Deep Spiking Neural Network, Deep Liquid State Machine e Deep Echo State Network

André Furlan - ensismoebius@gmail.com

Universidade Estadual Paulista Júlio de Mesquita Filho

2023

Deep spiking neural networks

Introduction I

Spiking Neural Networks (SNNs) mimic the workings of the brain by utilizing action potentials, in contrast to continuous values transmitted between neurons.
The term "Spiking" originates from the behavior of biological neurons, which sporadically emit action potentials, creating voltage spikes that convey information (KASABOV, 2019). Figure 4 illustrates these spikes.
An SNN **is not** a one-to-one simulation of neurons. Instead, it approximates certain computational capabilities of specific biological properties. Some studies explore the nonlinearity of dendrites and other neuron features (JONES; KORDING, 2020), yielding remarkable results in the classification.

Introduction II

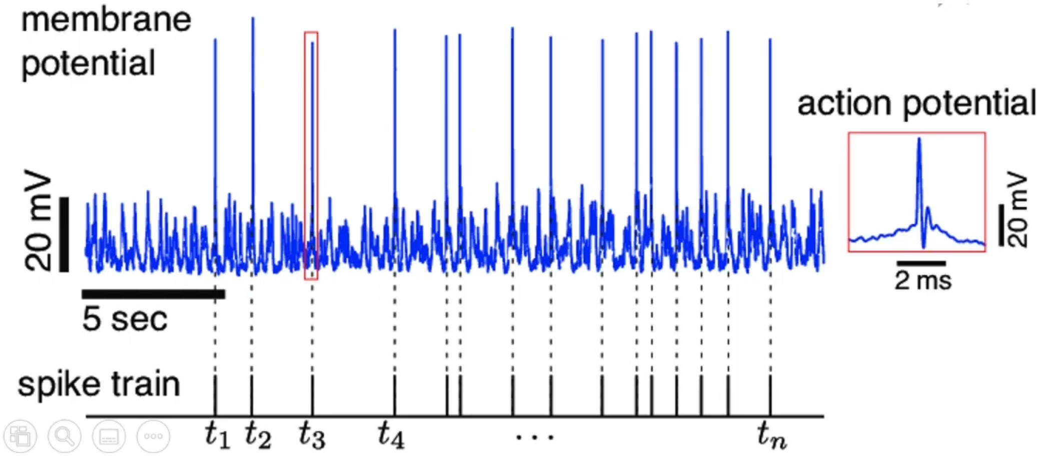


Figure: Spikes from a noisy signal. Source (GOODMAN et al., 2022)

General characteristics

SNNs possess several noteworthy characteristics that distinguish them from traditional machine learning techniques, including classical neural networks. These distinctions encompass (KASABOV, 2019):

- Proficiency in modeling temporal, spatial-temporal, or spectro-temporal data.
- Effectiveness in capturing processes involving various time scales.
- Seamless integration of multiple modalities, such as sound and vision, into a unified system.
- Aptitude for predictive modeling and event prediction.
- Swift and highly parallel information processing capabilities.
- Streamlined information processing.
- Scalability, accommodating structures ranging from a few tens to billions of spiking neurons.
- Minimal energy consumption when implemented on neuromorphic platforms.

Energy characteristics I

Spiking Neural Networks (SNNs) are often considered power-efficient for several reasons:
Event-Driven Processing: SNNs are inherently event-driven. Instead of constantly updating neuron activations and synapse weights as in traditional artificial neural networks (ANNs), SNNs only transmit spikes (action potentials) when a neuron's membrane potential reaches a certain threshold. This event-driven approach reduces the amount of computation required and can lead to significant energy savings.
Sparse Activity: SNNs tend to exhibit sparse activity, meaning that only a small percentage of neurons are active at any given time. This sparsity reduces the number of computations that need to be performed, which is especially beneficial for hardware implementations where most of the energy consumption comes from active components.
Low Precision: SNNs can often work with lower precision than ANNs. While ANNs typically use high-precision floating-point numbers for neuron activations and synaptic weights, SNNs can use lower precision fixed-point or binary representations. Lower precision computations require less energy to perform.

Energy characteristics II

Neuromorphic Hardware: SNNs can be efficiently implemented on specialized neuromorphic hardware, which is designed to mimic the energy-efficient behavior of biological neural systems. These hardware platforms are optimized for the event-driven nature of SNNs, further reducing power consumption.

Energy-Aware Learning Rules: SNNs can employ learning rules that take into account energy efficiency. For example, some learning rules prioritize strengthening or weakening synapses based on their contribution to network activity, which can lead to more energy-efficient learning.

Spike Encoding: SNNs can encode information in the timing and frequency of spikes, which can be a highly efficient way to represent and process data, particularly for event-based sensors like vision sensors or auditory sensors.

Theory: Spikes and potencials

In order to emulate such behavior, let's begin with a simple model: The "Leaky Integrate and Fire neuron" (LIF). The LIF model describes the evolution of membrane potential which the potential decay over time.

$$\tau \cdot \frac{dV}{dt} = -V \tag{1}$$

When a neuron receives a spike, the membrane potential V increases according to a synaptic weight w.

$$V = V + w \tag{2}$$

Such behaviors are depicted in Figure 5.

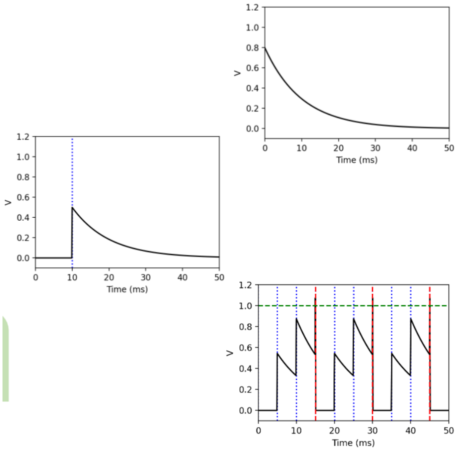


Figure: Evolution of a Spike. Source (GOODMAN et al., 2022)

Theory: Training

As shown in Figure 5, when a neuron reaches a certain threshold, it resets ($V = 0$). Note that $V = 0$ is just one of the possible values of reset, after this the neuron may or may not enter in a refractory period. **These, an another values cited ahead, can be parameterized or learned**.

How do SNNs get trained? Well, this is still an open question. An SNN neuron has an activation-function behavior that is more relatable to a **step function**. Therefore, in principle, we can't use gradient descent-based solutions because this kind of function **is not** differentiable (KASABOV, 2019).

But there are some insights out there that may shed some light on this subject: While some *in vivo/in vitro* observations show that brains, in general, learn by strengthening/weakening and adding/removing synapses or even by creating new neurons or other cumbersome methods like RNA packets, there are some more acceptable ones like the ones in the slide (KASABOV, 2019):

Theory: Training

- Spike Timing-Dependent Plasticity (STDP): The idea is that if a pre-synaptic neuron fires **before** the post-synaptic one, there is a strengthening in connection, but if the post-synaptic neuron fires before, then there is a weakening.
- Surrogate Gradient Descent: The technique **approximates** the step function by using another mathematical function, which is differentiable (like a sigmoid), in order to train the network. These approximations are used only **in the backward pass**, while keeping the step function in the forward pass (KASABOV, 2019).
- Evolving Algorithms: Use the selection of the fittest throughout many generations of networks.
- Reservoir/Dynamic Computing: **Echo state networks** or **Liquid state machines** respectively. These will be discussed further in this presentation.

Demonstration: Simulating a spike neuron I

Before entering into demonstration we need to understand how the exposed theory can be implemented into our neuron, let's see the code:

```python

class Neuron:
    """
    Leak integrate and fire neuron — LIF

    """
    def __init__(self, tau=10, threshold=1):

        # Initial membrane voltage
        self.voltage = 0

        # The smaller tau is the faster the voltage decays
        # When tau is large the neuron acts as an intergrator summing its inputs
        # and firing when a certain threshold is reached.
        # When tau is small the neuron acts as a coincidence detector, firing a
        # spike only when two or more input arrive simultaneosly.
        self.tau = tau

        # The threshold above which the neuron fires
        self.threshold = threshold

        # Time step for decaying (i still don't known what this really is)
        # Bigger the number faster the decay
        self.timeStep = 0.5

```

Demonstration: Simulating a spike neuron II

```python
        # The rate by which the membrane voltage decays each time step
        self.alpha = np.exp(-self.timeStep/self.tau)

    def set_tau(self, tau):
        self.tau = tau
        self.alpha = np.exp(-self.timeStep/self.tau)

    def fire_spike(self):
        if self.voltage > self.threshold:
            self.voltage = 0
            return 1
        return 0

    def add_synaptic_weight(self, weigth):
        # Membrane voltage integration
        self.voltage += weigth

    def iterate(self):
        # Membrane voltage leak
        self.voltage = max(self.voltage * self.alpha, 0)
        return self.fire_spike()
```

And now the simulation (code available in repository):

Demonstration: Simulating a spike neuron III

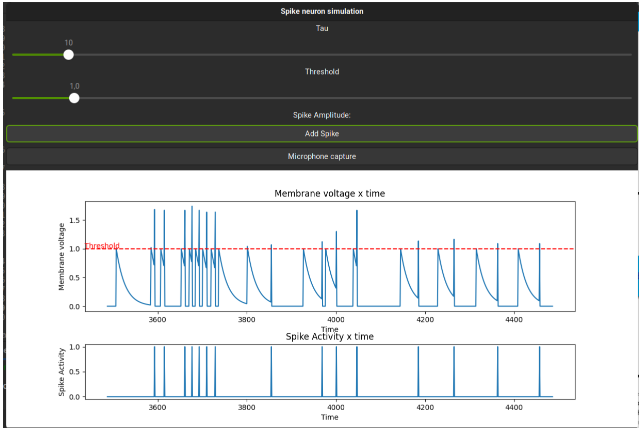


Figure: A simulation a of spiking neuron

Introdução

Estrutura da apresentação

Deep Spiking Neural Network

Introduction I

Spiking Neural Networks (SNNs) mimic the workings of the brain by utilizing action potentials, in contrast to continuous values transmitted between neurons.
The term "Spiking" originates from the behavior of biological neurons, which sporadically emit action potentials, creating voltage spikes that convey information (KASABOV, 2019). Figure 4 illustrates these spikes.
An SNN **is not** a one-to-one simulation of neurons. Instead, it approximates certain computational capabilities of specific biological properties. Some studies explore the nonlinearity of dendrites and other neuron features (JONES; KORDING, 2020), yielding remarkable results in the classification.

Introduction II

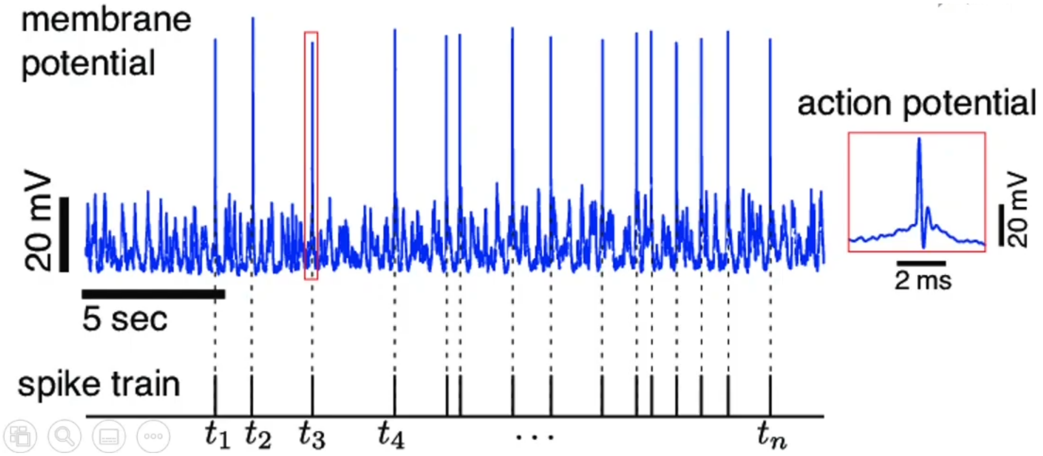


Figure: Spikes from a noisy signal. Source (GOODMAN et al., 2022)

General characteristics

SNNs possess several noteworthy characteristics that distinguish them from traditional machine learning techniques, including classical neural networks. These distinctions encompass (KASABOV, 2019):

- Proficiency in modeling temporal, spatial-temporal, or spectro-temporal data.
- Effectiveness in capturing processes involving various time scales.
- Seamless integration of multiple modalities, such as sound and vision, into a unified system.
- Aptitude for predictive modeling and event prediction.
- Swift and highly parallel information processing capabilities.
- Streamlined information processing.
- Scalability, accommodating structures ranging from a few tens to billions of spiking neurons.
- Minimal energy consumption when implemented on neuromorphic platforms.

Energy characteristics I

Spiking Neural Networks (SNNs) are often considered power-efficient for several reasons:
Event-Driven Processing: SNNs are inherently event-driven. Instead of constantly updating neuron activations and synapse weights as in traditional artificial neural networks (ANNs), SNNs only transmit spikes (action potentials) when a neuron's membrane potential reaches a certain threshold. This event-driven approach reduces the amount of computation required and can lead to significant energy savings.
Sparse Activity: SNNs tend to exhibit sparse activity, meaning that only a small percentage of neurons are active at any given time. This sparsity reduces the number of computations that need to be performed, which is especially beneficial for hardware implementations where most of the energy consumption comes from active components.
Low Precision: SNNs can often work with lower precision than ANNs. While ANNs typically use high-precision floating-point numbers for neuron activations and synaptic weights, SNNs can use lower precision fixed-point or binary representations. Lower precision computations require less energy to perform.

Energy characteristics II

Neuromorphic Hardware: SNNs can be efficiently implemented on specialized neuromorphic hardware, which is designed to mimic the energy-efficient behavior of biological neural systems. These hardware platforms are optimized for the event-driven nature of SNNs, further reducing power consumption.
Energy-Aware Learning Rules: SNNs can employ learning rules that take into account energy efficiency. For example, some learning rules prioritize strengthening or weakening synapses based on their contribution to network activity, which can lead to more energy-efficient learning.
Spike Encoding: SNNs can encode information in the timing and frequency of spikes, which can be a highly efficient way to represent and process data, particularly for event-based sensors like vision sensors or auditory sensors.

Theory: Spikes and potencials

In order to emulate such behavior, let's begin with a simple model: The "Leaky Integrate and Fire neuron" (LIF). The LIF model describes the evolution of membrane potential which the potential decay over time.

$$\tau \cdot \frac{dV}{dt} = -V \tag{3}$$

When a neuron receives a spike, the membrane potential V increases according to a synaptic weight w.

$$V = V + w \tag{4}$$

Such behaviors are depicted in Figure 5.

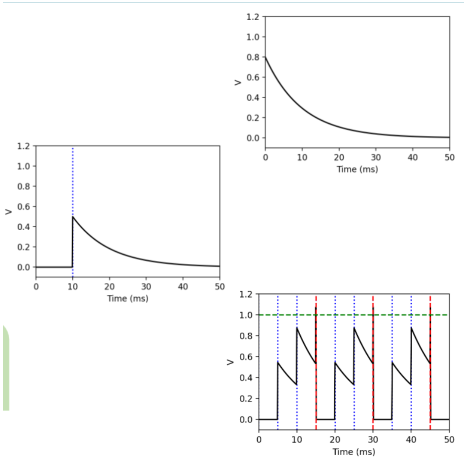


Figure: Evolution of a Spike. Source (GOODMAN et al., 2022)

Theory: Training

As shown in Figure 5, when a neuron reaches a certain threshold, it resets ($V = 0$). Note that $V = 0$ is just one of the possible values of reset, after this the neuron may or may not enter in a refractory period. **These, an another values cited ahead, can be parameterized or learned**.

How do SNNs get trained? Well, this is still an open question. An SNN neuron has an activation-function behavior that is more relatable to a **step function**. Therefore, in principle, we can't use gradient descent-based solutions because this kind of function **is not** differentiable (KASABOV, 2019).

But there are some insights out there that may shed some light on this subject: While some *in vivo/in vitro* observations show that brains, in general, learn by strengthening/weakening and adding/removing synapses or even by creating new neurons or other cumbersome methods like RNA packets, there are some more acceptable ones like the ones in the slide (KASABOV, 2019):

Theory: Training

- Spike Timing-Dependent Plasticity (STDP): The idea is that if a pre-synaptic neuron fires **before** the post-synaptic one, there is a strengthening in connection, but if the post-synaptic neuron fires before, then there is a weakening.
- Surrogate Gradient Descent: The technique **approximates** the step function by using another mathematical function, which is differentiable (like a sigmoid), in order to train the network. These approximations are used only **in the backward pass**, while keeping the step function in the forward pass (KASABOV, 2019).
- Evolving Algorithms: Use the selection of the fittest throughout many generations of networks.
- Reservoir/Dynamic Computing: **Echo state networks** or **Liquid state machines** respectively. These will be discussed further in this presentation.

Demonstration: Simulating a spike neuron I

Before entering into demonstration we need to understand how the exposed theory can be implemented into our neuron, let's see the code:

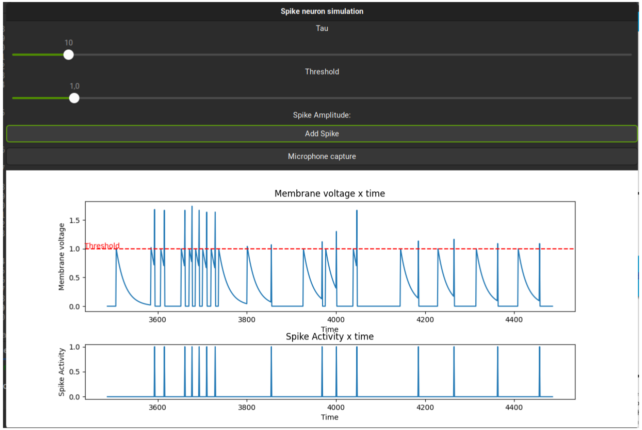
```

class Neuron:
    """
    Leak integrate and fire neuron — LIF

    """
    def __init__(self, tau=10, threshold=1):

        # Initial membrane voltage
        self.voltage = 0

        # The smaller tau is the faster the voltage decays
        # When tau is large the neuron acts as an intergrator summing its inputs
        # and firing when a certain threshold is reached.
        # When tau is small the neuron acts as a coincidence detector, firing a
        # spike only when two or more input arrive simultaneosly.
        self.tau = tau

        # The threshold above which the neuron fires
        self.threshold = threshold

        # Time step for decaying (i still don't known what this really is)
        # Bigger the number faster the decay
        self.timeStep = 0.5

```

Demonstration: Simulating a spike neuron II

```
        # The rate by which the membrane voltage decays each time step
        self.alpha = np.exp(-self.timeStep/self.tau)

    def set_tau(self, tau):
        self.tau = tau
        self.alpha = np.exp(-self.timeStep/self.tau)

    def fire_spike(self):
        if self.voltage > self.threshold:
            self.voltage = 0
            return 1
        return 0

    def add_synaptic_weight(self, weigth):
        # Membrane voltage integration
        self.voltage += weigth

    def iterate(self):
        # Membrane voltage leak
        self.voltage = max(self.voltage * self.alpha, 0)
        return self.fire_spike()
```

And now the simulation (code available in repository):

Demonstration: Simulating a spike neuron III


Spike neuron simulation

Tau

10

Threshold

1,0

Spike Amplitude:

Add Spike

Microphone capture

Figure: A simulation a of spiking neuron

Deep Liquid State Machine

The liquid state machine (LSM) has emerged as a computational model that is more suitable than Turing machines for describing neural networks. It is used to process continuous streams of data, typically in the form of spike trains, and it maps streams of inputs into streams of outputs. These outputs may depend on the previous states created by the streaming data (MAASS,).
LSM is model for adaptive computing systems. Considering that the training stage is the most expensive and sensible one, these kind of networks are trained only at the readouts. These readouts, usually, consists of only a single neuron which is called *"projection neuron"* that extracts information from a micro-circuit somewhere in the LSM. These structures can be used as inputs to another area of the neural network and can be modeled as a perceptron, linear gate, sigmoidal gate, spike neuron and others (MAASS,). The networks structure consists of several neurons randomly and recurrently connected forming the *"liquid part"* which is interpreted by the readouts. The word *"liquid"* can sometimes be take literally like in this work (FERNANDO; SOJAKKA, 2003) where a bucket of water has been taken the role of the liquid part of the network as can be seen in Figure 7.

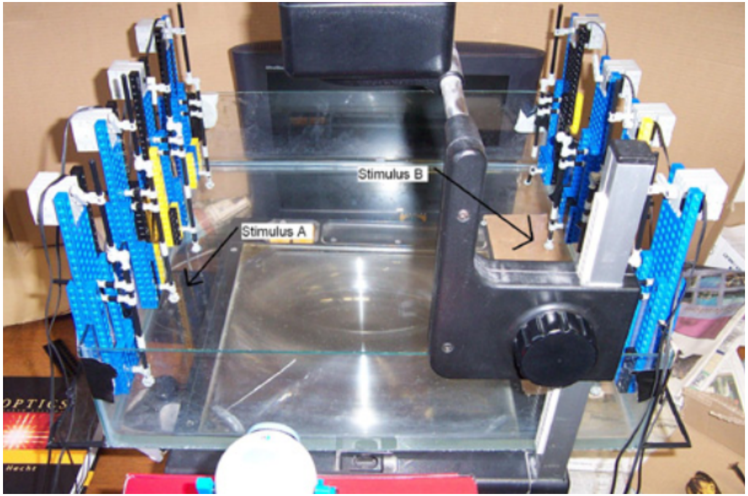
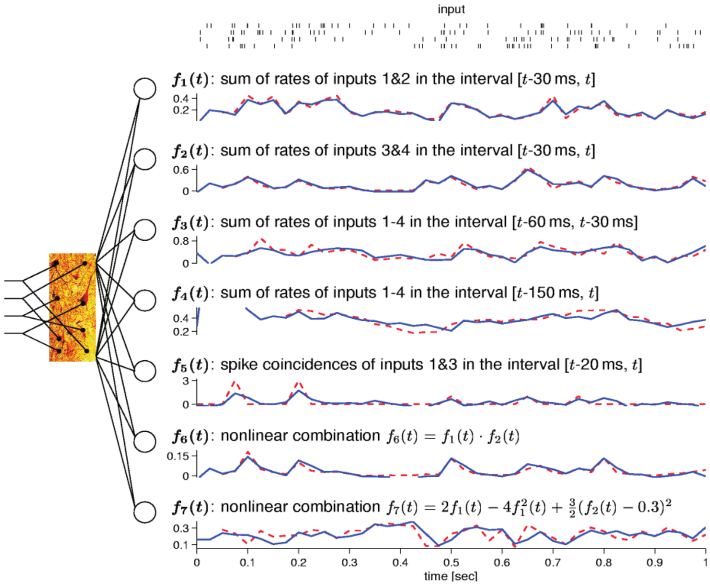


Figure: A liquid network. Source: (FERNANDO; SOJAKKA, 2003)

LSM can be adapted to multiple computation as the projection neurons can extract as many different characteristics one desires as can be seen in Figure 8.



input
$f_1(t)$: sum of rates of inputs 1&2 in the interval [t-30 ms, t]
0.4
0.2
$f_2(t)$: sum of rates of inputs 3&4 in the interval [t-30 ms, t]
0.6
0
$f_3(t)$: sum of rates of inputs 1-4 in the interval [t-60 ms, t-30 ms]
0.8
0
$f_4(t)$: sum of rates of inputs 1-4 in the interval [t-150 ms, t]
0.4
0.2
$f_5(t)$: spike coincidences of inputs 1&3 in the interval [t-20 ms, t]
3
0
$f_6(t)$: nonlinear combination $f_6(t) = f_1(t) \cdot f_2(t)$
0.15
0
$f_7(t)$: nonlinear combination $f_7(t) = 2f_1(t) - 4f_1^2(t) + \frac{3}{2}(f_2(t) - 0.3)^2$
0.3
0.1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
time [sec]

Figure

More liquid

Also, according to (HASANI et al., 2020) the inspiration comes from the nervous system of the nematode *C. elegans* which, despite having just 302 neuron, have pretty complex behavior. In this model the networks parameter changes over time according to a set of nested differential equations **while it is already in use** i.e. **this model does not need a training phase in order to adapt**.
Conversely, LNNs introduce dynamic connectivity patterns, allowing information to flow and interact in a fluid manner.
Liquid Neural Networks, also known as Liquid State Machines (LSMs), were first introduced by Wolfgang Maass(MAASS; NATSCHLäGER; MARKRAM, 2002). The primary departure from traditional ANNs lies in their dynamic and recurrent architecture. While conventional ANNs consist of fixed layers of interconnected neurons, LNNs employ a vast collection of interconnected neurons that are constantly in a state of change. This dynamic behavior allows LNNs to process temporal data, sequential information, and streaming inputs with remarkable flexibility and efficiency.

- Adaptability: Their dynamic nature enables them to respond dynamically to varying data distributions, making them well-suited for tasks involving non-stationary data.

- Robustness: LNNs have shown improved robustness against noise and input variations. The fluid-like behavior allows them to self-adjust and filter out irrelevant information, leading to enhanced generalization capabilities.
- Exploration of Solution Space: LNNs encourage solution space exploration by providing flexibility in the network's structure. The dynamic connectivity patterns enable the network to explore diverse pathways, potentially discovering novel solutions to complex problems.
- Reduced Overfitting: Due to their continuous learning capabilities, LNNs are less prone to overfitting, which often occurs in static networks, resulting in more accurate and generalizable models.

Being f the neural network, $I(t)$ its inputs, t current time, θ and A the hiperparameters and τ is time constant than the equation 5 represents the Liquid Time-Constant recurrent neural networks (LTC) hidden state's derivative (HASANI et al., 2020).

$$\frac{dx(t)}{dt} = -\left[\frac{1}{\tau} + f(x(t), I(t), t, \theta)\right] x(t) + f(x(t), I(t), t, \theta)A \quad . \tag{5}$$

To simplify a neuron in these kind of networks can be expressed like in the listing .

```python
import numpy as np
import matplotlib.pyplot as plt

class LiquidTimeConstantNeuron:
    def __init__(self):
        self.time_constant = 1.0

    def adjust_time_constant(self, input_data):
        # Adjust the time constant based on input data
        self.time_constant = 1.0 + input_data

    def update_state(self, state, time_step):
        # Update the state based on the time constant and time step
        return state - (state / self.time_constant) + np.sin(time_step)

# Create a Liquid Time-Constant Neuron
ltc_neuron = LiquidTimeConstantNeuron()

# Simulate over time
duration = 10.0
time = np.arange(0, duration, 0.1)
states = [0.0]

for t in time[1:]:
    input_data = np.sin(t)  # Simulated input data (can vary over time)
    ltc_neuron.adjust_time_constant(input_data)
    new_state = ltc_neuron.update_state(states[-1], t)
    states.append(new_state)

# Plot the results
plt.plot(time, states)
plt.xlabel("Time")
plt.ylabel("State")
plt.title("Neuron with Adjustable Time Constant")
plt.show()
```

```python
import torch
import torch.nn as nn

class ESN(nn.Module):
  def __init__(self, input_size, reservoir_size, output_size):
    super(ESN, self).__init__()
    self.reservoir_size = reservoir_size
    self.W_in = nn.Linear(input_size, reservoir_size)
    self.W_res = nn.Linear(reservoir_size, reservoir_size)
    self.W_out = nn.Linear(reservoir_size, output_size)

  def forward(self, input):
    reservoir = torch.zeros((input.size(0), self.reservoir_size))
    for i in range(input.size(1)):
      input_t = input[:, i, :]
      reservoir = torch.tanh(self.W_in(input_t) + self.W_res(reservoir))
      output = self.W_out(reservoir)
    return output

# Example usage
input_size = 10
reservoir_size = 100
output_size = 1

model = ESN(input_size, reservoir_size, output_size)
```

Deep Echo State Network

Referências I

FERNANDO, C.; SOJAKKA, S. Pattern recognition in a bucket. In: BANZHAF, W. et al. (Ed.). *Advances in Artificial Life*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 588–597. ISBN 978-3-540-39432-7.

GOODMAN, D. et al. *Spiking Neural Network Models in Neuroscience - Cosyne Tutorial 2022*. Zenodo, 2022. Disponível em: ⟨https://doi.org/10.5281/zenodo.7044500⟩.

HASANI, R. et al. *Liquid Time-constant Networks*. 2020.

JONES, I. S.; KORDING, K. P. *Can Single Neurons Solve MNIST? The Computational Power of Biological Dendritic Trees*. 2020. Disponível em: ⟨https://arxiv.org/abs/2009.01269⟩.

KASABOV, N. K. *Time-space, spiking neural networks and brain-inspired artificial intelligence*. [S.l.]: Springer, 2019.

Referências II

MAASS, W. Liquid state machines: Motivation, theory, and applications. In: ______. *Computability in Context*. [s.n.]. p. 275–296. Disponível em: ⟨https://www.worldscientific.com/doi/abs/10.1142/9781848162778_0008⟩.

MAASS, W.; NATSCHLäGER, T.; MARKRAM, H. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, v. 14, n. 11, p. 2531–2560, 2002.