partial o} h_{(i-1)}^T$$

Vanishing typically depends on the diagonal matrix, when the inputs are too large or small. We've seen that the derivatives of the activation functions are typically very small so this term multiplied trough time could vanish the gradient



Nota: with sigmoid function the gradient can vanish unless the inputs to the sigmoids are almost zero at every layer (because the derivative is low also in the peak).

Possible solutions:

- with **ReLU** this problem is partially solved as the diagonal matrix is now the identity so no more vanishing on the tails of the function
- **residual NN**: other solution are the **skip connections** that allows to skip layers in the computation so that the gradient is propagated back from upper layers when computing the gradient skipping some layers in between

For the same reason the product of transpose of weight matrices can become too huge: **gradient exploding**

Numerical Stability and Initialization

The **choice** of the **initialization scheme** and **activation function** plays a significant role in learning & numerical stability. Poor choices in the initialization can cause exploding or vanishing gradients during training.

Random-weights initialization is key to break symmetry

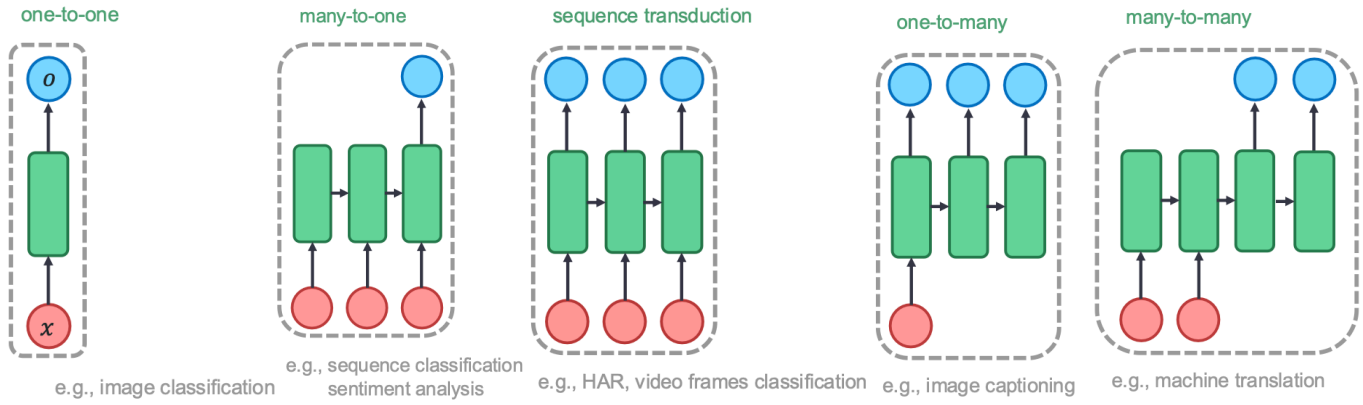
Xavier initialization & ReLU activation function are often used in deep ff architectures to avoid vanishing and exploding in the first training phases

LEARNING WITH SEQUENCES

Sequence tasks are crucial because many real-world data types—such as **text**, **audio**, and time-series signals—are inherently sequential. Capturing temporal dependencies allows models to make accurate and context-aware predictions. These tasks underpin key applications like machine translation, speech recognition, time-series forecasting, sentiment analysis and many others.

Temporal tasks and static tasks are fundamentally different: a temporal sequence is **order-sensitive**. Contextual information, i.e., dependencies across time steps or previous inputs, are crucial in such scenarios, as they directly influence the interpretation and prediction of future elements in the sequence.

Sequence modeling applications (Tasks)



Autoregressive models

Models that **regress the value of a signal on the previous values of that same signal**. The object of interest is $P(x(t) \mid x(t-1), \ldots x(1))$.

Problem: the number of inputs $x(t-1), \ldots, x(1)$ increases with t and if we treat our historical data as a training set, each example has a different number of features

There're two possible approaches:

- **Windowing:** predict the next element based on a fixed recent input window
- **Latent model:** create a fixed-size (state) representation of the sequence history. This latent representation is updated at each time step and the prediction is based on the latent representation.

Windowing / Tapped delay line

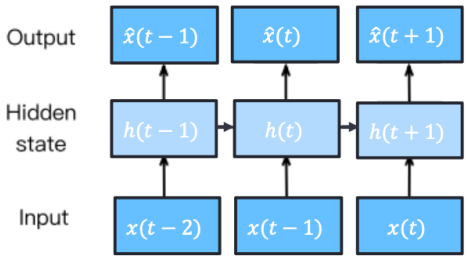
Assumption: when predicting the near future (in our setting next element), it is not necessary to look back in the input history up to the beginning of the sequence, it's enough to look τ steps before to predict next value. So to predict $x(t)$ we use $x(t-1), x(t-2), \ldots x(t-\tau)$.

This allows us to train any machine/deep learning model that requires **fixed-length vectors as inputs**

Latent autoregressive model:

Develop a model that uses some summary $h(t)$ of the past observations:

- update $h(t) = g(h(t-1), x(t-1))$
- estimate $x(t)$ with $\hat{x}(t) = P(x(t) \mid h(t))$



$h(t)$ is called **latent representation** as it's never observed externally

Sequence Modeling Task

The main idea is to estimate the joint probability of the entire sequence $P(x_1, \ldots, x_T)$ with a model. This problem can be reduced to an **autoregressive prediction** by the chain rule of probability:

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

Markov model (fixed time window)

Sometimes it makes sense to condition only on the previous k time steps rather the entire history up to first time step. This can be done exploiting the **markovian condition** that states that the future is conditionally independent of the past, given the recent history

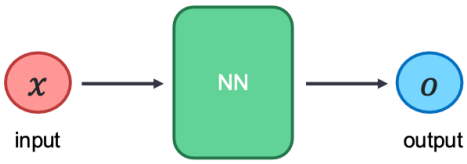
k determines **the order of the Markov model**:

- k-th order Markov model: $P(x_1, \ldots, x_T) = P(x_1) \prod_{t=2}^{T} P(x_t \mid x_{t-1}, \ldots, x_{t-k})$

Re-visiting FFNN to deal with time

No notion of sequence / causality

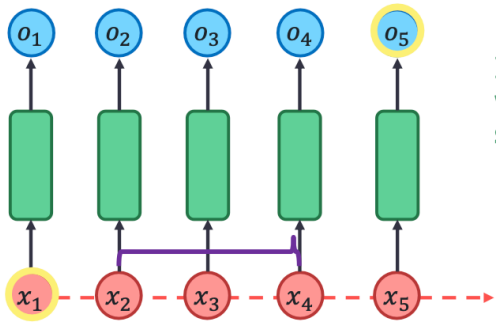
$$\mathbf{o} = f(\mathbf{W}_{(2)} \phi(\mathbf{W}_{(1)} \mathbf{x} + \mathbf{b}_{(1)}) + \mathbf{b}_{(2)})$$



Idea: apply the same architecture to the whole input series, time step per time step **ignoring temporal dependencies**

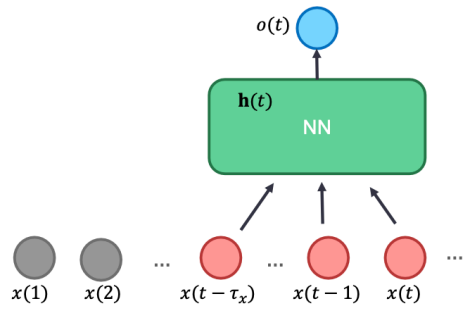
Trivial solution to consider temporal dependencies: buffering (windowing) the input capturing short-extent local structure. Consider that with this approach the only dependencies considered are the ones inside the window, outside of the window temporal dependencies are ignored

$$\mathbf{o}(t) = f(\mathbf{x}(t), \mathbf{x}(t-1), \mathbf{x}(t-2))$$



Time delay NN

Following this idea we can create our FFNN on a temporal task buffering the input on the NN



$$\mathbf{o}(t) = f(\mathbf{x}(t), \mathbf{x}(t-1), \ldots, \mathbf{x}(t-\tau))$$

Concatenation

$$\mathbf{h}(t) = \phi(\mathbf{W}_{(1)} \begin{bmatrix} - & \mathbf{x}(t) & - \\ - & \mathbf{x}(t-1) & - \\ & \vdots & \\ - & \mathbf{x}(t-\tau_x) & - \end{bmatrix} + \mathbf{b}_{(1)})$$

$$\mathbf{o}(t) = \mathbf{W}_{(2)}\mathbf{h}(\mathrm{t}) + \mathbf{b}_{(2)}$$

the window is also called **shift register** or **tapped delay line** of order τ

Pro: if the buffer includes sufficient information, it is possible to solve the temporal task with standard BP training

Cons: τ_x is an hyperparameter to be optimized in order to maximize the gain from the window.

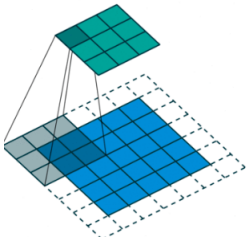
The **memory depends linearly both on window and input size**. This because the inputs $\mathbf{x}(t)$,… $\mathbf{x}(t-\tau)$ are concatenated so the longer the window size the more weights we need to use to transform the input

Convolutional NN

Convolutional Neural Networks (CNNs) are deep learning models designed to process grid-like data, especially images. They excel at capturing spatial patterns through convolutional layers.

Convolutional Neural Networks (CNNs) take inspiration from neuroscience, particularly from the structure of the **visual cortex**. In the visual cortex, **simple cells** respond to oriented edges and bars at specific locations, while **complex cells** combine inputs from simple cells, allowing for **positional invariance** and larger **receptive fields**. CNNs mimic this layered, hierarchical processing by using filters to detect low-level features and progressively build more abstract representations through deeper layers.

Convolutional layer



The core building block of a CNN is a **learnable filter**, or **kernel**, a small matrix with a limited receptive field that spans the full input depth, which is **convolved across the input's** spatial dimensions to compute dot products and produce feature maps.

How convolution is performed? The kernel is placed over the top-left corner of the image, where each of its elements is multiplied by the corresponding pixel it overlaps. The resulting products are summed to produce a single output value, which becomes a pixel in the feature map. The kernel is then slide horizontally by a fixed stride, and the process is repeated across the entire image.

Convolutional layers coincides with **simple cells**

Useful notions in convolution:

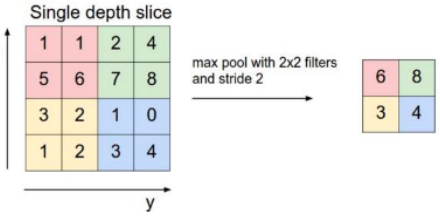
- **Stride:** the number of pixels shifts over the input matrix. Larger strides result in smaller feature maps
- **Padding:** adds pixels/elements around the input matrix/image. It's used to control the spatial size of the output volumes
- **Dilation**: a parameter that controls the spacing between the values in a kernel

Pooling layer

Applying a fixed operation over a sliding window.

Common pooling operations are **max, avg**…

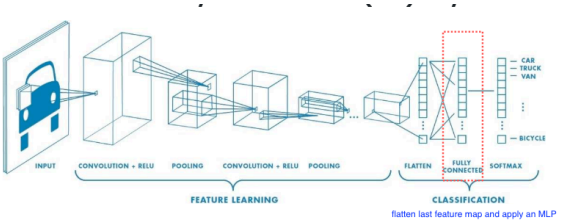
It reduces the spatial dimensions (width and height) of the input volume for the next convolutional layer.



Pooling layers coincides with **complex cells**

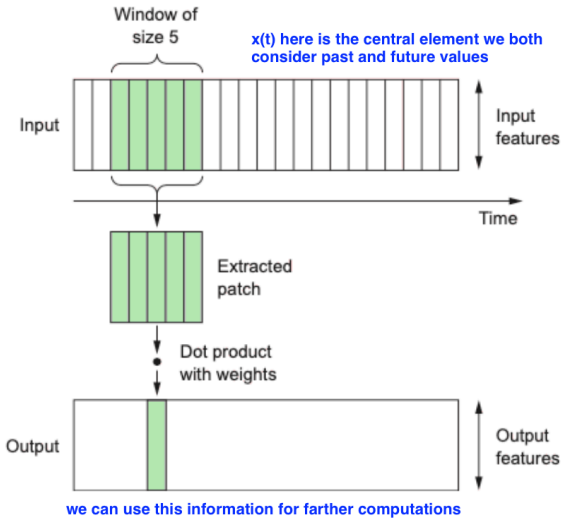
CNN – Fully connected (FC) layer

At the end of the architecture a **flatten** layer is applied to flatten the last feature map. This flatten representation is passed trough a FFFCNN to classify the input. Between pooling and convolutional layer some non linear functions are applied (typically ReLU)



CNN for sequence processing: 1D-CNN

It consists of a kernel applied over the sequence (like temporal delay NN from a different perspective) so the main idea is to apply convolution operations to temporal data.
Typically they operate in a **non-casual scenario** where we can consider both past and future timesteps in our convolution by centering the convolution and considering n time seps before and ahead the current one.



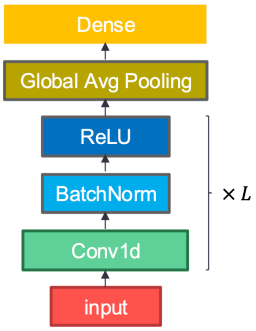
Recall: convolution is a linear operation (can be parallelized), the non linearity is typically applied after the convolutional layer.

- **Kernel**: a one-dimensional filter of fixed length that moves along the time series, computing dot products between the filter and local segments of the sequence at each position.
- **Stride**: defines how many time steps the filter shifts forward at each step of the convolution.
- **Padding**: extra values added to the edges of the time series to allow the convolution to include boundary elements in the computation.
- **Dilated convolution**: enhance the receptive field of the filter by dilating the filter over the input time serie. It consists in inserting spaces between each element of the kernel, a dilation of d means that there are $d-1$ spaces between each element of the kernel

Pooling layers: reduce the dimension of the feature map. In 1D CNNs:

- **Max Pooling**: takes the max over each window (i.e., pool size) → return a sequence
- **Average Pooling**: takes the mean over each window (i.e., pool size) → return a sequence
- **Global Pooling**: computes the fixed operation (e.g., max or avg) over the entire sequence it's particularly useful to **compress the whole sequence to a single value** (for example to classify the sequence…)

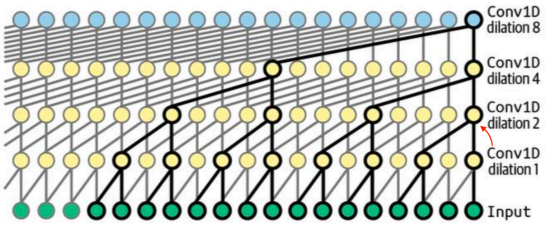
1D CNN - architecture



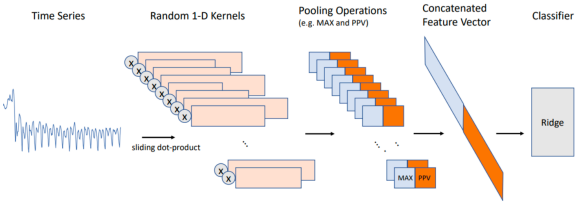
Time is treated as a spatial dimension, **1d-convolution** allows to extract local 1d subsequences while **1d-pooling** allow to subsample from the representation created using kernels.After the 1D convolution, a **non-linearity** (such as ReLU) is applied to introduce complexity into the model's representations. This is typically followed by a trainable normalization layer, such as **Batch Normalization**, which helps stabilize and accelerate training by **reducing internal covariate shift**. It also improves generalization and allows the use of higher learning rates. The architecture and configuration of the **dense (fully connected) layers** that follow depend on the specific task

Some well known architectures:

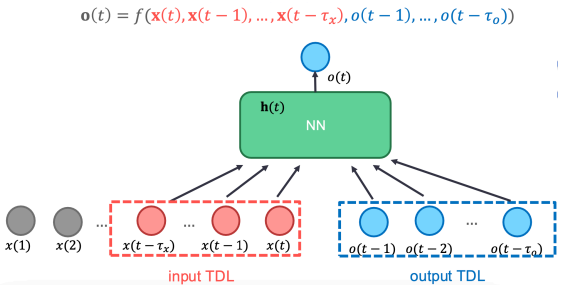
- **Wavenet:**
  - **Idea:** stacking multiple 1d- convolutional layers with increasing dilation trough layering
  - 1st layer relay on very local informations
  - from 2nd layer on a dilation is applied so the receptive field is augmented, with a convolutional kernel of the same size we are able to enlarge the general view of the model on the original input



- **ROCKET:**
  - **Idea:** a very effective 1d-CNN for time series classification
  - **Architecture:** a large number of **random linear kernels** (random size, weights, bias, dilation, padding), then a **global max pooling & ppv** (% of positive values) and finally a **linear output layer** to classify the time series
  - The idea is that using a huge number of random 1D kernels with random stride padding etc, followed by pooling operator is enough to create a discriminative representation of the time series that can be passed trough a simple linear readout that enable to classify it.



- **Nonlinear AutoRegressive with eXogenous inputs (NARX):**
  - A time delay NN that exploit 1D convolution but in this case we give in input not only the window on the current time step but also the previous value of the output
  - During training can rely on **teacher forcing** (use the expected output values instead of the output ones) to enforce the training phase
  - **PRO** : vanishing gradients issues are mitigated by the output TDL, which **determines jump ahead connections when the network is unfolded in time** (the architecture shape directly creates one-hop gradient flowing backward from output to hidden layers)



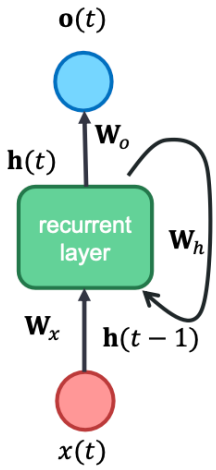
(Part 3 Lecture 2) RNN, BPTT

So far, we have addressed tasks on sequences using a **windowing approach**, inspired by **Markov models**. The idea is to approximate the conditional probability $P(x_t | x_{t-1}, \dots x_1)$ with a truncated version $P(x_t | x_{t-1}, \dots x_{t-n})$, assuming independence from values that are too far in the past. If we want to incorporate in the model too far informations we need to increase n, but increasing it, the number of trainable parameters increases (as the number of parameters depends on the dimension of the window).

In this section, we're going to introduce **Recurrent Neural Networks** (**RNNs**), a powerful approach inspired by **latent variable models.** In this case we try to approximate the probability of generating the new value in the sequence $P(x_t | x_{t-1}, \dots x_{t-n})$ with $P(x_t | h_t)$ using an hidden state h_t that stores the sequence information up to time step $t-1$. h_t is a function of current input and previous latent representation ($h_t = f(x_t, h_{t-1})$.

RECURRENT NEURAL NETWORKS (RNN)

RNNs are **neural networks with hidden states**. At each time step t, we have an input $x(t) \in \mathbb{R}^{N_x}$.



The hidden layer its computed by applying a non linear activation function ϕ to the linear combination of both input value at current time $\mathbf{x}(t)$ and hidden representation at previous time $\mathbf{h}(t-1)$. The **state transition function** is computed as: $\mathbf{h}(t) = \phi(\mathbf{W}_x \mathbf{x}(t) + \mathbf{W}_h \mathbf{h}(t-1) + \mathbf{b}_h)$.

RNNs are called **recurrent** as the computation $\mathbf{h}(t)$ of is the same at each time step and it involves the hidden state at previous time. $\mathbf{h}(t)$ captures historical information of the sequence up to the current time step.

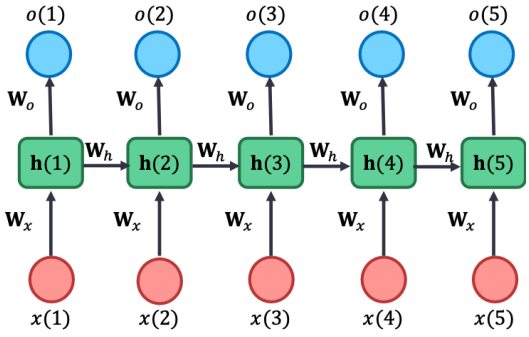
The output layer is computed as $\mathbf{o}(t) = \mathbf{W}_o \mathbf{h}(t) + \mathbf{b}_o$ and non linearities can be applied depending on the task.

Weight sharing: the parameters used to compute the hidden state are the same for all the time steps. For this reason **the parametrization cost does not scale with the number of time steps**

Properties of RNNs:

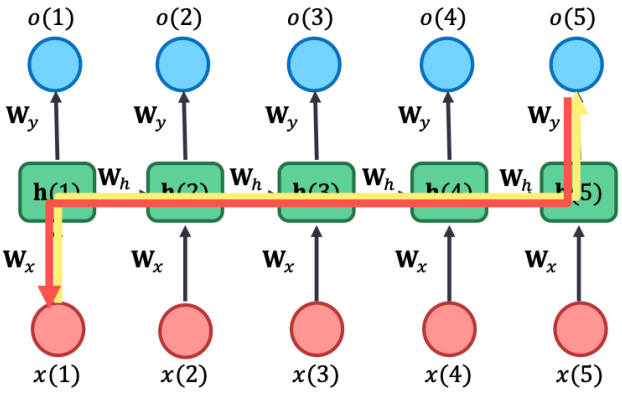
- **Causality:** a system is causal if the output at time t_0 only depends on input at time $t < t_0$
- **Stationarity:** time invariance of the state transition function. The operation performed by the recurrent layer is the same for every time-step , **regardless of the length of the sequence**. For this reason RNNs could process data of variable length with a fixed-size model
- **Adaptivity:** the state transition function is realized by a neural net with free parameters, hence it is learnt from the data

Unfolding of RNN trough time →



Conceptually: a very deep NN where # of layers corresponds to # of time steps in the input sequence
Issues with RNNs

Both in **forward propagation** (of the input) and in **backward propagation** (of the gradients) issues could happens due to the "deepness" of the unfolded network.
In the forward pass, suppose that the input $x(1)$ is relevant for the output $o(5)$. In this case, the hidden state $h(5)$ must encode information from $x(1)$. However, this poses a challenge, as the signal from $x(1)$ must propagate through multiple non-linear transformations to influence $h(5)$, potentially leading to information loss or degradation. A similar issue arises in the backward pass, where the contribution of a distant output to the gradient of an earlier input becomes increasingly attenuated (**gradient vanishing** or **exploding**).



in short: shows the same issues as deep feed forward neural nets

How to prevent exploding: gradient clipping

When optimizing some objective function l wrt $\mathbf{w}$, we push $\mathbf{w}$ in the direction of the negative gradient $\mathbf{g}$: $\mathbf{w} \leftarrow \mathbf{w} - \eta\mathbf{g}$. If l is Lipschitz continuous with constant L then:

$$||l(\mathbf{x}) - l(\mathbf{y})|| \leq L\, ||\mathbf{x} - \mathbf{y}||$$
$$||l(\mathbf{w}) - l(\mathbf{w}) - \eta\mathbf{g}|| \leq L\, \eta\, ||\mathbf{g}||$$

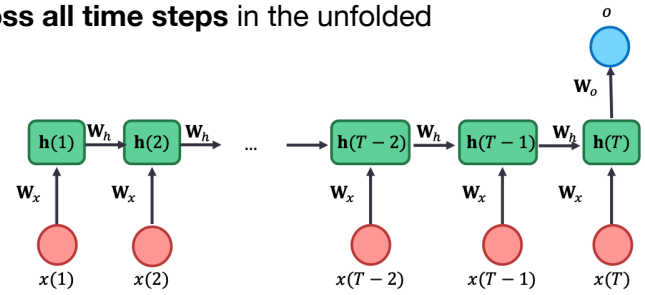
So when $||\mathbf{g}||$ is excessively large we have the problem of exploding gradients. The solution is to **clip the gradient** so that its magnitude never exceeds a threshold θ, while preserving its direction, ensuring the update remains aligned with the original gradient $\mathbf{g}$.

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{||\mathbf{g}||}\right)\mathbf{g}$$ ie project the gradient $\mathbf{g}$ onto a ball of radius θ

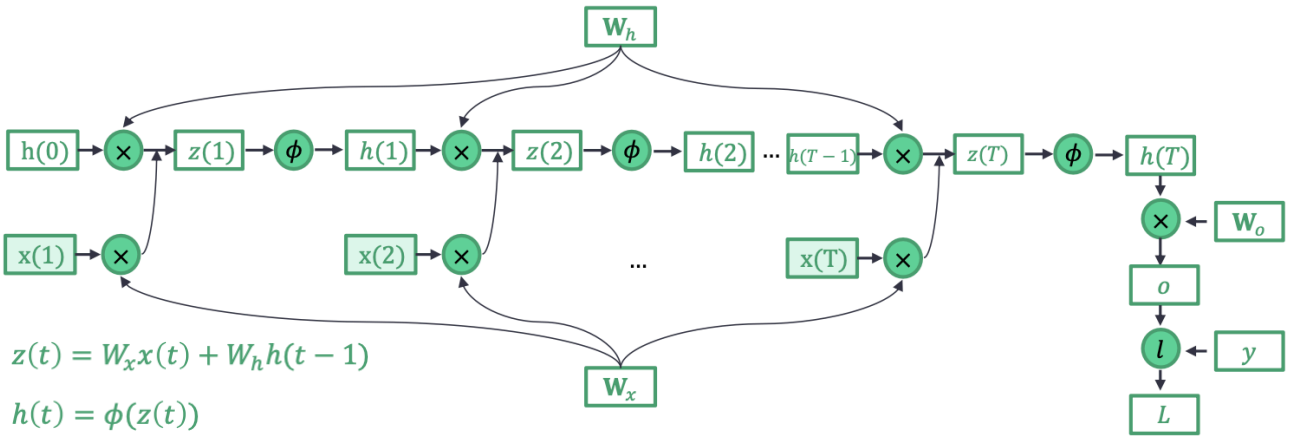
BACKPROPAGATION TROUGH TIME (BPTT)

Backpropagation Through Time (BPTT) is the training algorithm for RNNs. It works by unfolding the network across time steps, transforming it into a feed-forward network with shared parameters. Gradients are then backpropagated through this unfolded structure, and **the contributions to each parameter are aggregated across all time steps** in the unfolded computational graph.

For the sick of simplicity let us consider a **many-to-one task on sequences** like classification or regression task. We have M sequences of max length T in a **supervised setting:** one output value for each sequence. Unfolding the NN trough time we obtain the deep NN in the right image.



Computational graph of RNNs



We need to compute the loss wrt W_h and wrt to W_x

Gradients rule of thumb

(Recall)

1. $\frac{\partial \mathbf{W}\mathbf{x}}{\partial \mathbf{x}} = \mathbf{W}^T$

2. $\mathbf{h} = \phi(\mathbf{z})$ then $\frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \frac{\partial \phi(\mathbf{z})}{\partial \mathbf{z}} = \begin{bmatrix} \phi'(z_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \phi'(z_{N_h}) \end{bmatrix} = D_z$

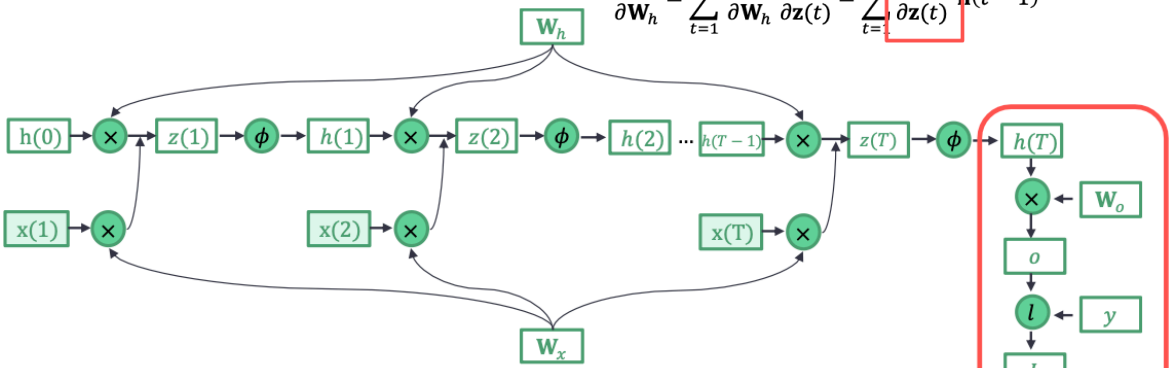
3. If $\mathbf{z} = \mathbf{W}\mathbf{x}$ then $\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial \mathbf{z}}{\partial \mathbf{W}} \frac{\partial L}{\partial \mathbf{z}} = \frac{\partial L}{\partial \mathbf{z}} \mathbf{x}^T$

(New)

4. $f(x, y, z)$ with x, y, z being function of t $\rightarrow$ $\frac{df}{dt} = \frac{\partial x}{\partial t}\frac{\partial f}{\partial x} + \frac{\partial y}{\partial t}\frac{\partial f}{\partial y} + \frac{\partial z}{\partial t}\frac{\partial f}{\partial z}$

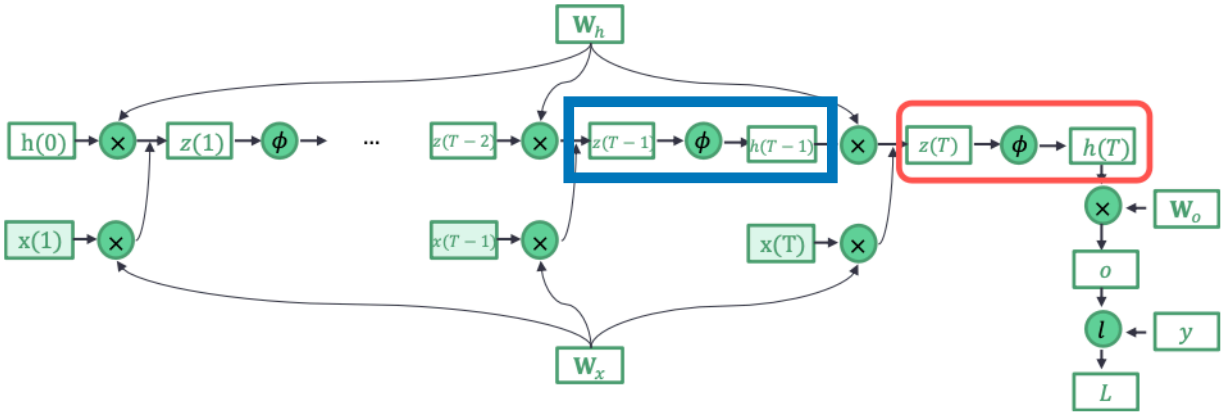
Gradients in RNNs

$$\frac{\partial L}{\partial \mathbf{W}_h} = \sum_{t=1}^{T} \frac{\partial \mathbf{z}(t)}{\partial \mathbf{W}_h} \frac{\partial L}{\partial \mathbf{z}(t)} = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{z}(t)} \mathbf{h}(t-1)^T$$



$$\frac{\partial L}{\partial \mathbf{W}_x} = \sum_{t=1}^{T} \frac{\partial \mathbf{z}(t)}{\partial \mathbf{W}_x} \frac{\partial L}{\partial \mathbf{z}(t)} = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{z}(t)} \mathbf{x}(t)^T$$

$$\frac{\partial L}{\partial \mathbf{W}_o} = \frac{\partial o}{\partial \mathbf{W}_o} \frac{\partial L}{\partial o} = \frac{\partial l}{\partial o} \mathbf{h}(T)^T$$



- $\frac{\partial L}{\partial h(T)} = \frac{\partial o}{\partial h(T)}\frac{\partial L}{\partial o} = W_o^T \frac{\partial l}{\partial o}$
- $\frac{\partial L}{\partial z(T)} = \frac{\partial h(T)}{\partial z(T)}\frac{\partial L}{\partial h(T)} = D_{z(T)} W_o^T \frac{\partial l}{\partial o}$

going one step back in time the computation of the derivative of the loss wrt $z(t)$ is exactly the same but considering W_h and not W_o:

- $\frac{\partial L}{\partial h(T-1)} = \frac{\partial z(T)}{\partial h(T-1)}\frac{\partial L}{\partial z(T)} = W_h^T D_{z(T)} W_o^T \frac{\partial l}{\partial o}$
- $\frac{\partial L}{\partial z(T-1)} = \frac{\partial h(T-1)}{\partial z(T-1)}\frac{\partial L}{\partial h(T-1)} = D_{z(T-1)} W_h^T D_{z(T)} W_o^T \frac{\partial l}{\partial o}$

For a generic time step t:

$$\frac{\partial L}{\partial z(t)} = D_{z(t)} W_h^T D_{z(t+1)} W_h^T \dots D_{z(T-1)} W_h^T D_{z(T)} W_o^T \frac{\partial l}{\partial o}$$

$$= \prod_{k=t}^{T-1} \left(D_{z(k)} W_h^T\right) D_{z(T)} W_o^T \frac{\partial l}{\partial o}$$

unfolding in time — output computation

resulting total gradients for the weight matrices:

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^{T} \prod_{k=t}^{T-1} \left(D_{z(k)} W_h^T\right) D_{z(T)} W_o^T \frac{\partial l}{\partial o} h(t-1)^T$$

$$\frac{\partial L}{\partial W_x} = \sum_{t=1}^{T} \prod_{k=t}^{T-1} \left(D_{z(k)} W_h^T\right) D_{z(T)} W_o^T \frac{\partial l}{\partial o} x(t)^T$$

Training a RNN faces the same gradient **vanishing/exploding** gradient issue as analyzed in deep FF neural networks. $\frac{\partial L}{\partial z(t)}$ can become negligibly small or excessively large as we go back in time (propagating gradients back)

Time complexity of an epoch on a single time series (per sequence): $O(TN_h^2)$
Overall complexity for E epochs on an M sized dataset: $O(EMTN_h^2)$
Where T is the length of the longest time-series

Truncated BPTT

BPTT consists in alternating forward and backward passes through time and requires storing all the intermediate values from $t = 1$ until $t = T$. For this reason **can become impractical for arbitrary long time-series**.

Truncated BPTT consists in truncating the BP after τ time-steps, avoiding the need to backpropagate through the entire sequence, thus reducing computational cost and memory usage.

resulting total gradients for the weight matrices:

$$\frac{\partial L}{\partial W_h} = \sum_{t=T-\tau+1}^{T} \prod_{k=t}^{T-1} \left(D_{z(k)} W_h^T\right) D_{z(T)} W_o^T \frac{\partial l}{\partial o} \, h(t-1)^T$$

$$\frac{\partial L}{\partial W_x} = \sum_{t=T-\tau+1}^{T} \prod_{k=t}^{T-1} \left(D_{z(k)} W_h^T\right) D_{z(T)} W_o^T \frac{\partial l}{\partial o} \, x(t)^T$$

BPTT – discussion

BBTT has potential **slow convergence** and **can converge in local minima** as any gradient descent approach, because relies on local optimizations. Learning can be challenging due to the dynamical nature of the system, as the evolving states over time may cause stability issues during training. Finally it suffers of gradient **vanishing/exploding** because of the unfolding trough time,

Learning **long term dependencies** could be very challenging

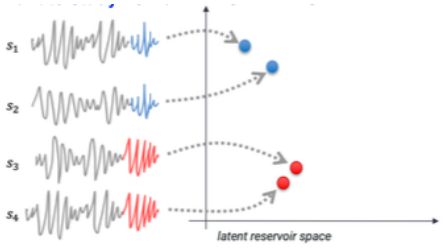
BPTT alternative: Real-time Recurrent Learning RTRL

RTRL is an online algorithm that updates RNN weights at each time step via exact gradient computation without caching past activations. At the cost of an **higher computational** time complexity ($O(N_h^4)$), but **more efficient** in **memory requirements** (does not depend on the length of the time-series)

RNNs – Computational Power

Recurrent Neural Networks, in the vanilla formulation, are naturally suitable to simulate finite-state automata. For the **universal approximation theorem** we know that "all Turing machines may be simulated by fully connected RNNs build on neurons with **sigmoidal activation functions**"

RNNs – Architectural Bias



Architectural bias in RNNs arises from the very structure of the network and dictates which state-space mappings are even possible, **before any learning takes place**. Even with small, random initial weights, an RNN naturally implements a form of **Markovian memory**: input sequences that share the more the same suffix are driven to closer points in the hidden-state space (as in the image on right). As a result, the network already clusters time-series by their last time-steps, producing meaningful representations without weight updates.

This predisposition toward a Markovian organization holds true even after training (for this is called bias), so it defines the baseline "lens" through which any subsequent learning must operate. Studying these intrinsic dynamics—how an untrained RNN encodes temporal patterns—helps us understand and leverage the computational capabilities built into echo-state networks.

Very useful for sequence classification: this is useful for ESN, RNNs in the reservoir clusterize the sequences and then a simple linear readout layer is enough to separate them (but this could depend on the task at hand)

This **not apply well for language-based problems** as suffixes are not so relevant in language tasks

Limitations of backpropagation:

- **Memory overhead** for storing the neural activations
- **Sequentiality and synchronicity** of forward & backward passes: to backpropagate i need all the computation done in forward steps, so first perform forward computations then propagate back
- **Lack of modularity**: sparsity is enforced with post-hoc pruning or pre-designed block-wise architectures
- **Biological implausible:**
  - no evidence for **derivatives propagation** and storing **neural activations**
  - no evidence for **mirrored connections** (top-down ≠ bottom-up): ie there's no plausibility for using W_h for forward pass and W_h^T for backward pass
  - No evidence for two separate circuits (forward + backward)

Weight transport

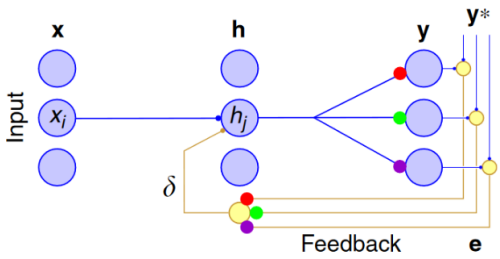
In standard neural networks, learning happens through two key phases: the **forward pass** and the **backward pass**. These involve two separate computational flows, but they rely on **symmetric information** to function correctly.

Forward pass: $h_l = \sigma(W_l h_{l-1} + b_l) \quad \forall l = 2,\dots,L$**,** the signal moves from the input toward the output.

Backward pass: $\delta_l = (W_{l+1}^T \delta_{l+1}) \circ \sigma'(a) \quad \forall l = 1,\dots,L-1$, used in backpropagation to compute gradients, where σ' is D in previous notation and δ_{l+1} is the error propagated from upper layer so the derivative wrt to the input from the upper layer in terms of computational graph.

This implies **two circuits** (forward & backward) that shares the **same weight values** and the **same neuron activations.**

This requires the **transport of weight information** from the forward to the backward pathway. In real biological or hardware systems, this is not trivial and **often implausible**.

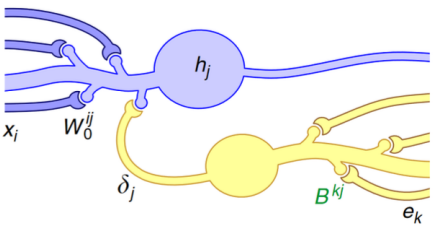


This also introduces a strong **sequential dependency** in the learning process: **you must halt the forward computation to perform the backward pass**. This is not feasible in real-world scenarios where inputs arrive continuously over time.

LEARNING BEYOND BACKPROPAGATION (integrated with prof Cossu Lecture)

(1) Feedback Alignment

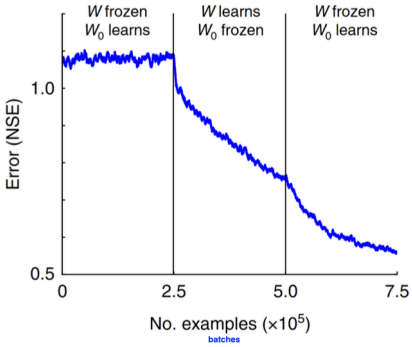
Feedback alignment allows to **remove weight transport** by introducing a **random feedback weight** matrix B. This matrix is selected random and fixed to replace W^T in backward computations. This allows to **decouples the forward and the backward passes**. FA pushes the weights in a similar direction wrt backpropagation but how?

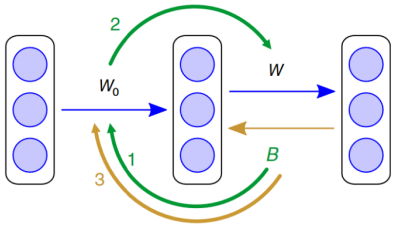


Three different version of feedback alignment exists:

1. **FA**: W_2 and W_3 are no more transposed and used in backpropagation circuits, but we use B_2 and B_3 random matrices. This process is still **synchronous** because of the backpropagation of gradient trough layers. Practically W_l **aligns with** B_l^T but there's **no**

theoretical foundations behind this results. In general FA update i**s not the gradient of any function** → cannot guarantee to follow any minimization path.





$\Delta W = eh^T = e(W_0 x)^T = ex^T W_0^T$

$\Delta W_0 \propto Bex^T$

Intuition: during an ablation study, they proves that the backward weights influence forward weights, causing weight alignment. They consider a network with W_0 input weights and W hidden weights. They started frozen W and learning W_0 then the opposite and alternates. As it's possible to observe in the graph the model in first phase does not learn and then start learning. In the third phase, although the setting is the same as in the first phase, the model is able to learn, how come?

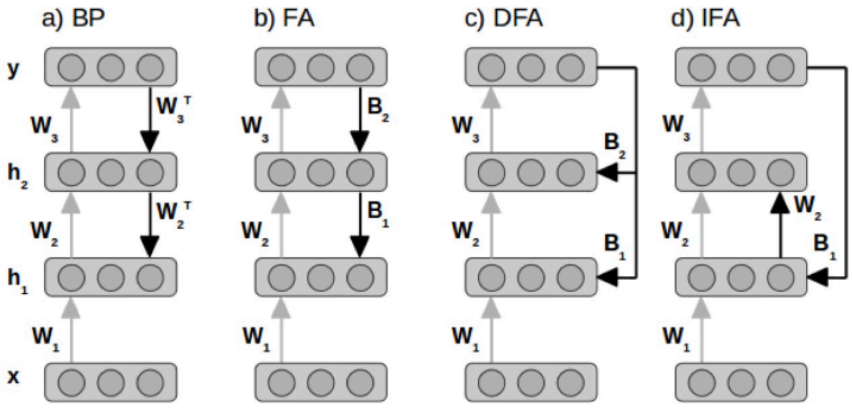
W_0 is trained using the backward signal B, which means it implicitly captures and transmits information from B. As a consequence, the forward weights W begin to align with B, since gradient information from B becomes accessible through W_0. Initially, W cannot learn because it is frozen. However, once it is allowed to update, the network begins to learn by leveraging the information already embedded in W_0. In later phases, even if W is frozen again, learning can still occur through updates to W_0 , as information continues to propagate via B.

FA typically converges to BP updates and has good results but still **can underperform in complex scenarios**.

2. **DFA**: instead of propagating one layer before the other, they propagate the gradient from output layer directly to hidden layer, using random matrices. Each layer **could work in parallel in propagating back the gradient**.

3. **IFA**: propagate the gradient to first hidden layer and then propagate forward using the same weight matrix. **Less effective wrt other methods** and also **synchronous** because gradient should flow trough layers. With DFA we're loosing some expressive power wrt to backpropagation because we simplify too much. Indirect method reintroduce some sequentiality

The key insight behind these methods is that the exact way the error is propagated—such as multiplying it by weight matrices—is less crucial than simply ensuring that the network receives information about the output error. What's important is that **some form of error signal reaches the internal layers to guide learning**, even if it's approximate.



(2) Evolution-based approach

Natural selection drives evolution when the following necessary and sufficient conditions are met:

- **Variation**: Individuals within a population exhibit different traits.
- **Differential Reproduction:** not all individuals reproduce at the same rate. This is often due to environmental factors that favors certain traits. Importantly, it's the survival of individuals that are *fit enough* (not necessarily the absolute "fittest") that leads to differential reproduction.
- **Heredity:** traits can be passed on from parents to offspring. While offspring are different, they also share similarities with their parents due to heredity.

Outcome of Natural Selection: advantageous traits, which increase an individual's likelihood of survival and reproduction, become more prevalent in the population over time. Conversely, less advantageous traits tend to disappear. This process, where traits that make individuals more likely to survive are passed on, is the core idea of natural selection.

Digital evolution: the main idea is to generate digital simulation of evolution process, driven by an iterative process over generations. It **starts from initial conditions** and an **environment** and follow and **evolution algorithm**. If we can simulate this process we can “create” artificial life

Evolution General Algorithm:

EAs are **iterative optimization algorithms** inspired by natural selection. The basic structure involves the following steps:

Initialize Population P_0 of Individuals with (N) Features (in ML individuals are NN, features are for example parameters because they are initialized differently)

Evaluate Fitness of Population P_0: assess the quality or "fitness" of each individual in the initial population (for us could be performance of NNs on some task)

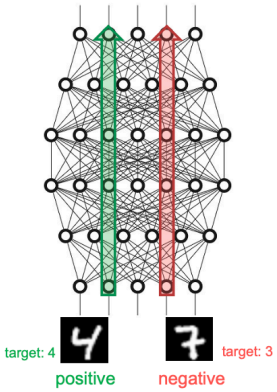
For each generation (t = 1, 2, ...):

- **Select** Parents from (P_{t-1}) (e.g., based on fitness): there are various parent selection methods, and a parent can be selected multiple times (selection is often **biased towards individuals with higher fitness**, giving them a greater chance to reproduce)
- **Reproduction** and **recombination** of Features to get Children (not really present in EP): parents produce offspring (children) by combining their features through a process called recombination or crossover. An example for us could be sum-up weights value of parent NNs
- Apply **Mutation** to Features (e.g., at random, adaptive mutation rate): introduce small, random changes (mutations) to the features of the offspring. Mutation helps to maintain diversity in the population and explore new regions of the search space (mutation rate can be fixed or adapt during the algorithm's execution).
- **Evaluate Fitness** of New Population: assess the fitness of the newly generated offspring.
- Select Subset of (K) **survived** Individuals to get (P_t) (e.g., random sample weighted by fitness): choose (K) individuals from the combined pool of parents and offspring (or just offspring, depending on the strategy) to form the population for the next generation ((P_t)). Selection is often based on fitness, giving higher-performing individuals a greater chance of survival. Parents could survive or not depending on the strategy

Nota: **no derivatives in this approach**, we could use any function (**also non differentiable** ones)

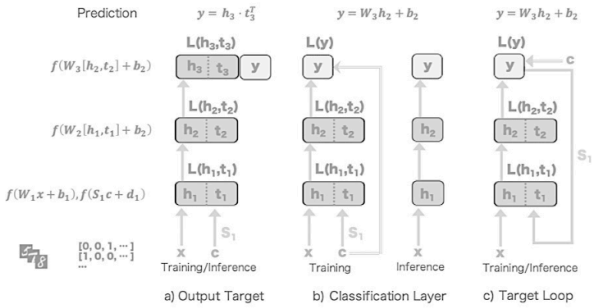
Example: NEAT: NeuroEvolution through Augmenting Topologies with genetic algorithms: in this case both weight and topology evolves trough time. Start from minimal solution to incrementally creates more complex architecture

(3) Forward Forward



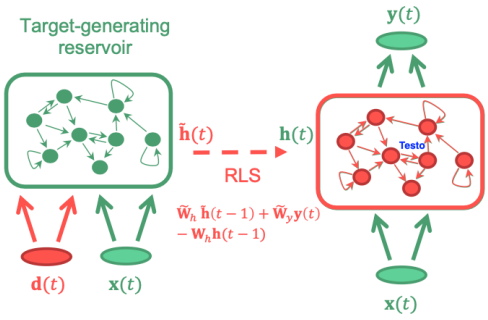
It that **avoids backward weight updates by using only forward passes**. The idea is to feed the network with input–target pairs and train each layer locally using a "goodness" measure $\sum_j h_j^2$. A **positive forward pass** uses real data (correct x–y pair) and increases the goodness, reinforcing useful activations. A **negative forward pass** uses mismatched or incorrect labels and decreases the goodness, helping the network learn to distinguish true associations. This process encourages early layers to represent input x and deeper layers to reflect the target y (as in deep NN), allowing the network to develop meaningful representations layer by layer.

(4) SigProp



This approach is conceptually similar to the Forward-Forward algorithm. In each layer, two representations are created: h_i, computed from the input alone, and t_i, computed using both the input and the target. During training, the **network updates h_i to approximate t_i, using the target-informed output as a reference to shape the target-free one**. This is done without backpropagation, often with simple **local updates** like SGD. At inference time, only the h pathway is used. If training succeeds, the system behaves as if it had access to the target during inference, because h has learned to mimic t.

(5) Full FORCEE



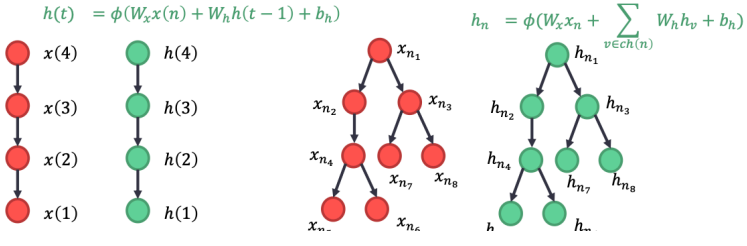
Full-FORCE learning uses a teacher-student setup with two recurrent networks. The teacher network has access to both the input x(t) and the desired output d(t) at each time step, while the student network only receives x(t). The goal is to **train the student by forcing its hidden state to match that of the teacher,** typically using methods like recursive least squares. This guides the student to evolve in a way that makes it easier to predict d(t) from its internal state. **Training happens in a single pass, without iterative backpropagation**.

The update is **asynchronous**: you don't need to wait all the hidden states computation, as soon you compute $\tilde{h}_1$ you can compute h_1 and so on.

The MASTER can has random and untrained weights and thanks to architectural bias and some constraints on weights (like ESN) the system is stable.

The **main idea behind this approaches** is that if i want to eliminate the backpropagation I need to eliminate the need of propagating back information about the targets

TREES AND GRAPHS: Recursive Neural Networks (RecNNs)

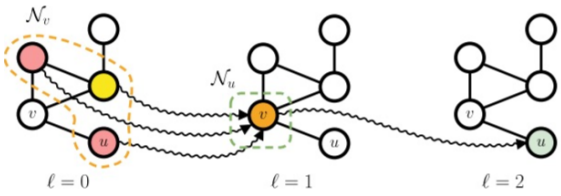


This models are a generalization of RNNs for processing hierarchical structures. The input is no longer a sequence, but a tree structure (even a graph). Time-step are now replaced by **current node** and instead of considering previous time-step as contextual information, **children** are considered. The approach is a **bottom-up recursive encoding**: a recurrent layer is unfolded on the tree from children to root like from past to

A sequence (left) is a particular case of a tree (right).

future in sequences. Once you reach the root you can use the representation created to classify or regress some value concerning the whole tree.

In case of a graph the time-step is replaced by **current vertex** and contextual information are no more previous time-step but the **neighborhood** of the node.
Also in this case, starting from the network, applying a DGN, using the last hidden representation of the node, it's possible to regress or classify a value for the whole graph or for each node or for each edge…



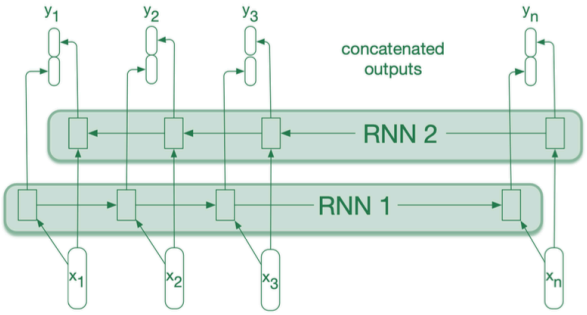
(Part 3 Lecture 3) Bi-directional and Deep RNN, gated RNN, LSTM, attention mechanism

BIDIRECTIONAL RNN

Training of vanilla RNN is biased towards **capturing only short-term relations** because of gradient vanishing. **Learning long-term dependencies is notoriously difficult in RNNs**.

Vanilla RNNs rely on **causal assumption**: I compute the information at timestep t with inputs up to time step t. Sometimes it makes sense to **"relax" causality** for example **scanning the sequence in both directions:**

- **Forward RNN (left-to-right)**: the hidden state at a given time t is a function of the inputs $x_1, x_2, \dots x_t$. $h_t^f = RNN_{forward}(x_1, \dots, x_t)$
- **Backward RNN (right-to-left)**: the hidden state at a given time t is a function of the inputs $x_n, x_{n-1}, \dots, x_t$ (i.e., the RNN is trained on the reversed input sequence). $h_t^b = RNN_{backward}(x_n, \dots, x_t)$

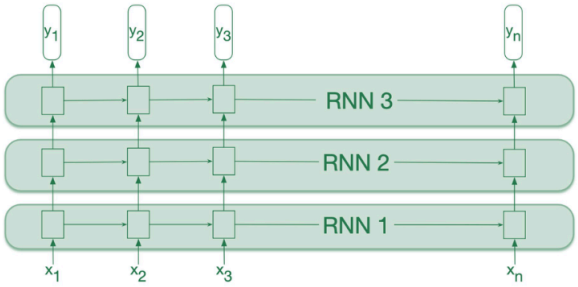


Finally the two outputs are **concatenated**: $h_t = [h_t^f; h_t^b]$

Property: each component has the same architectural bias, but overall you have more information

STACKED RNNs (Deep RNN)

The idea is to stack more RNN one upon the other. The output from one RNN is used as (driving) input to the following RNN in the stack. **This enables the representations at different levels of (temporal) abstraction of the input across layers**. Optimal number of layer is an hyperparameter so it's task dependent



How to construct deep RNNs:
There're many ways of composing RNNs. Stacked RNNs are considered as state of the art.

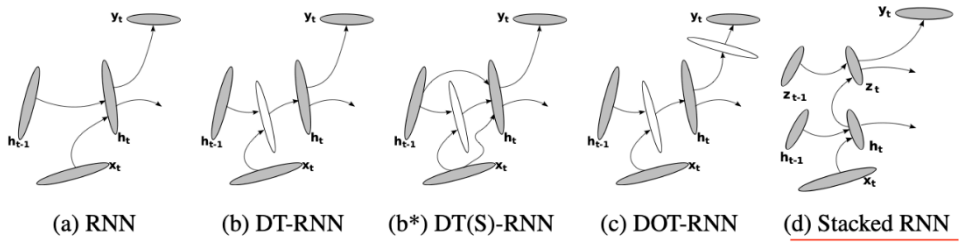


Figure 2: Illustrations of four different recurrent neural networks (RNN). (a) A conventional RNN. (b) Deep Transition (DT) RNN. (b*) DT-RNN with shortcut connections (c) Deep Transition, Deep Output (DOT) RNN. (d) Stacked RNN

The intrinsic role of depth in RNNs

Architectural bias of Deep RNNs in a reservoir computing prospective. In stacking many layers **we can observe different temporal scales**. Higher layers retains informations longer wrt to lower layers (higher memory). Intuition is that **if you organize well the architecture you can retain more memory with a deep RNN wrt to a very long and non-deep network**

GATED RECURRENT ARCHITECTURES

Gated recurrent NN takes **neurobiological inspiration**. The human brain is capable of **retaining important information** while discarding what is irrelevant, a process regulated by specific neurotransmitters and neural circuits, such as those in the hippocampus. This **selective memory** mechanism is supported by the ability of biological neurons to activate or inhibit themselves depending on context, allowing precise control over which signals are propagated and which are suppressed. Additionally, the brain exhibits adaptive temporal dynamics, adjusting its responses based on the timing and sequence of events, making its processing highly context-aware.

Atkinson-Shiffrin Memory Model is the basis for understanding how selective retention and forgetting work in the brain. Conceptualizes human memory as a system composed of 3 stores:

1. **Sensory Memory**: very short duration (milliseconds to a few seconds).
2. **Short-Term Memory (STM)**: short duration (~20–30 seconds), and information can be maintained through rehearsal.
3. **Long-Term Memory (LTM)**: stores information over extended periods, potentially indefinitely, with virtually unlimited capacity.

Recalling on Vanilla RNNs limitations:

- **Vanishing gradient:** gradients diminish as they are backpropagated over time making learning long-range dependencies extremely difficult.
- **Exploding gradient:** gradients grow exponentially leading to large weight updates (instability). Can be solved just **clipping gradient value**.

For **vanishing issue** RNN struggles with tasks requiring context from distant past: an example is language translation; this because language modeling failing to capture long-distance dependencies in text as in following sentence

```
The scientist explained the complex theory, but the students found
it difficult to grasp because the explanation was too abstract.
```

Initial context is not effectively carried forward. Predictions become less accurate as distance from the initial context increases.

Solution: use a more advanced architecture to learn long term dependencies

(1) Leaky Units

First compute a candidate hidden state $s(t)$combining previous hidden state $h(t-1)$ and current input $x(t)$. Then, using a leaky parameter α, combines previous hidden state and current candidate deciding how much leaking from previous time step:

$$\mathbf{s}(t) = tanh(\mathbf{W}_x\mathbf{x}(t) + \mathbf{W}_h\mathbf{h}(t-1))$$
$$\mathbf{h}(t) = \alpha\mathbf{h}(t-1) + (1-\alpha)\mathbf{s}(t)$$

(2) Long Short-Term Memory (LSTM)

Key idea: the network can learn what to store and what to throw away from the state

There're **two types of state information**:

- $h(t)$ **short-term state** → contains information on the current time-step and it's used for making predictions / further processing
- $c(t)$ **long-term state** → it's an internal memory of the LSTM unit that carries information across time-steps allowing **information retention over long sequences**

Gates:

Gates are standard NN layer with **sigmoid activation + pointwise multiplication.** Anything that passes through the gate is forced to be between 0 and 1 to modulate how much of the input (the input to the gate) should pass trough:

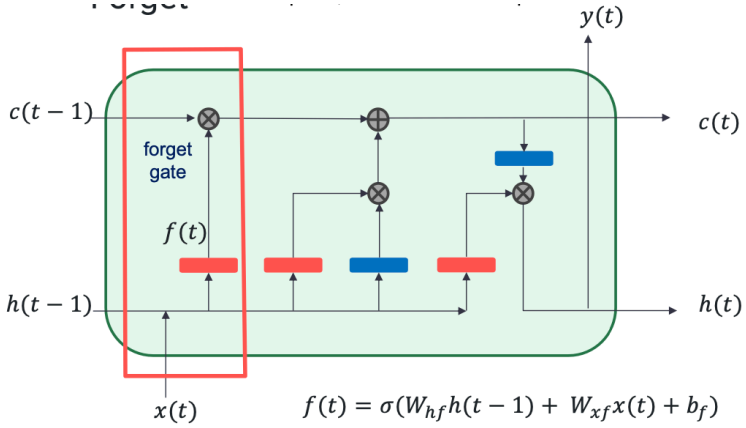
- 0 means not propagated
- 1 means propagated
- A value between 0 and 1 means shrink the information.

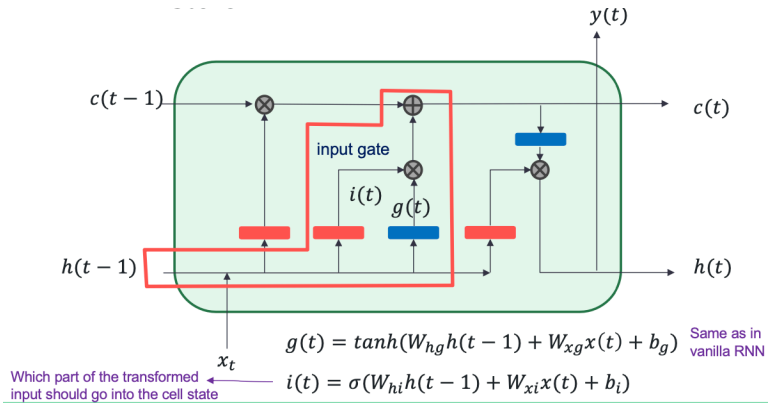
LSTM rely on **three types of gates to control the information flow**:

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

Crucial idea of LSTM: thanks of gates, LSTM can

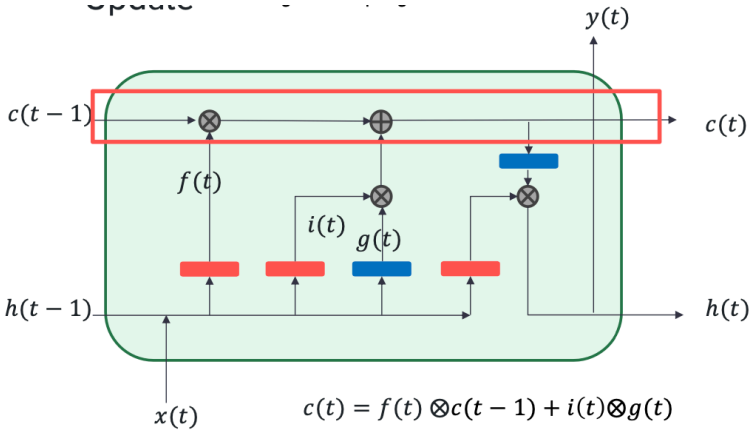
- **Forget** relevant parts of the previous long-term state. The **forget gate** decides how much of the old state should pass in or kept out, based on the new input



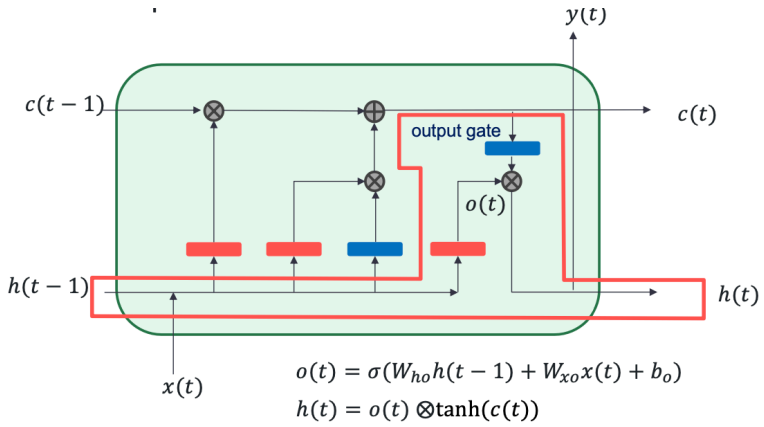


- **Store** relevant new information into the long-term state. The **input gate** modulates the inclusion of new input information into the cell state

- **Update** selectively the long-term state. The cell state is updated based on the operations of the forget and input gates. The idea is to forget irrelevant information in previous long term state (The cell $c(t-1)$) and then integrate relevant information from the input (these are mediated by input and forget gate).



- **output** the relevant part of the long-term state. The output gate decides how much from the cell state show externally. If needed, $h(t)$ is also used to compute the output at this time (in case of seq-to-seq task for example)



Gradient computation flow without interruptions from last layer to first layers. **Training is slower** because the computational graph is more complex.

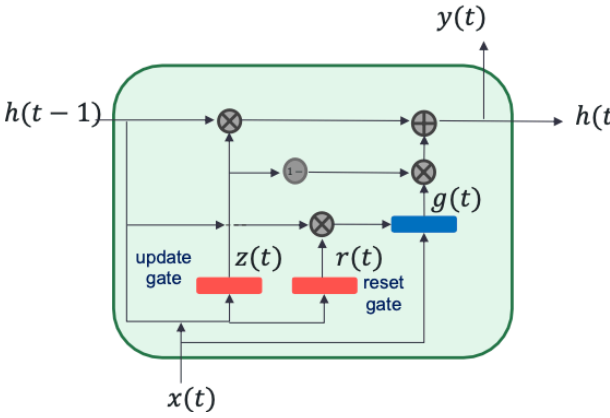
LSTM equation recap:

- $f(t) = \sigma(W_{hf}h(t-1) + W_{xf}x(t) + b_f)$
- $g(t) = tanh(W_{hg}h(t-1) + W_{xg}x(t) + b_g)$ vanilla RNN
- $i(t) = \sigma(W_{hi}h(t-1) + W_{xi}x(t) + b_i)$
- $c(t) = f(t) \otimes c(t-1) + i(t) \otimes g(t)$ *long-term state*
- $o(t) = \sigma(W_{ho}h(t-1) + W_{xo}x(t) + b_o)$
- $h(t) = o(t) \otimes \tanh(c(t))$

short-term state

training is slower

(3) Gated Recurrent Unit (GRU)



Gated Recurrent Unit is a simpler and cheaper alternative to LSTM cell. They are composed by a **one state** $h(t)$ and **two gates:**

- $r(t)$ **reset gate**: modulate how much of the previous state $h(t-1)$ participates in the update with the new input
- $z(t)$ **update gate**: modulate how much of the previous state contributes to the new state

How much to take from the previous state in the new state update

$$z(t) = \sigma(W_{hz}h(t-1) + W_{xz}x(t) + b_z)$$
$$r(t) = \sigma(W_{hr}h(t-1) + W_{xr}x(t) + b_r)$$
$$g(t) = \tanh(W_{hg}(r(t) \otimes h(t-1)) + W_{xg}x(t) + b_g)$$
$$h(t) = z(t) \otimes h(t-1) + (1 - z(t)) \otimes g(t)$$

how much to update the state

vanilla RNN

$z(t)$ update gate tells how much consider from previous memory and how much from current state $g(t)$ ($z(t) = 1$ means not modify h, $z(t) = 0$ means change everything)

ATTENTION MECHANISM

Attention mechanism takes neurobiological inspiration. Attention in the brain **allows selective focus on relevant stimuli** while ignoring others, guided by areas like the prefrontal and parietal cortex. It enables the binding of different features (e.g., color, shape, location) into a coherent perceptions and resolves competing sensory inputs through biased competition, where higher brain areas enhance processing of the most important information.

Two main works on attention:

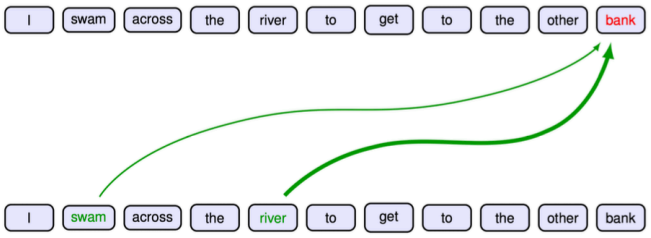
- Attention was developed initially as an enhancement to RNNs for machine translation (encoder-decoder architectures)
- Then **self attention** was introduced: focus exclusively on the attention mechanism getting rid of the recurrent part. Attention can be all you need in NLP tasks

Motivating example:

Considering the two sentences we want to translate the word in red:

- I swam across the river to get to the other bank
- I walked across the road to get cash from the bank

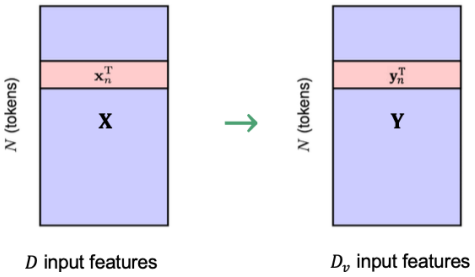
In there cases bank assumes two different meanings basing on the contextual information. In the first case probably swam and river are the most relevant information to understand the meaning of the word while in the secondo case cash and road could be.



Nota: a given input feature have different weight depending on the sample.

Attention processing

Input data is a **set of vectors** $\{\mathbf{x}_n\}_{n=1}^{N} \in \mathbb{R}^D$, or tokens (not necessarily a sequence). The main idea is to create a representation Y of this set of vectors X in such a way that every Y is a representation of every vectors in X



This is obtained computing the a_{nm} called **attention weights:**

- a_{nm} is close to zero for input tokens $\mathbf{x}_m$that have a little influence on $\mathbf{y}_n$
- a_{nm} is larger (and positive) for input tokens $\mathbf{x}_m$that have a greater influence on $\mathbf{y}_n$

So that $\mathbf{y}_n = \sum_{m=1}^{N} a_{nm}\mathbf{x}_m$ with $a_{nm} \geq 0$ and $\sum_{m=1}^{N} a_{nm} = 1$

Property: we have a different set of attention coefficients for each output vector $\mathbf{y}_n$

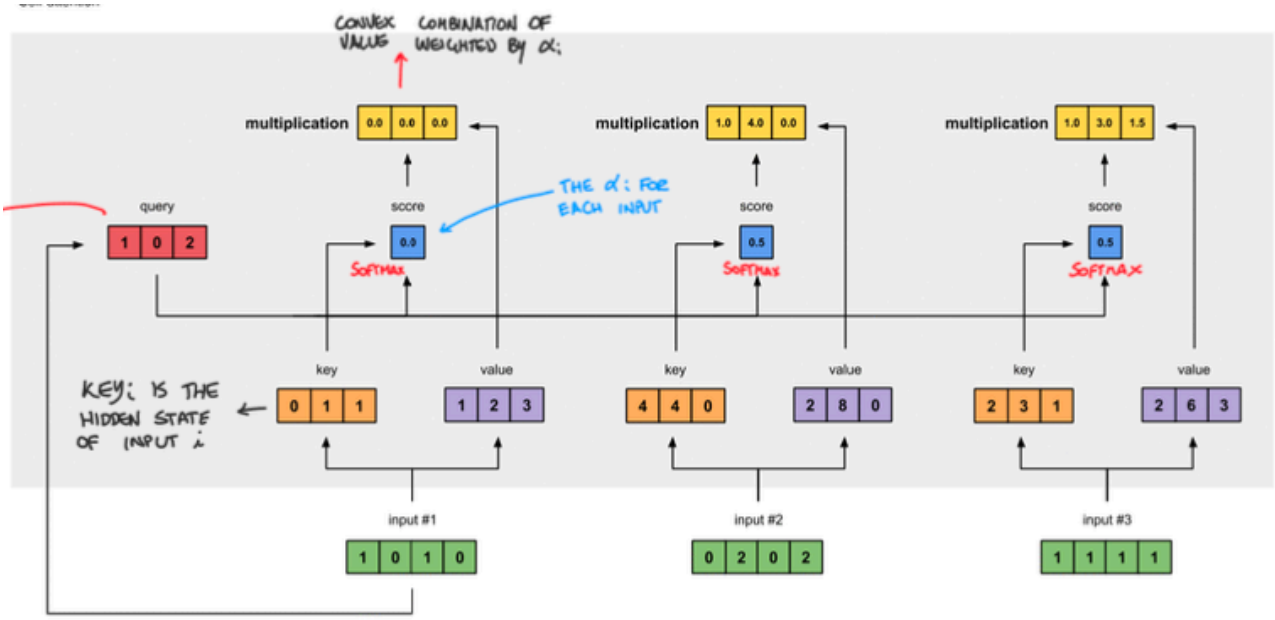
Self-attention module: Query, Keys, Values

Attention requires a different mechanism than standard neural network layers: in this case each input vector (x_n) plays three different roles and based on the role it plays it's multiplied trough some trainable weight matrices.

Why self: what this module compute is the attention of a word in a sentence with respect to itself and the other words in the same sentence

It works using each input in three different ways:
- **Query**: it is the **current focus of attention** when being compared to all the other words embeddings (it's the context vector)
- **Key**: as an element being compared to the current focus of attention
- **Value**: used to compute the output for the current focus of attention

Hard vs Soft attention: the **hard attention** selects the single value whose key is most similar to the query, assigning it a weight of 1 while all others receive a weight of 0. In contrast, **soft attention** assigns a continuous weight to each value, based on how similar its key is to the query, allowing all values to contribute proportionally (**like a probability distribution**).



The main idea is to focus on a single token $\mathbf{x}_n$ and iterate over all tokens in the sequence computing attention weights wrt all other tokens. This is done for each $\mathbf{x}_n$ in our input sentence.

The man idea is: how much $\mathbf{x}_n$, should attend to a token $\mathbf{x}_m$?

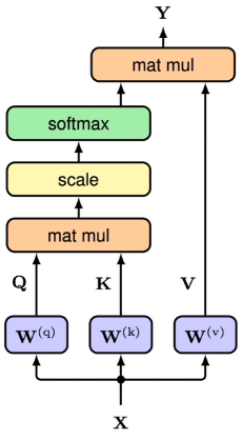
$$a_{nm} = \frac{\exp(x_n^T x_m)}{\sum_{m'=1}^{N} \exp(x_n^T x_{m'})}$$

As **similarity measure** we can use the **dot product.** **Softmax** used to **normalize** the attention weights

To properly learn the attention coefficients in a task dependent way, we can **introduce specific trainable parameters** (W) for query, keys and values.

$$Q = XW^{(q)},\ K = XW^{(k)},\ V = XW^{(v)}$$

Then $\mathbf{Y} = \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \equiv \text{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{D_{\mathrm{k}}}}\right]\mathbf{V}.$



Multi-head attention

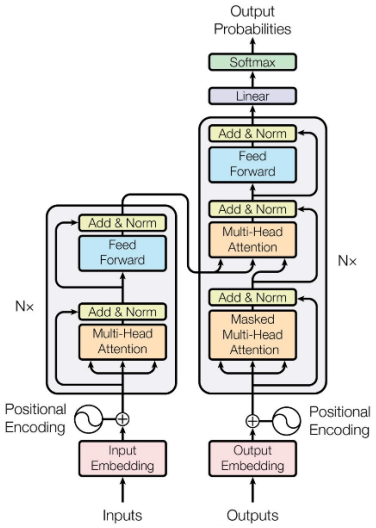
Instead of using one single attention module, the idea is to consider multiple-head (ie multiple modules), each one focusing on different aspects of the sentence. Then Y is computed as a concatenation of the representation created by the multiple heads of attention.

$$\begin{aligned}\mathbf{Q}_h &= \mathbf{X}\mathbf{W}_h^{(\mathrm{q})} \\ \mathbf{K}_h &= \mathbf{X}\mathbf{W}_h^{(\mathrm{k})} \\ \mathbf{V}_h &= \mathbf{X}\mathbf{W}_h^{(\mathrm{v})}.\end{aligned} \qquad \mathbf{H}_h = \mathrm{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)$$

$$\mathbf{Y}(\mathbf{X}) = \mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right] \mathbf{W}^{(\mathrm{o})}.$$

TRANSFORMER

Sequence transduction model based solely on attention (and other feed-forward) networks. It's an encoder-decoder architecture based on multi-head attention, normalization modules, skipping connections, positional encoding modules (that encodes position in the sequence as the input is passed without any ordering information) and masked attention for the decoder (as it not must rely on future observation in computing the attention).



Encoder is used for non causal task as it's unuseful for example for text generation so LLM like GPTs are **decoder only models**

Positional encoding: using attention (that is a FFNN module) position information are completely lost. This module allows to inform the model about the order of the elements in the sequence. Positional embedding are summed up to word embedding (also word are transformed into vector of numbers before processing them trough the transformer)

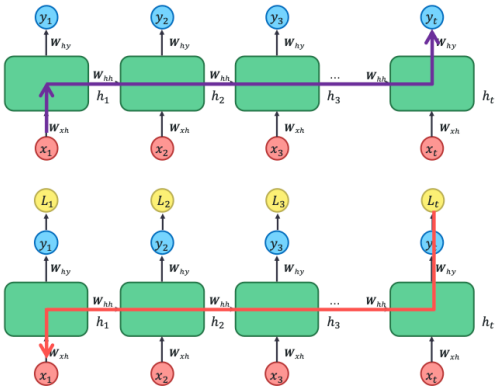
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

(Part 3 Lecture 4) Alternatives To backpropagation

Nota: given the similarities with backprop lesson, this part is integrated **here**

(Part 3 Lecture 5) Reservoir Computing

Reservoir computing strategies use a fixed dynamic system (the reservoir) to transform inputs, training only the output layer for efficient learning. An example of reservoir computing are liquid state machines, echo state networks…



Motivation: training recurrent neural networks (RNNs) is often **computationally demanding** — in terms of time, energy, and cost — and can suffer from issues such as **vanishing or exploding memory**. These models are prone to problems in both the forward and backward passes: during the forward phase, the **influence of past inputs may either decay or grow uncontrollably** as they pass through multiple nonlinear transformations; during training (the backward phase), **gradients can vanish or explode**, hindering the network's ability to learn long-term dependencies

Note: fading/exploding memory is a problem of forward computations while gradient vanish/exploding of backward propagation

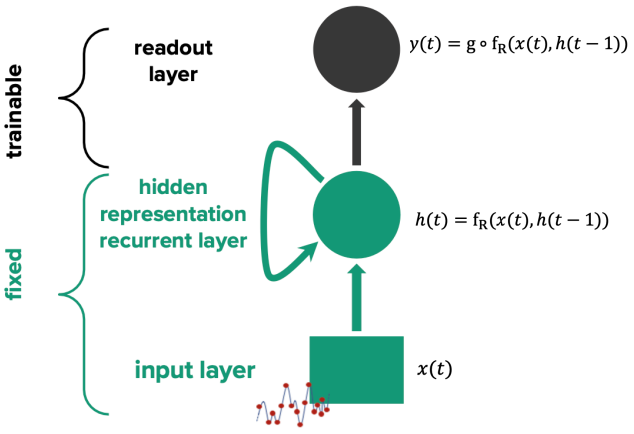
Why this happens: due to the model's architecture that is the same structure is replicated across time steps.

How to tackle propagation issues:
- clipping for exploding gradient
- gated RNN (e.g. LSTM or GRU) to learn long-term dependencies.

The latter comes at a **very high cost** in terms of **time**, **parameters**, **complex training strategies** and **complex parameter coupling**

!! Training Recurrent Neural Networks is slow, energy consuming and difficult !!

Idea: we will try to exploit architectural biases to simplify a bit, exploiting the fact that **randomization is computational cheaper then parameter optimization.**



Randomized Recurrent Neural Networks

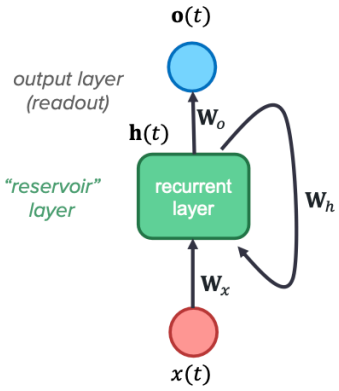
A RNN where the hidden layer is random and fixed, while the readout layer is trainable

Randomization means efficiency, this architecture need a cheaper and simpler training algorithm (only for the readout layer).

This implementation is **flexible** also **for neuromorphic implementations**

Note: this idea historically comes from the **cortico-striatal model** because it suggests that the cortex generates rich, dynamic activity patterns (using highly connected, recurrent connection), while learning occurs mainly in the downstream structures like the striatum.

ECHO STATE NETWORK



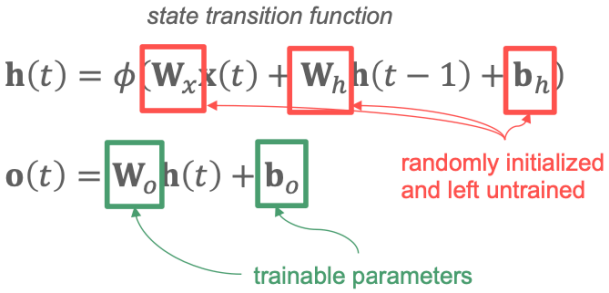
ESN are **an example of reservoir computing architecture**. Starting from a vanilla RNN with a single recurrent (hidden) layer and an output layer (the readout) the idea is to randomly initialize both input to hidden weights W_x and hidden-to-hidden weights W_h, and to left them untrained. Then the readout layer's weights W_o are trained.

Architecture of an ESN in a nutshell:

- **Reservoir**: untrained non-linear recurrent hidden layer
- **Readout**: (linear) output layer

The reservoir layer:

The **reservoir** $h(t)$ is a large layer of recurrent units sparsely connected and **randomly initialized under stability conditions of the dynamical system** that ensures the echo state property being satisfied. This parameters are **keep untrained**



The **reservoir** aims to **non-linearly embed the input to an higher dimensional feature space** where the original problem is more likely to be solved linearly (Cover's Th.). It's randomized basis expansion computed by a pool of randomized filters

The readout layer:

The **readout layer** $o(t)$ use the features in the reservoir state space for the output computation. It's typically implemented by using linear models so that learning involves a simple **convex optimization** via a closed form $\mathbf{W_o} = \mathbf{Y}\mathbf{H}^{\mathbf{T}}(\mathbf{H}\mathbf{H}^{\mathbf{T}} + \lambda\mathbf{I})^{-1}$

Network setup:

1. Initialize the parameters of the state transition function randomly
2. Scale the recurrent weight matrix to meet the contractive/stability property

Training:

1. Drive the network with the input signal
2. Discard an initial transient: can be avoided for time-series classification tasks
3. Train the readout with a direct approach, starting from the hidden values

Approximation capabilities

Echo State Networks are particularly powerful because they **can approximate any fading memory filter**. This means they are capable of modeling systems in which the influence of past inputs gradually decreases over time, a common characteristic in many real-world time series (current output value depend only on more recent inputs, a sort of **short-term memory network**). Despite training only the output layer and keeping the reservoir fixed, ESNs retain strong approximation capabilities. This theoretical property highlights their expressive power and justifies their effectiveness in processing temporal data with decaying memory effects.

Advantages

- **Clean mathematical analysis:** thanks to fixed reservoir dynamics, analysis is simpler and avoids biases of fully trainable RNNs. It works because of the architectural bias of contracting RNNs (see Markovian bias down)
- **Efficiency:** no need for backpropagation through time → faster and cheaper training. More then 1500x faster and efficient wrt full trained models
- **Neuromorphic hardware compatibility:** the reservoir can be implemented using physical systems with required dynamics and non-linearity (e.g., optical, electronic, mechanical).

Reservoir Computing: mathematical description

The reservoir is a **discrete-time input-driven dynamical system.** Dynamics are driven by the state transition function $F : \mathbb{R}^{N_x} \times \mathbb{R}^{N_h} \to \mathbb{R}^{N_h}$ that takes in input both the input signal $x(t)$ and the hidden state at time t - 1 and output the hidden state at time t.

$$\mathbf{h}(t) = F(\mathbf{x}(t), \mathbf{h}(t-1)) = \tanh(\mathbf{W}_x \mathbf{x}(t) + \mathbf{W}_h \mathbf{h}(t-1))$$

The **iterated version** of the state transition function $\hat{F} : (\mathbb{R}^{N_x})^* \times \mathbb{R}^{N_h} \to \mathbb{R}^{N_h}$ is recursively defined as:

$$\hat{F}(\boldsymbol{s}, \mathbf{h_0}) = \begin{cases} \boldsymbol{h_0} & if\ \boldsymbol{s} = [\,] \\ F(\boldsymbol{x}(t), \underbrace{\hat{F}([\boldsymbol{x}(1), \ldots, \boldsymbol{x}(t-1)], \boldsymbol{h_0})}_{\text{the state after t-1 inputs}}) & if\ \boldsymbol{s} = [\boldsymbol{x}(1), \ldots, \boldsymbol{x}(t)] \end{cases}$$

Note: this definition will be useful in proving the ESP

Example: for an input sequence s_1, s_2

$$\hat{F}([\,], \mathbf{h}_0) = \mathbf{h}_0$$
$$\hat{F}([s_1], \mathbf{h}_0) = F(s_1, \hat{F}([\,], \mathbf{h}_0)) = F(s_1, \mathbf{h}_0)$$
$$\hat{F}([s_1, s_2], \mathbf{h}_0) = F(s_2, \hat{F}([s_1], \mathbf{h}_0)) = F(s_2, F(s_1, \mathbf{h}_0)) = F(s_2, \mathbf{h}_1)$$

ECHO STATE PROPERTY

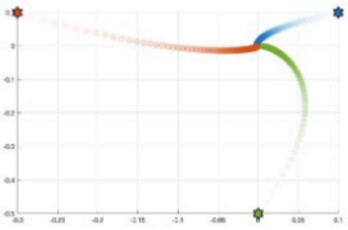
Echo state Property: A valid ESN should satisfy the "Echo State Property". The idea is to use a **random but stable** network, meaning that, regardless the initial condition, the system will eventually settle into a stable state after a transient phase

(**Def**) an ESN satisfy the ESP whenever: $\forall \mathbf{s} \in (\mathbb{R}^x)^N, \forall \mathbf{h}_0, \mathbf{z}_0 \in \mathbb{R}^{N_h}$:

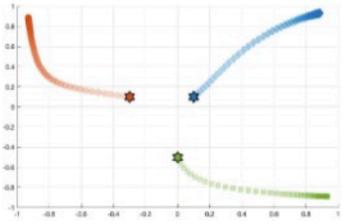
$$\|\hat{F}(\mathbf{s}, \mathbf{h}_0) - \hat{F}(\mathbf{s}, \mathbf{z}_0)\| \to 0, \quad as\ N \to \infty$$

Hint: the distance between the final states goes to 0 with the length of the input whatever is the initial condition ($\mathbf{h_0}$, $\mathbf{z_0}$).

The main idea is that the **state of the network asymptotically depends only on the driving input signal** as dependencies on the initial conditions are progressively lost



ESN with ESP



ESN without

Condition for the ESP

The ESP can be inspected by controlling the algebraic properties of the recurrent weight matrix $\mathbf{W}_h$ (thus it **controls the asymptotic stability property of the reservoir system**). Two major results are given:

- **Sufficient Condition**, involving the control of the maximum singular value of $\mathbf{W}_h$
- **Necessary Condition**, involving the control of the maximum eigenvalue in modulus of $\mathbf{W}_h$

What's left? To link the ESP to the actual input of your network

Sufficient = guarantee of ESP; Necessary = requirement for ESP (if missing, no ESP)

Sufficient condition for the ESP:

Theorem. If the *maximum singular value* of $\mathbf{W}_h$ is *smaller than* 1 then (under mild assumptions) the ESN satisfies the ESP for any possible input. This means **contractive dynamics for every input**:

$$\sigma_{max}(\mathbf{W}) = \|\mathbf{W}\|_2 < 1$$

Necessary Condition for the ESP:

Theorem. If the *spectral radius* of $\mathbf{W}_h$ is *larger than* 1 then (under mild assumptions) the ESN *does not satisfy* the ESP. This means **globally asymptotically stable dynamics around 0**:

$$\rho(\mathbf{W}) = \max(abs(eig(\mathbf{W}))) < 1$$

Relationship between the ESP conditions: from linear algebra we know that $\rho(\mathbf{W}) \leq \|\mathbf{W}\|_n$

Practical usage:

- Applying the sufficient condition is OK in theory, but **often impractical**: it is too strong!
- Usually, the **necessary condition is used** as an easy way for initialization of the reservoir

More About ESP (integrated from the end of the lecture)

An **unstable network exhibits sensitivity to input perturbations**: two slightly different (long) input sequences drive the network into (asymptotically very) different states. This is **good for training** as the state vectors tend to be more and more linearly separable, but it's **bad for generalization** (overfitting) if a temporal sequence similar to one in the training set drives the network into completely different states.

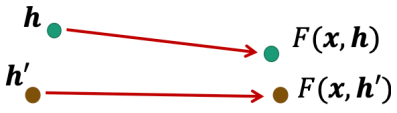
Contractions and the ESP

The **sufficient condition** considers scenarios where the state transition function exhibits contractive dynamics: *whatever is the driving input signal if the system is contractive then it will exhibit stability.*

Considering $\mathbf{h}(t) = F(\mathbf{x}(t), \mathbf{h}(t-1)) = \tanh(\mathbf{W}_x\mathbf{x}(t) + \mathbf{W}_h\mathbf{h}(t-1))$

Contractive Reservoirs: the reservoir has **contractive dynamics** whenever its state transition function F is **Lipschitz continuous** with constant $L < 1$ i.e. whenever:

$$\exists L \in \mathbb{R},\ 0 \leq L < 1,\ \forall \mathbf{x} \in \mathbb{R}^{N_x},\ \forall \mathbf{h}, \mathbf{h}' \in \mathbb{R}^{N_h}:$$
$$\|F(\mathbf{x}, \mathbf{h}) - F(\mathbf{x}, \mathbf{h}')\| \leq L\|\mathbf{h} - \mathbf{h}'\|$$



Theorem: 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:

Assuming F is contractive with coefficient $L < 1,\ \forall \mathbf{s} = [\mathbf{x}(1), \ldots, \mathbf{x}(N)], \forall \mathbf{h}_0, \mathbf{z}_0$:

$$\|\hat{F}(\mathbf{s}, \mathbf{h}_0) - \hat{F}(\mathbf{s}, \mathbf{z}_0)\| =$$
$$\|\hat{F}([\mathbf{x}(1), \ldots, \mathbf{x}(N)], \mathbf{h}_0) - \hat{F}([\mathbf{x}(1), \ldots, \mathbf{x}(N)], \mathbf{z}_0)\| =$$
$$\|F(\mathbf{x}(N), \hat{F}([\mathbf{x}(1), \ldots, \mathbf{x}(N-1)], \mathbf{h}_0)) - F(\mathbf{x}(N), \hat{F}([\mathbf{x}(1), \ldots, \mathbf{x}(N-1)], \mathbf{z}_0))\| \leq$$

$$L\|\hat{F}([\mathbf{x}(1),\ldots,\mathbf{x}(N-1)],\mathbf{h}_0) - \hat{F}([\mathbf{x}(1),\ldots,\mathbf{x}(N-1)],\mathbf{z}_0)\| \leq$$
$$\ldots$$
$$\leq L^N\|\mathbf{h}_0 - \mathbf{z}_0\| \to 0 \quad as\ N \to \infty$$

Theorem. Assume an ESN with *tanh* reservoir neurons and consider a reservoir space equipped with Euclidean distance (norm). If $\|\mathbf{W}\|_2 < 1$ then the ESP holds for any input.

Proof: $\forall \mathbf{x}, \mathbf{h}, \mathbf{h}'$:

$$\|F(\mathbf{x},\mathbf{h}) - F(\mathbf{x},\mathbf{h}')\|_2 = \|\tanh(\mathbf{U}\mathbf{x} + \mathbf{W}\mathbf{h}) - \tanh(\mathbf{U}\mathbf{x} + \mathbf{W}\mathbf{h}')\|_2$$
$$\leq \max(tanh')\|\mathbf{U}\mathbf{x} + \mathbf{W}\mathbf{h} - \mathbf{U}\mathbf{x} - \mathbf{W}\mathbf{h}'\|_2 \ = \ \|\mathbf{W}(\mathbf{h} - \mathbf{h}')\|_2 \leq \|\mathbf{W}\|_2\|\mathbf{h} - \mathbf{h}'\|_2$$

NOTE: the inequality comes from he fact that tanh is Lipschitz continuous with L = max(tanh') = 1

This proves that the **reservoir state transition function is a contraction**, and the ESP is valid.

Markovianity, contractive systems and ESN:

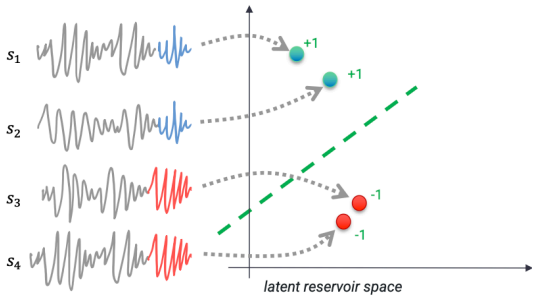
Contractive dynamical systems exhibit a **suffix-based organization of the state space**: input sequences sharing a common suffix lead to similar states, with similarity increasing with the length of the shared suffix. This gives the system a Markovian flavor: similar sequences map to nearby states, while different sequences map to distant ones.

Recalling from **architectural bias of contractive RNN**, we know that RNNs can discriminate between different input time-series even in absence of (or prior to) learning of the recurrent connections

Then in ESN the suffix-based Markovian state space organization is a fixed characterization: the reservoir constructs a high-dimensional suffix-based state space representation of the driving input time-series and then apply a linear readout layer.

Why does ESN works?

This behavior is due to the **Markovian bias** inherent in RNN architectures: they tend to map input sequences similarly when those sequences share common suffixes. This effect arises from the architectural design of RNNs and can be observed even when the network is left untrained.



The necessary condition

Theorem. If an ESN has *unstable dynamics* around the zero state and the zero sequence is an admissible input, then the ESP is *not* satisfied.

Linearize the reservoir around an initial state $\mathbf{h}_0$:

$$\mathbf{h}(t) = F(\mathbf{x}(t),\mathbf{h}(t-1)) \qquad \textit{(that we approximate with a jacobian function)}$$
$$\approx J_F(\mathbf{x}(t),\mathbf{h}_0)(\mathbf{h}(t-1) - \mathbf{h}_0) + F(\mathbf{x}(t),\mathbf{h}_0)$$

With: $J_F = \begin{bmatrix} \frac{\partial F_1}{\partial h_1} & \cdots & \frac{\partial F_1}{\partial h_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_N}{\partial h_1} & \cdots & \frac{\partial F_N}{\partial h_N} \end{bmatrix}$

Now suppose a constant 0 input is admissible for the system, and linearize around the zero state:

$$\mathbf{h}(t) \approx J_F(0,0)\mathbf{h}(t-1)$$

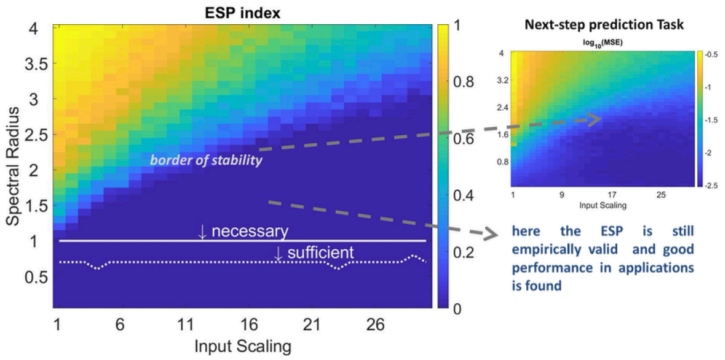
0 is a fixed point of the above system, whose stability depends on the eigenvalues of $J_F(0,0)$. In this very peculiar case $J_F(0,0) = \mathbf{W}_h$. If $\rho(\mathbf{W}) \geq 1$ then 0 is not a stable fixed point: there exists another initial state for which the trajectory will never converge to 0 (a counter example for the ESP). **Hence, $\rho(\mathbf{W}_h) < 1$ is a necessary condition for the ESP** because if it's violated there exist at least a case in which the ESP is not valid

Caution: scaling $\rho(\mathbf{W}_h)$ below 1 is, in general, neither sufficient nor necessary in practical applications **because the true dynamics are driven by non-trivial, obviously non-zero input**. The true dynamics cannot be well linearized, and non-linearities introduce bifurcations s.t. the ESP does not hold

ESP index: which is the real bound of ρ?

The **ESP index** is equals to 0 when there is perfect synchronization across all possible trajectories, and increases as synchronization decreases.

As we can observe from the plot, the Echo State Property (ESP) holds both below the sufficient and necessary thresholds. However, even with a spectral radius greater than 1, the model can still perform well — as shown in the blue region.



Reservoir initialization:

Initialization of $\mathbf{W}_h$:

1. Generate a random matrix $\mathbf{W}_r$ whose elements are drawn e.g. from a uniform dist. on [-1,1]
2. Scale $\mathbf{W}_r$ by the desired spectral radius: $\mathbf{W}_h \leftarrow \mathbf{W}_r \frac{\rho_{desired}}{\rho(\mathbf{W}_r)}$, now $\rho(\mathbf{W}_r) = \rho_{desired}$

Note: spectral radius is a key hyperparameter of the reservoir, choose a value < 1

Alternatively: generate a random matrix from $U\left(-\frac{3\rho}{\sqrt{3N}}, \frac{3\rho}{\sqrt{3N}}\right)$. *As number of neurons* N *approaches infinity, this initialization ensures that the spectral radius of* $\mathbf{W}_h$ *converges to the desired value — a particularly useful property for very large ESNs. In this case generation and rescaling could be computationally expensive*

Initialization of $\mathbf{W}_x$:

1. Generate a random matrix $\mathbf{W}_x$ whose elements are drawn e.g. from a uniform dist. on [-1,1]
2. Scale by an input scaling parameter ω_{in} (many strategies allowed)
   - By **range**: $\mathbf{W}_x \leftarrow \omega_{in}\mathbf{W}_x$ (now weights are in $[-\omega_{in}, \omega_{in}]$)
   - By **norm**: $\mathbf{W}_x \leftarrow \omega_{in}\frac{\mathbf{W}_x}{\|\mathbf{W}_x\|_2}$ (now the 2-norm of $\mathbf{W}_x$ equals ω_{in})

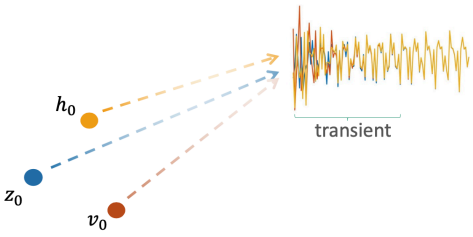
Note: input scaling ω_{in} is a key hyper-parameter of the **reservoir**

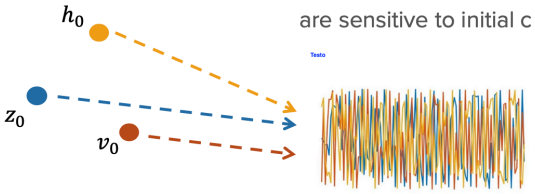
Dynamical transient

If the system is ***globally asymptotically stable***, then all possible (input-driven) trajectories will synchronize after an initial transient. The **washout** is the initial part of the time-series in which the state could be still affected by initialization condition (i.e., here the ESP might still not hold)

Important practice: the **washout states** of the reservoir **should be discarded**

Stable dynamics: orbits synchronize after the an initial transient, regardless the initial state —>





<—**Unstable dynamics:** orbits are sensitive to initial conditions

ESN Training:

1. Process the whole input serie and record the reservoir states: $H = [\mathbf{h}(1), \ldots, \mathbf{h}(N)]$
2. Washout the initial transient: $\mathbf{H} \leftarrow \mathbf{H}(:, N_w : N)$
3. Collect the target data similarly into a matrix: $Y = [\mathbf{y}(N_w), \ldots, \mathbf{y}(N)]$
4. Solve the linear regression problem for the readout: $\min_{\mathbf{W}_o} ||\mathbf{W}_o\mathbf{H} - \mathbf{Y}||_2^2$

Training the readout:

- **On-line training is <u>not</u>** the traditional choice for ESNs:
  - Least Mean Squares is typically not suitable because of high eigenvalue spread (i.e. large condition number) of $\mathbf{H}$
- **Off-line training is standard** in most applications:
  - Closed form solution of the least squares problem by direct methods:
    - pseudo-inversion method: $\mathbf{W}_o = \mathbf{Y}\mathbf{H}^\dagger = \mathbf{Y}\mathbf{H}^T(\mathbf{H}\mathbf{H}^T)^{-1}$
    - Ridge regression with regularization: $\mathbf{W}_o = \mathbf{Y}\mathbf{H}^T(\mathbf{H}\mathbf{H}^T + \lambda\mathbf{I})^{-1}$

Practicalities:

- typically *tanh activation function* is used (sometimes with leakage).
- Linear readout trained in closed form (but any regressor/classifier can be used

Note: When working on a *single time-series* you do <u>not</u> need to:

- re-initialize the state when going from training to validation sets
- discard the initial transient (because the reservoir has time to stabilise naturally during continuous sequence processing)

This two are needed if working with multiple time series

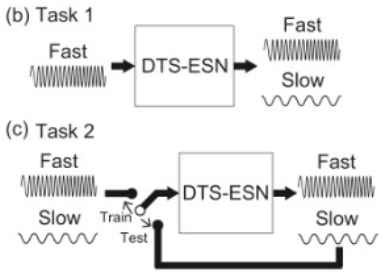
ESN hyperparameters are reservoir size N_H, spectral radius ρ, input scaling ω_{in}, readout regularization λ and others task dependent…

Practical tips:

- use a **sparse reservoir** for faster computations
- Using $\rho < 1$ gives the ESP in most situations (in experiments explore also values >1)
- Use larger **values of ρ when more memory is needed**
- If possible scale ω_b separately for the bias term

ARCHITECTURAL VARIANTS

Leaky Integrator – Echo State Network



This architecture is useful when dealing with time series data that contains patterns evolving at **different temporal scales**. In a standard ESN, the reservoir state is fully updated at each time step. However, in a Leaky ESN, the state update is more gradual.

The update rule becomes:

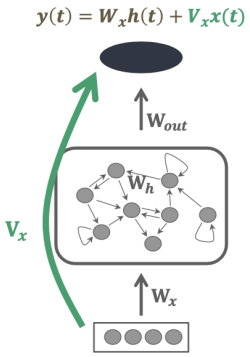
$$\mathbf{h}(t) = (1-\alpha)\mathbf{h}(t-1) + \alpha \tanh(\mathbf{W}_h\mathbf{x}(t) + \mathbf{W}_h\mathbf{h}(t-1) + \mathbf{b}),$$

with $\alpha \in [0,1]$

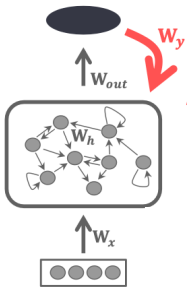
A **smaller alpha** makes the reservoir update more slowly, meaning it keeps memory of past states longer while a **larger alpha** reacts more quickly to new inputs.

Leaky units allow ESNs to handle **temporal tasks that require both short-term and long-term memory**

Input-output and output-feedback connections



In the **input-output skip connection** the input signal $x(t)$is used also to compute the output $y(t)$, trough skip connections V_x. The input-output skip connection allows the input to directly affect the output, **improving the network's short-term responsiveness** without depending entirely on the reservoir dynamics. This is especially useful when the input signal carries immediately relevant information.

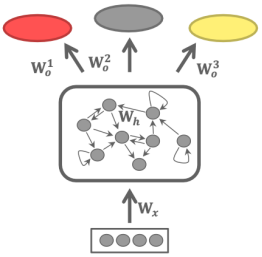


In the **output-feedback connection** the **previous output** is feed back into the reservoir, enabling the network to condition its internal state on past predictions. Using this strategy we can have a "**trainable reservoir**" because the dynamics depends on the trainable output so its like indirectly training the reservoir:

$$\boldsymbol{h(t) = \sigma(W_x x(t) + W_h h(t-1) + W_y y(t-1))}$$

$$\boldsymbol{h(t) = \sigma(W_x x(t) + (W_h + W_y W_{out}) h(t-1))}$$

trained



Multiple readouts

The reservoir is operating in a purely unsupervised mode; if multiple tasks involve the same input time-series (but different targets) the same reservoir could be used, computing different readout for different tasks.

Applications Examples: Echo State Networks (ESNs) can be applied to a wide range of tasks, including chaotic time-series modeling (useful for predicting complex and nonlinear dynamics), intelligent sensing thanks to their lightweight and energy-efficient design, general time series forecasting, and hybrid sensor-robotic systems where low-latency and adaptability are crucial. Their versatility also makes them suitable for applications in control systems, speech processing, and real-time signal prediction

ACTIVE RESEARCHES ON RESERVOIR COMPUTING

The random reservoir is not the only possible choice; we can also consider:

1. **High entropy of neurons activations** that diversify the temporal response of the reservoir neurons (not trained but not completely random)
2. Enhancing **long short-term memory capacity**
3. **Close to the edge of chaos**: reservoir at the border of stability that show optimal performances whenever the task at hand requires long short-term memory

1. Intrinsic plasticity

Intrinsic plasticity is a technique used to empower ESN performances in **unsupervised** fashion.

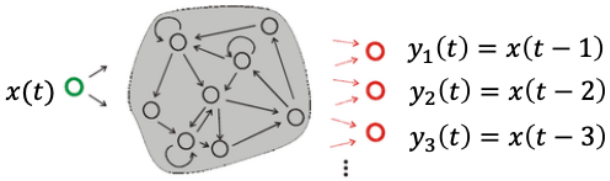
The key idea is to exploit this neurobiologically inspired mechanism that adapts two neurons' added parameters (**gain** a and **bias** b) to modify the activation function in order to maximise the entropy of the neuron's output.

Each neuron has an activation function $f(x) = f(a\,x + b)$ in which the gain a amplify the signal and b the bias shift it. The idea is to **tune this parameters minimising the Kullback-Leibler** (KL) **divergence** between the current output distribution and a target distribution (typically Gaussian with mean μ and variance σ^2, that are hyper-parameters)

This **improves the memory capacity** of the ESN helping the model in the generalization phase.

2. Short-term memory

In this case we'd like to assess the ESN abilities in remembering passed input.



$$MC = \sum_{k=1}^{\infty} r^2(x(t-k), o_k(t))$$

The **memory capacity** MC measures how many versions of the delayed input the model could learn. In the formula $x(t-k)$ is the delayed input and $o_k(t)$ is the model output for a model trained with a delay equal to k.

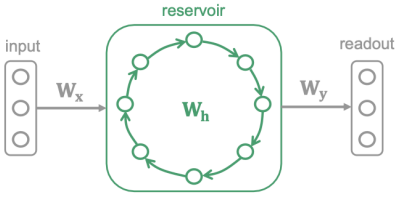
In practice we **learn different delayed next step prediction tasks on the same input sequence and we measure as far as the model is able to remember**. In real cases no more the $2 * N_H$ delays are tried as up to this number the model is no more able to reconstruct correctly the input.

Nota: for iid input signal $MC \leq N_H$. This maximum value is reached with **linear** reservoirs with orthogonal hidden-to-hidden matrix.

Trade-off memory vs representation power: Linear reservoirs can reach this max MC, but lacks of modeling complex relationship while non linear reservoirs can model any complex relations but could loose memory informations.

2. Cycle reservoirs

It's a non random ESN where hidden matrix has circular shape. The hidden matrix W_h is **orthogonal** empowering the MC. The **desired**



spectral radius $\hat{w}$ is a scaling hyperparameter used to rescale reservoir weights. This ESN is very efficient to be implemented as the multiplication with W_h corresponds to a simple shift operation (last element goes in first position and the whole array is multiplied by $\hat{w}$: $O(1)$ rather than $O(N^2)$.

Orthogonality means that its columns are **linearly independent** and its **norm is preserved** (useful for network dynamic stability)

$$\mathbf{W_h} = \begin{pmatrix} 0 & 0 & \dots & 0 & \hat{w} \\ \hat{w} & 0 & \dots & 0 & 0 \\ 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 \\ 0 & \dots & 0 & \hat{w} & 0 \end{pmatrix}$$

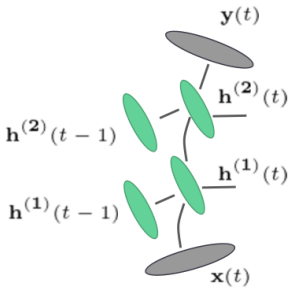
The dynamic of the system can be described as: $J(t) = D(t)\mathbf{P}$
Where:

- $D(t)$ changes at each timestep and its the "dynamic components of activation"
- $\mathbf{P}$ a fixed orthogonal matrix, representing the static structure of the reservoir

super huge compression: we can represent the reservoir just with a number ($\hat{w}$). This because applying the reservoir is just as shifting and multiply by that value!

3. Deep Reservoirs

What if we stack many hidden layers in ESN? Experimentally it's proven that if we increase the layers the MC increases but as the spectral radius exceed 1 we have a drop in the network MC.



Composition: multiple reservoirs one on top of the other, first is driven by the external input while hidden layers are driven by the dynamics of previous stacked hidden layer

$$\mathbf{h}^{(i)}(t) = \tanh(\mathbf{W}_h^{(i)}\mathbf{h}^{(i)}(t-1) + \mathbf{W}_x^{(i)}\mathbf{h}^{(i-1)}(t) + \mathbf{b}^{(i)})$$

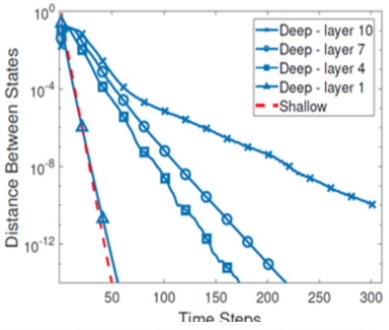
driving input

...

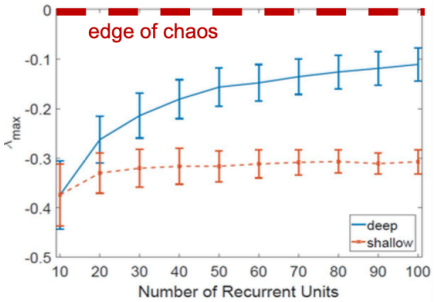
$$\mathbf{h}^{(1)}(t) = \tanh(\mathbf{W}_h^{(1)}\mathbf{h}^{(1)}(t-1) + \mathbf{W}_x^{(1)}\mathbf{x}(t) + \mathbf{b}^{(1)})$$

Architectural bias of deep RNN: multiple layers enables the RNN to act at **multiple time-scales** and **multiple frequencies** ie at different layer each RNN focuses on different frequencies of the input signal. Thanks to this representation power this models develop a **richer dynamics even without training of the recurrent connections**.

In this experiment, they considered two input sequences: one is a a slightly perturbed version of the other. The **distance between their states over time shows that as the number of layers increases**, convergence through the ESP becomes slower, increasing the risk of instability. This can be beneficial depending on the task—for example, in applications requiring long memory or noise robustness, like time series forecasting.



Lastly, increasing the number of recurrent units will bring the Lyapunov exponent λ_{max} toward 0. Theory tells us that **when this coefficient approaches 0 the state dynamics are closer to the edge of chaos**.



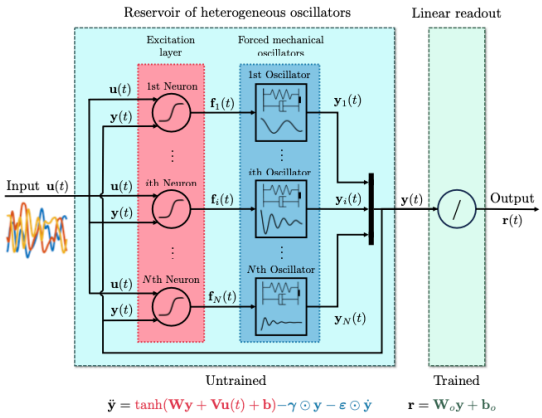
$$\lambda_{max} = \max_k \frac{1}{T} \sum_{t=1,\dots,T} \ln |\lambda_k(J(t))|$$

Furthermore, there is evidence that **near the edge of chaos**, the system exhibits **more expressive**, but also **more delicate dynamics.**

Random Oscillators Network

The Random Oscillators Network (RON) is a new architecture for reservoir computing (RC) that combines **standard tanh neurons** with **forced**, **untrained mechanical oscillators**. The (readout) output function is trained as in Reservoir Computing.

It implements a 2^{nd} order dynamical system and it reaches **state of the art performances** at a **fraction of the training costs**



Issue with ESN: they enforce that all possible trajectories converge after a transient phase and this is the essence of the **fading memory property**. However, fading memory isn't always desirable: if early time steps are crucial for later computations, ESNs may lose important information, as they struggle with long-range dependencies.

The following solutions aim to address this limitation.

Euler Reservoirs

Euler method inspired reservoirs shows a **non-dissipative stable dynamics** by design.

The dynamics of the hidden state are derived from the **discretization of a continuous-time ODE using the forward Euler method**.

$$h' = \tanh(W_x\, x + W_h h + b)$$

$$\mathbf{h}(t) = \mathbf{h}(t-1) + \underbrace{\varepsilon}_{\text{step size}} \tanh(\boxed{\mathbf{W}_x}\, \mathbf{x}(t) + (\boxed{\mathbf{W}_h} - \boxed{\mathbf{W}_h^T} - \underbrace{\gamma}_{\text{diffusion}}\mathbf{I})\mathbf{h}(t-1) + \boxed{\mathbf{b}})$$

untrained

It has two parameters:

- ϵ the **step size** of the Euler method
- γ the **diffusion parameter** that push a little more trough stability helping with memory

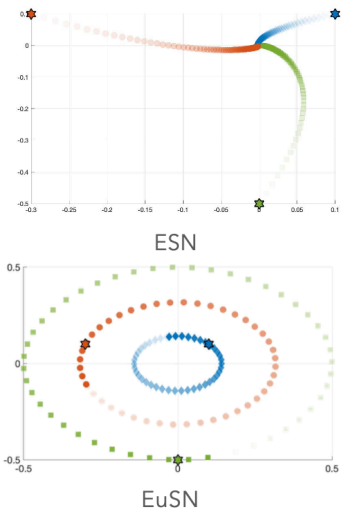
The key idea behind this ESN is to start from a continuous time model and then structure the recurrent matrix in a way that it not dissipate information **using an anti-symmetric weight matrix**

Which antisymmetric weight matrix to use? Start from W_h and subtract its transposed

An **antisymmetric matrix** ensures that all eigenvalues have a **real part equal to zero**. This is crucial because the real part of the eigenvalues determines the system's stability:

- If the **real part is negative**, the system is dissipative and quickly loses information.
- If it's **positive**, the system becomes unstable and can explode over time.
- When the **real part is zero**, the system is marginally stable and non-dissipative, preserving the dynamics without amplification or decay.

Using antisymmetric weights helps maintain a balance—avoiding both vanishing and exploding trajectories—thereby supporting stable long-term memory propagation.



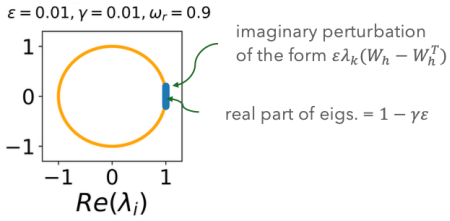
Thanks to this structure, dynamics are **arbitrarily close to the edge of stability**, a regime known to maximize a network's memory capacity and computational richness.

This models ensure stability by **preventing the hidden state dynamics from exploding or vanishing**; signals maintain consistent amplitude over time, as seen in the plots. Additionally, the hidden states follow **closed**, **circular trajectories**, which helps **preserve input information over time**.

The **Jacobian** of bias-free autonomous system is:

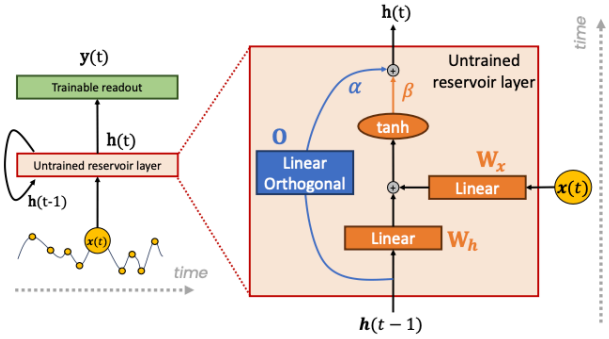
$$\mathbf{J}_{EuSN} = \mathbf{I}(1 - \gamma\epsilon) + \epsilon(\mathbf{W}_h - \mathbf{W}_h^T)$$

Depending on first component the eigenvalues start from $1 - \gamma\epsilon$ values, so if $\gamma \cdot \epsilon$ eps is near 0 the eigenvalues are distributed around 1. The second term is the imaginary perturbation



Residual Reservoir Computing

This implementation corresponds to a **leaky** version of the **ESN**. The **parameter** α controls how much to weight the linear, orthogonal branch versus the nonlinear, non-orthogonal branch, enabling the model to balance stability and expressive power by combining structured memory (orthogonal branch) with nonlinear dynamics (conventional reservoir branch).



$$\mathbf{h}(t) = \alpha\mathbf{O}\,\mathbf{h}(t-1) + \beta\phi(\mathbf{W_h}\mathbf{h}(t-1) + \mathbf{W_x}\mathbf{x}(t) + \mathbf{b})$$

With both $\mathbf{O}$ and $\mathbf{W}_h$ are untrained.

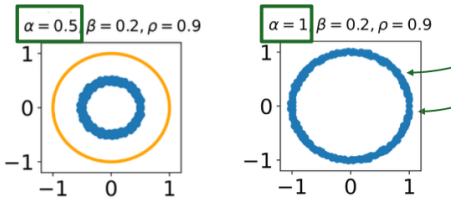
The **linear orthogonal branch** represents a **temporal residual connections** that create a path for the long-range propagation of the input. This **residual connections** its not applied layer by layer, as in deep learning, but from a time step to the next time step

The proximity hyper parameter α determines how close to the edge of stability: 1 means near edge of stability, 0 means far from edge of stability

The **jacobian** of the bias-free autonomous system is:

$$\mathbf{J}_{ResESN} = \alpha\mathbf{O} + \beta\mathbf{W}_h$$

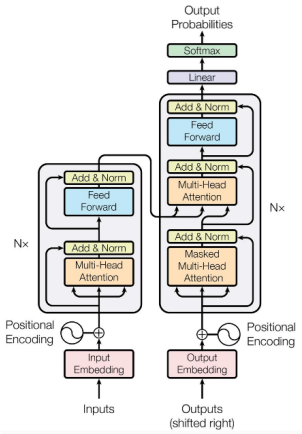
The **eigenvalues** are distributed within a $\beta\|\mathbf{W}_h\|$-tube (thickness of the tube) around a circle of radius α (this because the eigenvalues are distributed across a circle of radius α in the complex space, with a max perturbation of $\beta\|\mathbf{W}_h\|$)



Despite training only the output weights both **Euler ESN** and **Residual ESN** achieve performance close to fully trainable RNNs, while being up to 100 times more efficient in terms of computational time and resource usage. Furthermore these non-dissipative approaches show superior performance across almost all datasets, while **EuSN** give a huge boost on classification of time-series.

(Part 3 Lecture 6) Beyond ESN

Most of the content from this lecture is already included in the Reservoir Computing notes. What follows is the part that was **not** covered there.



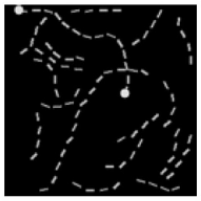
ALTERNATIVES TO TRANSFORMER

Transformers are a powerful deep learning architecture introduced to **handle sequential data without relying on recurrence**. Unlike traditional models such as RNNs, Transformers use **self-attention** mechanisms to capture relationships between elements in a sequence, regardless of their distance. This enables the model to focus on the most relevant parts of the input when processing each element. This models led to state-of-the-art performance in tasks like machine translation, text generation, and beyond.

Issue: this **model costs scales quadratically** as $O(L^2)$ with the sequence length L in terms of other **computations** and **memory** required. This is BAD because if we can't encapsulate all the meaningful informations in the context of(sequence in input) length the model can allucinate. Moreover they are very expensive in term of computation and memory required as they involves a lot of multiplications.

Idea: we could try to avoid transformers, using **RNNs that scale linearly with the sequence length**, potentially more suitable for very long input sequences.

Long range arena: In the search for a viable alternative to Transformers, a wide range of models have been tested on the **Pathfinder task**. It involves 256x256 images containing paths, which are transformed into sequences of pixels. The goal is to classify whether two marked points are connected by a path. Since only one pixel is observed at a time, recognizing these patterns becomes challenging.



(a) A positive example.

Transformers excel at this task thanks to their ability to **attend to all pixels simultaneously**. However, no other model matches their accuracy while being faster, highlighting the need for efficient alternatives to Transformers.

STATE SPACE MODELS

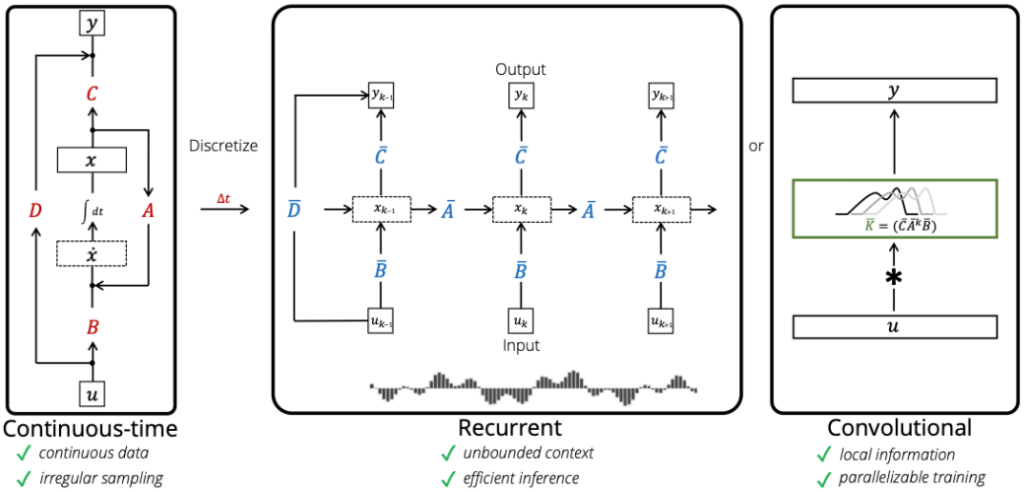
It consists in a simplified class of recurrent layers in which we remove the intermediate non-linearity.

- The recurrent update is: $x'(t) = \mathbf{A}x(t) + \mathbf{B}u(t)$
- The output computation is: $y(t) = \mathbf{C}x(t) + \mathbf{D}u(t)$

Where $x(t)$ is the hidden state, $\mathbf{A}$ is the recurrent weight matrix, and $u(t)$ is the input at time t

Why removing non linearity? RNNs perform computations in a strictly sequential manner, which leads to higher training times and increased computational cost. Removing non linearity **our aim is to create a convolutional variant of our RNNs that allows to parallelize and speed up the training**, but preserving the representation power of RNNs

Three different <u>equivalent</u> views of the same model:



- The **Recurrent implementation** has **unbounded context**, is **efficient in inference** but its strictly **not parallelizable in training**
- The **convolutional** implementation is very powerful as its **strong parallelizable**. The idea is to apply the transformation by A over time: at first time you multiply by A, second time step (starting from scratch) is multiplied by A^2 as it multiply another time by A and so on time k is multiplied by A^k. This is enabled to the fact that non-linearity is avoided.

The question now is **how to initialize A to maximize the memory capacity**? From a reservoir computing point of view we can use **orthogonal matrices**, alternatively authors find out the **HiPPO matrix initialization** that allows to map the sequence in a space such that the memory capacity is maximized. Then a discrete time formulation is derived:

$$
\begin{aligned}
x_k &= \overline{\boldsymbol{A}} x_{k-1} + \overline{\boldsymbol{B}} u_k \qquad & \overline{\boldsymbol{A}} &= (\boldsymbol{I} - \Delta/2 \cdot \boldsymbol{A})^{-1}(\boldsymbol{I} + \Delta/2 \cdot \boldsymbol{A}) \\
y_k &= \overline{\boldsymbol{C}} x_k & \overline{\boldsymbol{B}} &= (\boldsymbol{I} - \Delta/2 \cdot \boldsymbol{A})^{-1}\Delta \boldsymbol{B} \qquad & \overline{\boldsymbol{C}} &= \boldsymbol{C} \\
y_k &= \overline{\boldsymbol{C}}\overline{\boldsymbol{A}}^k\overline{\boldsymbol{B}} u_0 + \overline{\boldsymbol{C}}\overline{\boldsymbol{A}}^{k-1}\overline{\boldsymbol{B}} u_1 + \cdots + \overline{\boldsymbol{C}}\overline{\boldsymbol{A}}\overline{\boldsymbol{B}} u_{k-1} + \overline{\boldsymbol{C}}\overline{\boldsymbol{B}} u_k \\
y &= \overline{\boldsymbol{K}} * u.
\end{aligned}
$$

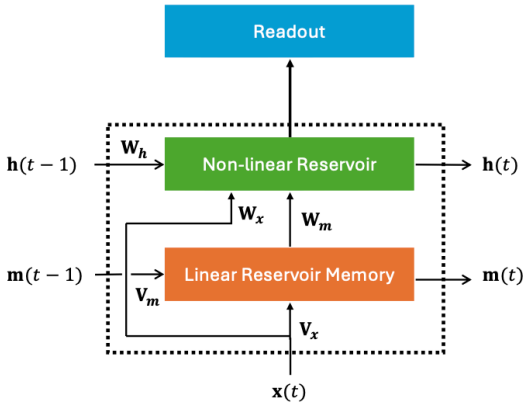
The model operation can be written as a **discrete convolution** (used for training) and then as a RNN in forward (for efficiency reasons).

To enrich the dynamics, the **non-linearity is added in between linear layers** (so non linearity is not in the dynamic of the system but in between them)

how to optimize previous models by avoiding training of these linear layers? Reservoir Memory Network (RMN) are the solution.

Reservoir Memory Network (RMN)

This model has a **dual reservoir:** it combines a **linear memory cell** with a **non-linear reservoir**, efficiently managing long-term untrained dependencies while preserving the computational efficiency of Reservoir Computing.



Idea: one linear and one non linear reservoir, the **first gives long memory capacities** and the **second the non linear needed for representation reasons**

Memory cell:

Linear reservoir layer driven by the external input: $\mathbf{m}(t)) = \mathbf{V}_m\mathbf{m}(t-1) + \mathbf{V}_x\mathbf{x}(t),$ with $\mathbf{x}(t)$ being the input

Here we can use a **circular reservoir** with **optimal memory capacity**

Non-linear reservoir: fed by both the external input and the output of the memory cell.

If using **Leaky-ESN**:

$$\mathbf{h}(t) = (1-\alpha)\mathbf{h}(t-1) + \alpha \tanh(\mathbf{W}_\mathrm{h}\,\mathbf{h}(t-1) + \mathbf{W}_\mathrm{m}\mathbf{m}(t) + \mathbf{W}_\mathrm{x}\mathbf{x}(t) + \mathbf{b}_\mathrm{h})$$

memory state; non-linear state; leaking-rate

If using **EuSN:**

$$\mathbf{h}(t) = \mathbf{h}(t-1) + \varepsilon \tanh((\mathbf{W}_\mathrm{h} - \mathbf{W}_\mathrm{h}^\mathrm{T} - \gamma\mathbf{I})\,\mathbf{h}(t-1) + \mathbf{W}_\mathrm{m}\mathbf{m}(t) + \mathbf{W}_\mathrm{x}\mathbf{x}(t) + \mathbf{b}_\mathrm{h})$$

memory state; non-linear state

Very **huge improvement** compared to standard ESNs and EuSNs. More then 10% more accurate then best ESN and matches ESN performances with a reduction of more then 96% of trainable parameters (readout). This means that **could outperform ESN with a minor cost**

RMN exhibits competitive, and in some cases superior, performance compared to several fully trainable RNN architectures like GRU, NRU, LMU, RNN etc.

(Part 3 Lecture 7) RNNs for text generation

Idea**:** train a NN to predict the next element in a sequence using the previous elements in the sequence as input. Concatenate the prediction to the input and continue iterating. Instead of words we talk about **token**, the atomic part of sentences.

A **token** could be a character, a part of a word or a single word.

Language modeling means modelling $P(\text{ next token} \mid \text{previous tokens})$

If at the output level we got the output distribution of each token (the probability that we are modelling) given the input we are able to **sample** from this output distribution. The **sampling strategy** corresponds to picking from the language model distribution.

IMMAGINE GENERATING TEXT

How previous said it's an **iterative process** over the sequence that we're generating.

CharRNN: the language model can be implemented as an RNN, operating at character level (either vanilla, deep RNN, gated RNN, LSTM, …)

Sampling strategies:

- **Greedy sampling:** the sampling strategy is done in a way that always we select the most probable next character
- **Stochastic sampling**: sample the next token from the estimated probability. In this case we can use a **temperature** hyperparameter to control the randomness of the sampling strategy. A 0 temperature means focus the most on high probable characters as less probables becomes less probables at all, 1 means use the estimated probability. The **logits** (the output before passing trough the softmax output) are divided by the temperature.