ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

An introduction to Spiking Neural Networks

Author: Ghassen KARRAY

Project in Computational Neurosciences

Supervisors

Alexander Mathis

Alberto Chiappa

Lucas Stoffl

Submission date: 11.06.2021

Lausanne, academic year 2020-2021

Contents

Abstract

Spiking Neural Networks, a concept initially used by computational neuroscientists, is stimulating a lot of interest from the Machine Learning research community. They represent computational models of biological neural networks. Harnessing their full potential in a practical ML setup has however proved to be difficult, due to the high complexity of their neuron models, network architectures and learning rules. In this report, we try to summarize the key background concepts needed to formally state the problem. Next, we give a short overview on the solutions proposed in literature. We continue with a hands-on case study on implementing a Reinforcement Learning agent using SNNs. The agent did not learn and reach good performance. The case study however gives the reader a good idea on the typical challenges one would encounter in a similar setup. We conclude with a little series of research question directly related to the main topic, SNNs.

I Introduction

Machine Learning (ML) has been increasing in popularity within the research community over these past few years. Even though its actual success is very recent, the field is not new and dates back to the 1950s [1][2][3].

Although some of the major theoretical breakthroughs happened the following decades, and the building blocks of the tools that are being used today already established by the 2000s, it is really after the 2012 edition of the ImageNet Challenge, which was a large scale competition for the image classification task, that the field's evolution became impressive.

Said edition was won by the AlexNet model [4], a Convolutional Neural Network-based (deep) architecture that was trained on GPUs. According to *The Economist* : "*Suddenly people started to pay attention, not just within the AI community but across the technology industry as a whole*" [5].

This was actually the start of the success story of **Deep Learning** (DL), rather than Machine Learning, but the broader field of ML as a whole was actually revived by this event.

Machine Learning is the study of computer algorithms (also called models) that learn through experience and by the use of data. Deep Learning is a special case of Machine Learning in which the mathematical model that learns is an Artificial Neural Network (ANN), a mathematical abstraction inspired from the biological neural networks that constitutes the brain.

Although Deep Learning refers to the usage an ANN using multiple layers, the term is implicitly being use to denote the overall process which is used when training ANNs in the state-of-the-art (present input to the input layer, propagate activation through the multiple layers in a feed-forward fashion, compute a loss, back-propagate the error signals through the layers, update model parameters, repeat).

Inspirations from the brain happen not only on the model level of ML, but also on the conceptual level, as the concept of learning by itself is directly inspired from the brain's cognitive ability to learn.

Computational Neuroscience is the branch of Neuroscience that employs mathematical models (also called computational models) combined with theoretical analyses to construct abstract models of nervous systems in order to understand the principles that govern their dynamics.

Machine Learning also relying on rigorous mathematical modelling, the parallel with Computational Neuroscience (when studying the specific *Learning* ability) cannot be unseen. However, the two differ in their end goal. The first aims for a the more pragmatic goal of being used to solve problems. Namely, we say a model is good/bad at performing a task. The second however aims for being used as a way to validate hypotheses made in a scientific research setup. Namely, computational neuroscientists build these models after long hours of actual biological neural data. The computational neuroscientist construct a mathematical model that captures principles that governs neuronal dynamics, then run simulations to validate whether or not said model predict the actual biological neural data.

In some sense, some of ML's models are actually, or can be seen (to some extent) as computational neuroscience models that would not have bio-plausibility as end goal. To cite only a few, there's *first generation ANNs* (Hopfield networks [6], Boltzmann Machines [7] ...), but also *second generation ANNs* (these are the ones we call **Deep Neural Networks** (DNN) throughout this report, namely, Multi Layer Perceptron with non-linear activations), i.e ANNs used in Deep Learning.

Deep Learning is now reaching a relatively high level of maturity, and its community is becoming more and more interested in expanding the field to use even more bio-plausible models of neural networks to use in Machine Learning setups.

Spiking Neural Networks (SNN) fill this criteria perfectly, as they would be the next step towards full bio-plausibility. The two key differences they present in comparison with classical ANNs are :

- Their activation function is non-differentiable ; They output Dirac delta pulses, also called ”spikes”.
- They incorporate the concept of time in their model definition (in some cases, they are modeled as Recurrent Neural Networks (RNN) in order to be seen as classical ANNs)

Spiking Neural Networks are specifically the kind of models used in Computational Neuroscience. Thus, once again Machine Learning is taking inspiration from Computational Neuroscience, however this time, the Deep Learning field is hopefully mature enough to embrace the full complexity of the actual models used by computational neuroscientists. As we will see throughout this report, this is not an easy task.

The report will be divided as such : we will first present, in Section II some of the background needed to fully grasp the concepts we discuss in following sections. Section III gives a small overview on SNNs. Section IV goes through some of the existing frameworks that handles them. In Section V we go through some of the challenges encountered when trying to use SNNs in a Reinforcement Learning setup and in Section VI, we discuss some potential research topics directly related to SNNs in a Machine Learning setup.

II Background

II.1 Machine Learning

There exists many definitions of Machine Learning out there. The one given by *UC Berkeley* [8] is straightforward and conveys the information needed : Basically, the learning system of a machine learning algorithm can be broken down into three main parts :

- **A decision process :** In general, machine learning algorithms are used to make a prediction or classification. Based on some input data, which can be labelled or unlabeled, the algorithm will produce an estimate about a pattern in the data.
- **An error function :** Serves to evaluate the prediction of the model. If there are known examples (supervised), an error function can make a comparison to assess the accuracy of the model.
- **A model optimization process :** If the model can fit better to the data points in the training set, then weights are adjusted to reduce the discrepancy between the known example and the model estimate. The algorithm will repeat this evaluate and optimize process, updating weights autonomously until a threshold of accuracy has been met.

Although it isn't part of the definition (nor is it present in other definitions, always implicit), almost exclusively all known machine learning algorithm are algorithms designed to run on Von Neumann architectures, as it is the computational model everyone uses by default.

II.1.i Supervised Learning

In this setup, the dataset being used has been pre-labeled and classified by users to allow the algorithm to see how accurate its performance is.

Given a set of N training examples of the form $\{(x_1, y_1), ..., (x_N, y_N)\}$ such that x_i is the feature vector of the i-th example and y_i is its label, a learning algorithm seeks a function $g : X \to Y$, where X is the input space and Y is the output space such that $g(x) = \arg\min R_{emp}(f) = \frac{1}{N} \sum_i L(y_i, g(x_i))$, with R_{emp} denoting what the empirical risk, and L denoting the loss function chosen by the user.

When an analytic solution to this equation is not defined, we try to approximate $g(x)$ by a function $\hat{g}(x \mid w)$ such that w denotes the model parameters that we would update iteratively with Gradient Descent until a threshold of accuracy has been met. This is the area where Deep Neural Networks are most typically used [9].

II.1.ii Unsupervised Learning

The raw dataset being used is unlabeled and an algorithm identifies patterns and relationships within the data without help from users.

Given a set of N training examples of the form $\{(x_1), ..., (x_N)\}$ such that x_i is the feature vector of the i-th example (no labels). A learning algorithm, in this context, seeks to build a compact internal representation. It is unsupervised in the sense that we do not tell the model what it must learn (no labels), but allow it to find patterns and draw conclusions from the unlabeled data.

Typical unsupervised learning tasks are : clustering (grouping unlabeled data based on their similarities or differences), dimensionality reduction (reducing the number of features of input data to a manageable size while also preserving the integrity of the dataset as much as possible), data generation (generating data based on distribution of input dataset).

Boltzmann Machines, Hopfield Networks are examples of artificial neural networks that are used in Unsupervised Learning tasks such Memory Retrieval from partial pattern, or Pattern Recognition [**?**]. Variational Autoencoders are examples of architectures of Deep Neural Networks that are being used in unsupervised data generation [**?**].

II.1.iii Reinforcement Learning

The dataset in this context is derived from the consequences of the model's decisions. It uses a "rewards/punishments" system, offering feedback to the algorithm to learn from its own experiences by trial and error.

The typical framing of a Reinforcement Learning (RL) scenario (see Figure 1) is : an agent takes actions in an environment. Upon action selection, a reward is issued and the environment state is updated. These are fed back into the agent. Repeat ...

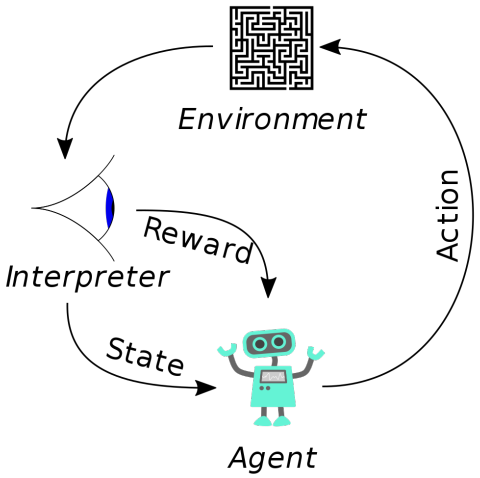

Figure 1: Representation of the Reinforcement Learning loop

A Reinforcement Learning task is most often modeled as a Markov Decision Process :

- A set of environment and agent states, S

- A set of actions, A, of the agent
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability of transition (at time t) from state s to state s' under action a.
- $R_a(s, s')$ is the immediate reward after transition from s to s' with action a.

At each time t, the agent receives the current state s_t and reward r_t. It then chooses an action a_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the transition (s_t, a_t, s_{t+1}) is determined. The goal of a reinforcement learning agent is to learn a *policy*: $\pi : A \times S \to [0, 1]$, $\pi(a, s) = \Pr(a_t = a \mid s_t = s)$ which maximizes the expected cumulative reward.

Deep Neural Networks can be used to model the policy of an agent as well as to estimate the expected cumulative reward for a given state s_t under a policy π (also called state-value function $V_\pi(s)$) [10].

II.2 Computational Neuroscience

Although the term was coined only recently [11], actual computational neuroscientific work began as early as 1907, when Louis Lapicque invented one of the most famous neuron models, that is still being used today (infact, it is the most used model in the Computational Neuroscience community to this day), the Integrate-and-Fire neuron model [12]. About 40 years later, Hodgkin & Huxley developed the first bio-physically detailed neuron model, which is called Hodgkin-Huxley neuron model in literature [13].
Neuroscientists tend to put neuron models in either one of these two categories :

- **Biophysical models**, which are the most detailed models and can be seen as *mechanistic* models, as they mathematically describe the internal mechanisms that give rise to the neuron's behavior, by formalizing the physical properties of the neuron. This is done by the description of the membrane potential as a function of the input current, which in turn is described by the activation/inactivation of *ion channels*, that let/block ions from flowing in and out of the neuron. Such models are also called **Hodgkin-Huxley models**.
- **Phenomenological models**, which are simpler models and can be seen as *functional* models, as they mathematically describe the function that is being performed by the neuron's behavior (*integration* of input, *firing* if a certain threshold is reached), by postulating a mathematical form that is not exactly derived from biophysical processes. This is done by the description of the membrane potential as a function of the input current. Such a function is determined by setting up the constraints that seem to dictate neuronal dynamics and formalizing them mathematically. A widely used family of such models are **Integrate-and-Fire models**.

Depending on the level of details needed to answer the research question at hand, the computational neuroscientist may choose one model or the other. It is not that one model is more bio-plausible than the other, as both are mainly used to validate hypotheses constructed from analysis of neural data, but rather a choice on the level of details needed to describe the investigated phenomenon.

The computational neuroscientist's job does not stop at modelling single neurons. It also encompasses the study of how single neurons connect with each other to give rise to neural networks. The topology of said neural networks is thoroughly investigated to derive mathematical abstractions of the principles that underlie it. Namely, it has been shown that connections in certain parts of the brain are not random, neither fully-connected, but rather structured in a specific ways [14]. Again, hypotheses on the connectivity structure are made and computational models of biological neural networks are created to validate said hypotheses.

Last but not least, the rigorous mathematical formulation of the cognitive process that is learning falls also into the interest of Computational Neuroscience, in some cases.
If the investigation is only on a high level, and does not map to a lower level biological process, it is likely to be in the realm of Cognitive Science. If however, the learning process modeled/investigated is expressed in terms of a biological process (most often, the *Plasticity* process), then it is Computational Neuroscience. In this context, computational neursocientists conjecture "learning rules" that are expressed with cognitive terms, and map those to biological processes that happen in the biological neural network.
Spike-timing-dependent plasticity (STDP) is one of such biological processes, that adjusts the strength of connections between neurons in the brain, demonstrated experimentally in [15].
Many learning rules have been designed based on this process [16][17][18][19][20][21].

II.3 Artificial Neural Networks

The term Artificial Neural Networks denotes all kinds of computational models that aims at modelling networks of neurons. These range from the simplest first generation neural networks alluded to in I to the extremely complex and main interest of this report, Spiking Neural Networks.
The Artificial part of the term stresses the fact that these are not biological neural networks, meaning that they are not actual physical and biological networks of neurons, but rather, abstract mathematical models of neurons, that we simulate on whatever computer architecture suits us best.

In this part, we will try to summarize the building block of the Deep Neural Networks that were the source of success of ML over the last years. To do so, we will begin with the neuron model.
In DNNs, a neuron can be seen as a small computing machine, that receives input from

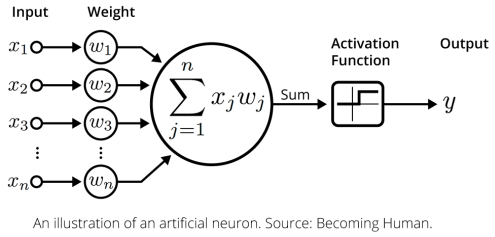

Figure 2: Representation of a neuron

outputs of other neurons, computes a function and pass it to further neurons down the chain (as input to other neurons etc ...). A representation of the computation that happens inside a neuron can be seen in Figure 2. The computation happenning can be decomposed into two distinct parts. First, the summation of weighted version of the output of previous neurons (+ eventually a bias term), then, computing an activation based on the weighted sum. The activation function can be linear or non-linear, however non-linear activations are preferred as they give better approximating capabilities to the model.

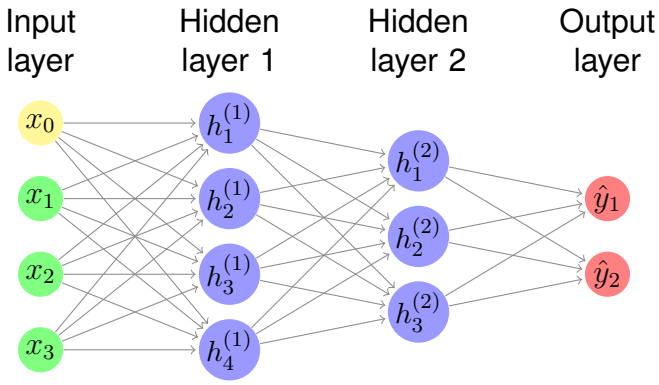

Figure 3: Representation a Fully Connected network with 2 hidden layers

To build a network out of those, we typically take a bunch of neurons, structure them in layers, and label them as input neurons. We then create multiple hidden layers, with varying number of neurons, and connect them together in a sequential fashion (each layer after the other). The final layer is denoted as the output layer. It can be a single scalar or a vector.

The most basic topology of the resulting Directed Acyclic Graph can be seen in Figure 3, the layers are said to be Fully Connected. Another connection strategy, making use of Weight Sharing, results in the so called Convolutional Layers.

The weights are attributes to the connection between neurons, also called the synapses (each synapse has its own weight). The bias term however is an attribute specific to neurons (each neuron has its bias term). These represent the learnable parameters. Whenever we say that the model is learning, we mean that the whole set of such parameters in a network are changing, each learning step. The direction of the change follows Gradient

Descent [9] types of rules. To put it simply, this makes the parameters change in such a way that the output of the network gets better at approximating a function of the input. Combined with the backpropagation method [9], it became the perfect candidate to the Swiss knife of complexe data intensive approximation tasks.

Of course, this toy model is but a simple architecture typically used relatively basic tasks. Typically, since the boom of Deep Learning in 2012 (alluded to in Section I), the pace at which the field is evolving is fascinating. Many complex architecture, tweaks to the optimization procedure and even other neuron types (those used in Recurrent Neural Networks [9]) have been created, helping solve engineering problems of all sorts.

The key takeaway of this section would be : individual artificial neurons (model described above) are approximating machines, and (deep) networks of such neurons are even more powerful approximating machines.

III Spiking Neural Networks

III.1 Model description

III.1.i In computational neuroscience

In computational neuroscience, the main goal of Spiking Neural Networks simulations is, and always has been, to validate mathematical abstractions made and hypothesized about the principles that govern neuronal dynamics. In that sense, such models seek to be, by definition, as much biologically plausible as possible (mimic biological networks of neurons, fit experimental data).
Once such a computational model is created, validated and becomes accepted in the community, it can be used and built upon to ask more complex research questions about neuronal dynamics. If that research question is answered and the answer is also validated within the computational model and by experimental data, the whole model can be extended. That gives, in a sense, a scientific component to the model.

Remark : It becomes relevant at this point to contrast the end goals of DNNs in an ML setup to the SNNs in a Computational Neuroscience setup. What makes the model good, from a Machine Learning perspective, is its performance in approximating the function given to it. Engineers typically use those models as a mean to solve a problem. We can refer to this as a *practical* end goal. On the other hand, what makes the model good, from a Computational Neuroscience perspective, is its ability to fit experimental data, explain biological phenomena and generalize. We can refer to this as a *scientific* end goal.

Now we can dive into what makes a Spiking Neural Networks, from a Computational Neuroscience perspective. And we will begin with neuron models.

Basically, depending on the research question asked, you would need different levels of details in your model. Typically, if you investigate the effect of certain chemical elements or ions on the nervous systems and the detailed response of individual neurons to them, you would need a biophysical model (namely, a Hodgkin-Huxley model, possible

multi-compartmental to an even higher accuracy) that encompasses the dynamics of ion channels present on the membrane of biological neurons [13].

On the other hand, say you do not want to worry at all about the dynamics of individual ion channels. You are still interested in the membrane potential of your neurons, but you are okay with making sound assumptions to approximate the complex behavior of the multitude of ion channels in order to make your model easily scale. In that case, a good idea is to use what are called Point Neurons. The most popular of those models is the Integrate-and-Fire model [12] (and its variants).
Such neuron models keep only one state variable, the membrane potential, in comparison to 4 state variable for the case of H-H neuron models. This makes their large scale simulation easier than bio physically detailed neurons.

This only state variable is enough already to construct the Spiking behavior out of it. Let us define the **leaky Integrate-and-Fire** model this way [22]:
Modelling the cell membrane of the neuron as a capacitor with capacity C and finite leak resistance R, the following is the differential equation that dictates the dynamics of the neuron's membrane potential :

$$\tau_m \frac{\partial u(t)}{\partial t} = -[u(t) - u_{rest}] + RI(t) \tag{1}$$

From an electrical engineer point of view, this is the equation of a leaky integrator with time constant $\tau_m = RC$, where u_{rest} is the resting membrane potential.
The solution of such an equation can be found analytically and is of the form :

$$u(t) = \frac{R}{\tau_m} \int_0^\infty e^{-\frac{s}{\tau_m}} I(t-s) ds$$

With $u_{rest} = 0$ for the sake of simplicity, for any time dependent input $I(t)$ defined for all $t \in \mathbb{R}$

Each time instant t_j where the membrane potential reaches a certain threshold θ, t_j is recorded, and the neuron is said to trigger a *spike*, and the variable u is reset to a certain u_{reset}. When say N other neurons connect to this neurons, through *synapses* that we model as having an *efficacy* represented by w_j for $j = 1, ..., N$, the current $I(t)$ received by the neuron can be written becomes expressed as follows :

$$I(t) = \sum_{j=1}^{N} \sum_{f} w_j \alpha_j (t - t_j^{(f)}) + I^{ext}(t)$$

In this context, α_j represents the generic time course of the post-synaptic current response, resulting from neuron j outputting a spike. The sequence $(t^{(f)})$ for $f = 1, ..., \infty$ represent the history of spiking times of neuron j.

Remark : It is also worth noting here, that the ways in which we connect individual neurons differ very much from the DNN case. The computational neuroscientists trying to fit experimental data about actual biological neural network topologies, they merely model what they see. And the ways with which biological neurons connect is not random, neither is it perfectly structured. Namely, it could be difficult to perfectly define a ”layer” of neurons, as the modeled neurons would be distributed in continuous space, thus making putting bounds to delimit where a population start and stop a tricky thing to do.
Even if you did approximate your way through it, a major difference with DNNs would be that, in some specific cases, the populations (equivalent to ”layers” in DNNs) you find yourself modelling might have interconnections.
Finally, although some micro-circuits appear to have some feed-forward aspects in parts of the brain, in most cases, you would find yourself modeling multiple populations that have feed-forward as well as *feed-back* connections. Not the type of feed-back connections modeled by Recurrent Neural Networks (self-connection), but the type that would make you're connection graph Cyclic. In this context, we would not gain much by ordering populations (such as layer #1, layer #2 ...).

Within both of the models discussed above, as well as almost any other computational model of neurons within the neuroscience community, the dimension of time is explicit in the model definition, as it is strongly believed the behavior of neuronal dynamics can be described with **continuous time dynamic systems**.

A final particularity of SNNs would be that, the spiking information, emitted the moment that the membrane potential reaches the firing threshold, does not reach the connected neurons instantaneously. Theoretically speaking, the integration and the decision to fire happen at the *soma* of the neuron, and it takes time for that information to travel down the axon to the synapse location. For a specific neuron, the synapses it forms with other neurons can be at arbitrary distance from the soma. This has to be addressed at some point by the computational neuroscientist when a model is created.

III.1.ii In machine learning

The key result that opened the door to scientific interest in SNNs for the Machine Learning community was that, like classical ANNs and DNNs, Spiking Neural Networks, or networks of neuron models typically used in Computational Neuroscience, were too capable of approximating functions, and thus, computing [23][24][25].
Given some assumptions about how information is encoded into and decoded from neural spikes information (or from another perspective, if you manage to find an encoding procedure that would encode real values into spikes, and decoding procedure that would do the inverse), you can use Spiking Neural Networks to approximate functions, and for many other sorts of Machine Learning tasks [26][27][28][29].

Remark : Infact, it has been at least theoretically proven that they are at least as much computationally powerful as their analog counter part (DNNs) [30], as DNNs can

be seen as a special case of SNNs, in which we assume that information is coded in the firing rate of spiking neurons. In that case, instead of keeping track of the actual output of a spiking neuron at each time step (spiking or not spiking), the neuron keeps track of another quantity, that is a theoretical firing rate, based on the input it integrates. The specific way in which the integrated input firing rates are related to the output firing rate is defined by the activation function.
There are also promising results in converting fully trained DNNs into SNNs, although for the time being, it is accompanied by a small decrease in performance [31][32].

As typical Machine Learning tasks tend to be completely agnostic to the notion of time, efforts have been made towards re-modeling neuron models used in SNNs as an recurrent neuron model (typically used in DNNs) [33].
Recurrent neuron models are neuron models similar to those discussed in Section II.3, however they have the particularity of having self-connection, which means they hold memory, to some extent, of their previous activations. They have had great success in tasks involving temporally structured data. We could harness the memory capabilities of RNNs to make them model the Integrate part of the Integrate-and-Fire model, and use Dirac delta functions as activation function. The individual neurons internal state variables should evolve as dictated by the system of differential equations of the corresponding neuron model (such as eq. 1 for the leaky Integrate-and-Fire).
Further more, we could incorporate delays incurred by axonal propagation of spikes into such a model.

One good question to be asked at this point : Do we keep thinking Neural Networks architectures in a feed-forward fashion (and with strict layer ordering), or do we switch to biological inspiration and allow for complex feed-forward + feed-back connections, when we build SNNs for Machine Learning ?
This area is still being explored to this day, and we need more time and research to understand more clearly the advantages and disadvantages obtained from both methods.

III.2 Learning

In the same veins as the previously asked question, one could ask a similar one about the learning rule we would use in a Spiking Neural Network. Would we try to implement some kind of Gradient Descent, possibly coupled with a backpropagation mechanism, or would we search for other learning rules from the numerous ones already postulated by computational neuroscientists (evoked in Section II.2). This question is also being actively explored.

It is important to pinpoint however, that using back-propagation as it is being used in DNNs presents one problem, on a practical level. Namely, the activation of spiking neurons in an SNN would be Dirac delta functions (spikes). This function is non-differentiable, which makes the derivatives supposedly computed in the back-propagation step not de-

fined. Efforts have been made to circumvent this problem, by approximating derivatives, or smoothing the Dirac activations to make them differentiable [33]. These methods however are not bio-plausible [26] (not that it is fundamentally important from an ML perspective).

From another perspective however, this opens up a whole new dimension which Machine Learning research could further investigate. Namely, **learning rules**.

III.3 Neuromorphic hardware

We cannot conclude our discussion on SNNs without evoking **Neuromorphic Hardware**. It is to the Spiking Neural Networks what GPUs are to the DNNs. It is specialized hardware specifically designed to optimize SNNs simulation. It tries to do so by running on a hardware architecture that mimics networks of neurons. Examples of such hardware systems are detailed in [34][35][36].

Although some of these chips may still have Von Neumann components running some coordination tasks (updating the weights for on-chip learning for example), the aim is to have the actual relevant computation for the task happen on non-Von Neumann type of architecture. The architecture has the particularity of being massively parallel (millions of neurons). This makes computation highly distributed. The communication is asynchronous and event-driven (or spike-driven), which makes the information exchanged extremely sparse, yielding very low power consumption [37].

Although initially designed to run Computational Neuroscience simulations, ML could harness such hardware if performance of SNNs in ML tasks improves over the next few years.

IV Existing frameworks

Now, let's have a look to the different programming frameworks we can use to create SNNs. As a mental representation, you can see these framework as points on a line, such as the two opposite direction of the line represent respectively Computational Neuroscience and ML. This is only a toy representation, as in reality things are much more complex than this. However, this mental representation is helpful at explaining the challenges you may encounter, and the reasons behind them.

IV.1 Computational Neuroscience frameworks

These are programming frameworks typically used by computational neuroscientists to build computational models of biological neural networks and run simulations to validate hypotheses constructed based on experimental data.

The choice of what framework to use in such a context is highly motivated by the level of details needed in model you are building. Typically, if you want to make a very realistic network made of multi-compartmental Hodgkin-Huxley models, in which you capture all the detailed dynamics of ion channels, you would rather use a software framework like NEURON [38] or [39]. NEURON is typically well suited for tasks in which you want to model cells with complex anatomical and biophysical properties. The simulation holds a

notion of a volume and you would create neurons with a specific morphology and place it in a specific position, which makes the neural networks built extremely realistic. The amount of computation needed for simulating a single neuron makes it impossible to scale to large scale networks of neurons without having mobilize computing clusters.

A better choice if you want to scale-up to a large number of neurons and are interested in more abstract network-related research questions, you can use more higher level software like Brian2 [40] or NEST [41]. Brian2 for example has both Hodgkin-Huxley and integrate-and-fire type models. It is useful for models with a few compartments, but not with detailed reconstruction of dendritic trees however.

NEST is also a framework better suited for modeling large scale networks, as it makes use of point neurons, but it can also simulate H-H types of models by the use of a two-compartmental model (intermediary between point and multi-compartmental neurons) [**?**]

IV.2 Nengo

Although technically it is part of the Computational Neuroscience, this framework really sits somewhat in the middle of the spectrum (Computational Neuroscience - Machine Learning).

Nengo [42] is a software tool that can be used to build and simulate large-scale models based on the Neural Engineering Framework [43] (NEF). What makes it depart a little from the Computational Neuroscience end of the spectrum are some the assumptions made in the NEF. One good thing though, is you can easily drop the NEF and still use Nengo for simulations of networks of spiking neurons. Meaning, the NEF ultimately uses Integrate-and-Fire types of neurons. It makes some assumptions about the encoding/decoding schemes (or rather, it constructs a useful way of encoding/decoding information in spikes), as well as about how connection weights are constructed and information is processed. But ultimately, it runs simulations on spiking neurons. So you could use the already built models of neurons available and connect them with weight derived as you see fit, and run the overall on Nengo's backend.

The NEF basically proposes three quantitatively specified principles that enable the construction of large-scale neural models. It takes the following axioms (that may or may not have biological plausibility) :

- **Representation :** A population of neurons collectively represents a time-varying vector of real numbers through non-linear encoding and linear decoding. *This principle fixes the encoding and decoding scheme with which we go from real values to spikes and the inverse. It is a form of population coding.*

- **Transformation :** Linear and non-linear functions on those vectors are computed by linear decodings that are used to analytically compute the connections between

populations of neurons. *This principle fixes how basic computational operations can be translated into neural computational operations.*

- **Dynamics :** The vectors represented by neural populations can be considered state variables in a (linear or non-linear) dynamical system, and recurrent connections can be computed using principle 2. *This principle fixes how the system evolves. Namely, it views the system as a dynamical one, which is bio-plausible and ease the incorporation of recurrent dynamics between population into the model.*

Nengo has some learning rules, that you can attribute to connection to make their weight generation process differ from the NEF, while abstracting some of the numerous learning rules postulated by computational neuroscientists. Namely, they propose the Oja [44] rule, particularly useful for unsupervised tasks, the BCM [45] rule, also useful for unsupervised tasks, and the PES [46], for supervised tasks.

IV.3 Machine Learning frameworks

On the other end of the spectrum, lies pure Machine Learning frameworks. Of those, PyTorch or TensorFlow, for example. If one wants to build a network that learns with backpropagation, there is no need re-invent the wheel. One could simply define a neuron model with the dynamics spiking neurons in one of those frameworks, and run it like plain old DNNs [47].

You can also use NengoDL [48], which combines Nengo's backend to run the simulations with elements from DL such as back-propagation for parameter optimization.

Another framework worth mentioning would be BindsNet [49]. This framework does not use elements like backpropagation from DL. Instead, it uses, for the time being, only learning rules from computational neuroscience literature. It relies however on Torch Tensors for neural network computations, which could potentially harness GPUs. It also comes with modules handling all the processing pipeline, targeted for ML applications. It makes it easier to focus not so much on setting up the environment for your model, but on actual network building and learning rule benchmarking, for new comers to SNNs from the Machine Learning world.

V Case study : Reinforcement Learning

This section will present a case-study on some of the challenges one would face when implementing an RL agent using SNNs. First, we will go through the different simulations that were run, trying to detail as much as possible the mathematical formulation of the computation of interest. These did not end in successful training of agents. The second part is a reflection on the chosen track as well as the key take-home messages.

V.1 Simulations

V.1.i Nengo

The following are the results of simulations run with Nengo [50]. The SNNs simulation designed with Nengo interfaces with an Open AI Gym [51] environment. The chosen environment is CartPole-v0. In this particular setup, the SNN would model the agent, and the Open AI Gym the environment represented in Figure 1. The algorithm we try to model is the Sarsa algorithm with Greedy Policy as defined in [10]. The architecture built for this purpose is presented in Figure 4.

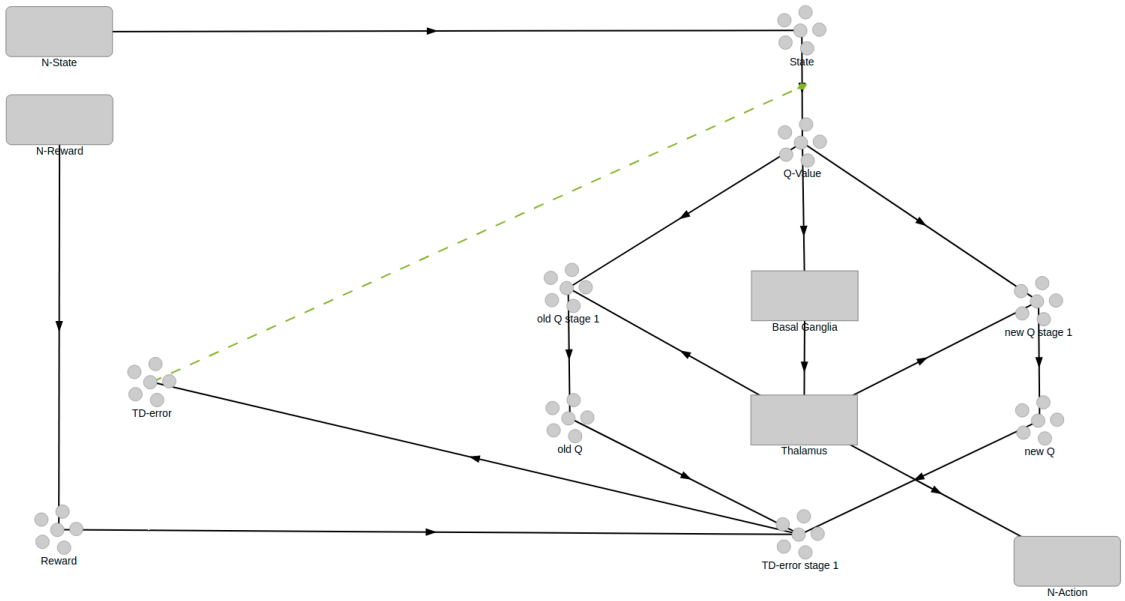

Figure 4: Representation of the architecture

To interface with the Nengo simulation, the Open AI Gym environment is wrapped in a class, which acts as a controller between the environment and the SNN simulation. This class would typically store useful state variables about the simulations, and methods of this class would interface with the Nengo simulation via Nengo Node objects.

Nengo Nodes can be visualized in 4 as rectangular objects. In this architecture, we have 3 Nodes N-State, N-Reward and N-Action. The SNN receives some input from N-State and N-Reward, and sends some output to the environment through N-Action. Node objects can be seen as nodes of the spiking neural network, in the sense that they output a value at every step of the simulation, can connect/can be connected to neurons. What makes them different from population of neurons is their output comes from Von Neumann-type of computation (in contrast with the non-Von Neumann-type of computation that is the neural computation).

At each update of the Nengo simulation, the state of the Gym Environment is fed to the SNNs through the N-State Node, and the reward through the N-Reward Node. The action is taken from the output of Thalamus sub-module, at each step, and used to update the

Open AI Gym.

To understand in details the encoding scheme used in Nengo, please refer to [43][50]. To put it simply, Nengo uses a non-linear encoding scheme with a linear decoding scheme (Principle 1 of the NEF). This principle is translated into a form of population coding. To represent the state $s_t = \begin{pmatrix} v & p & \theta & \dot{\theta} \end{pmatrix}$ with a population of N spiking neurons, such vector of *signals* is said to drive the neurons based on each neuron's tuning curve (which is similar to the current-frequency curve). The tuning curve describes how much a particular neuron will fire as a function of the input signal. By default, N tuning curve are chosen uniformly at random. This is the encoding procedure.

To decode the input signal originally encoded by the pattern of spikes, the spike train are first filtered with a temporal filter that accounts for postsynaptic current activity.
The filtered spike trains are then multiplied with decoding weights and summed together to yield an estimate of the input based on the spikes. The decoding weights are determined by minimizing the squared difference between the decoded estimate and the actual input signal, **at compile time** (a radius given as parameter to the populations of neurons sets the bounds over which we optimize the decoding parameters).

A connection between two populations of neurons (A to B) is seen, in Nengo, as a chaining of decoding-encoding. Thus, what is classically referred to as connection weights between populations of neurons is simply the matrix multiplication of decoder weights of A with the encoder weights of B. $w_{AB} = D_A E_B$.

To incorporate computation in connections between populations of neurons, Nengo relies on the second principle of the NEF. Say you want to connect population A to population B in such a way that the value represented by population B would be a certain function f of the value represented by population A. $V_B = f(V_A)$. In that case, you would determine the decoding weights of A, D_A by minimizing the squared difference between the decoded estimate and the result of apply f to the actual input signal.

It is relevant at this, to differentiate between two types of representation, as well as computation : Neural (representation in spike patterns of population, computation expressed in weights between populations) and Von Neuman (representation is float numbers and computation expressed in Python code)

In the architecture presented, the state vector is fed to the network through the N-State node. It is connected to the State population, which represents the state vector in neural representation. The Q-Value population represent the vector of Q-Values for each action in the action space. The connection weights between the State population and the Q-Value are not defined through the process of the NEF, as these are the ones that have to be learned. As we want to implement the Sarsa algorithm, we are looking for a way of estimating the Q-Values for each action given a state s_t. These have to be learned, that is way a learning rule is attributed to this connection, namely, the PES rule.

The PES rule can seen as performing gradient descent on the output weights of the presynaptic neuron based on an error signal. It is expressed as such :

$$\Delta w_{ij} = \kappa \alpha_j e_j E a_i(x)$$

where w_{ij} denotes weight between neuron number i of the pre synaptic population and neuron number j of the post synaptic population, κ the learning rate, α_j parameter of the post synaptic neuron, e_j encoder of the post synaptic neuron, E the error signal and $a_i(x)$ the activity of the pre synaptic neuron.
When given the TD-error as error signal, this learning rule has been demonstrated to be useful in Q values estimation [52].

The connection seen in green in Figure 4 is the connection through which is fed the TD-error value, represented by the TD-Error population, to the PES learning rule for the State-Q-Values connection.

The Q values are then fed to the Basal Ganglia sub-module. It is a reusable Nengo built-in network object, capable of a specific kind of neural computation. It can be seen as making the $\arg\max$ neural computation. Typically, if it is fed a vector $Q = \begin{pmatrix} 0.63 & 1.04 \end{pmatrix}$, it would output a vector of the form $BG = \begin{pmatrix} \beta & 0 \end{pmatrix}$ such that β is a negative real value. It outputs 0 in the dimension where the input was maximum, and negative value elsewhere.

Next, the output of the Basal Ganglia are fed to the Thalamus sub-module, which simple make the neural computation of 1-hot encoding. The output of the Basal Ganglia is transformed in such a way that it is rather 1 in the dimension where the input was maximum and 0 elsewhere. $BG = \begin{pmatrix} \beta & 0 \end{pmatrix}$ would become $THAL = \begin{pmatrix} 0 & 1 \end{pmatrix}$.

Such output is passed through the N-Action node, and the Open AI Gym environment can potentially be updated. We say potentially here, as you are not forced to update the Open AI Gym environment at each Nengo SNN simulation update. Typically, you may want to update the Open AI Gym environment once every K Nengo simulation steps, as you may want to have some computation happening that involve delayed connection. In this context, it would probably be best to skip some Gym updates until you are sure the output of the Thalamus is effectively response to the actual state s_t.

To compute the TD-error for action taken at step t, we need three ingredients (see Sarsa in [10]). First, the reward received after taking that action, namely, R_{t+1}. Second, the discounted Q-Value of the action that would be taken by the greedy policy at step $t+1$, namely, $\gamma Q(s_{t+1}, a_{t+1})$. Finally, minus the Q-Value of the action that was taken at step t, namely, $Q(s_t, a_t)$.

The reward is fed to the SNN through the N-Reward node, connecting with the Reward population. The computation of both of the other values is computed neurally.
For the computation of the TD-error for action taken at step t is computed and the weights are updated by it when the SNN is actually making the choice for step $t+1$. Let us suppose

the Nengo simulation reached the point where the state that is presented is the state at step $t+1$. Same for the reward. At this simulation step, the Q values would be those for state s_{t+1}. 2 copies of these values are made. Each of these copy is used in a chain of 2 populations of neuron to compute some information. The output of these two processes is the population *new Q* on the one hand and *old Q* on the other hand. Both of these populations, as well as the reward population, are combined into a preliminary population for TD-error computation.

The difference between the populations *new Q* and *old Q* lies in the delay of the connection. Namely, we would want *new Q* to represent information about the Q value at step $t+1$. To do so, since we are actually at point in a simulation where state s_{t+1}, and thus, the output of the Q-Value population is actually the one we need for *new Q*, we define the delay on the connection between *new Q* and the TD-error computation stages as $Delay$. If we make sure that the delay between *old Q* and the TD-error computation stages is $Delay + TIMESTEP \times K$ with K being the number of Nengo simulation steps before updating the environment Gym, and $TIMESTEP$ representing the timestep of the Nengo simulation, we would have, arriving at the TD-error computation stages, the reward at $t+1$, the Q relevant at $t+1$, the Q relevant at t.

We make use of those to compute the TD-error, which is represented by the TD-Error population, and the output value of which is fed to the PES learning rule of the learning connection, as an error signal.

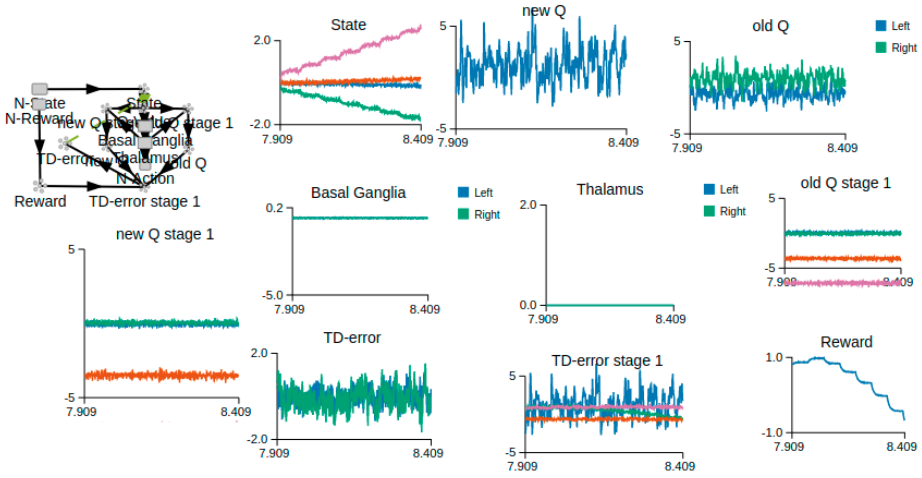

Figure 5: Visualizing population values with Nengo-GUI

Figure 5 presents a typical debugging setup for such an architecture. Typically, you would have the values represented by each population of the neural network.

V.2 Take-home points

Initially, the goal of this project was to train an agent using SNNs for a Reinforcement Learning task. Having some background in SNNs usage in computational neuroscience setups and classical ANNs usage in ML setups, I assumed it wouldn't be that hard merging

the two together and train my own SNN on a Reinforcement Learning task. It turned out to be a quite difficult task, and I did not succeed in accomplishing it by the end of this project. I did however, learn a lot from this journey that led to a failure.

Take-home (1) : The primary goal of this report is to summarize the concepts, challenges and solutions one would encounter when trying to use SNNs in an ML setup. Thus, the important take-home would be : take the time to fully understand the tool you are going to use as well as when and how to use it.

A second point that is important, in continuation with the previous, is the choice of the programming framework to use. It can sometime be cumbersome to go through all the detailed representations capabilities of state-of-the-art Computational Neuroscience frameworks if the task you are trying to accomplish is an ML task.

Take-home (2) : Choose the framework on which you will run SNNs simulation wisely. Preferably begin with a more ML-oriented type of frameworks, succeed in a simple task, and make your way through more complex frameworks/concepts.

VI Discussion

To conclude this report, it is worth mentioning that there are many aspects of SNNs that were not covered in this report. Many details not present, many nuances explained. It tries however, to make an overview of the current state-of-the-art of SNNs usage, by linking to useful references.

One of the most important aspects that should be addressed in following research would be the encoding/decoding schemes one would implement when using an SNN in an ML setup. The ML setup makes it particularly important to interpret the output spikes of SNNs as real values. This relied on making some assumptions on the types of neural codes used by biological neural networks.

Seeing biological neural networks (and consequently SNNs) as a model of computation makes it relevant to investigate coding scheme of such a system. Namely, we are looking for a conversion mechanism between plain information (binary representation) into information contained in some aspect of the system's dynamics. There are several such mechanisms investigated, such as rate coding (information contained is translated to firing rates), temporal coding (information is translated into temporal information contained within the spike trains) are particularly interesting, and SNNs proved to have the capabilities to harness both [23][24].

Beyond this, to what extent should we care about biological plausibility ? Is there something to be gained in searching for more bio-plausibility by, for example, differentiating between strictly excitatory neurons and strictly inhibitory neurons (contrary to letting neurons project to multiple other neurons in such way that some of the weights of the synapses are positive and some are negative) ?

Acknowledgements

I would like to express my deep and sincere gratitude to Prof. Alexander Mathis for making this project possible. I also want to dedicate a special thanks to Alberto Chiappa and Lucas Stoffl who had the patience of guiding me every week throughout the making of this project.

Bibliography

[1] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950.

[2] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[3] Nils J. Nilsson, Terrence J. Sejnowski, Halbert White, Terrence J. Sejnowski, and Halbert White. Learning machines, 1965.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[5] From not working to neural networking. `https://www.economist.com/special-report/2016/06/23/from-not-working-to-neural-networking`, 06 2016.

[6] J. J. Hopfield. *Neural Networks and Physical Systems with Emergent Collective Computational Abilities*, page 457–464. MIT Press, Cambridge, MA, USA, 1988.

[7] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147–169, 1985.

[8] What is machine learning ? `https://ischoolonline.berkeley.edu/blog/what-is-machine-learning/`, 06 2020.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[11] Eric L. Schwartz. *Computational Neuroscience*. MIT Press, Cambridge, MA, USA, 1990.

[12] Louis Lapicque. Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation. *J. Physiol. Pathol. Gen.*, 9:620–635, 1907.

[13] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952.

[14] Eyal Gal, Rodrigo Perin, Henry Markram, Michael London, and Idan Segev. Neuron geometry underlies a universal local architecture in neuronal networks. *bioRxiv*, 2019.

[15] Henry Markram, Joachim Lübke, Michael Frotscher, and Bert Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps. *Science (New York, N.Y.)*, 275:213–5, 02 1997.

[16] H.Sebastian Seung. Learning in spiking neural networks by reinforcement of stochastic synaptic transmission. *Neuron*, 40(6):1063–1073, 2003.

[17] Răzvan Florian. Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural computation*, 19:1468–502, 07 2007.

[18] Alireza Nadafian and Mohammad Ganjtabesh. Bio-plausible unsupervised delay learning for extracting temporal features in spiking neural networks, 2020.

[19] Kendra S. Burbank. Mirrored stdp implements autoencoder learning in a network of spiking neurons. *PLOS Computational Biology*, 11(12):1–25, 12 2015.

[20] Danilo Jimenez Rezende and Wulfram Gerstner. Stochastic variational learning in recurrent spiking networks. *Frontiers in Computational Neuroscience*, 8:38, 2014.

[21] Bernhard Nessler, Michael Pfeiffer, and Wolfgang Maass. Stdp enables spiking neurons to detect hidden causes of their inputs. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

[22] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, USA, 2014.

[23] Wolfgang Maass. Fast sigmoidal networks via spiking neurons. *Neural Comput.*, 9(2):279–304, February 1997.

[24] Nicolangelo Iannella and Andrew D. Back. A spiking neural network architecture for nonlinear function approximation. *Neural Networks*, 14(6):933–939, 2001.

[25] Terence David Sanger. Probability density methods for smooth function approximation and learning in populations of tuned spiking neurons. *Neural Computation*, 10(6):1567–1586, 1998.

[26] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, 2019.

[27] Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. Reinforcement learning using a continuous time actor-critic framework with spiking neurons. *PLOS Computational Biology*, 9(4):1–21, 04 2013.

[28] Peter Diehl and Matthew Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience*, 9:99, 2015.

[29] Filip Ponulak and Andrzej Kasiński. Supervised learning in spiking neural networks with resume: Sequence learning, classification, and spike shifting. *Neural computation*, 22:467–510, 10 2009.

[30] Wolfgang Maass and Henry Markram. On the computational power of circuits of spiking neurons. *Journal of Computer and System Sciences*, 69(4):593–616, 2004.

[31] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in Neuroscience*, 13:95, 2019.

[32] Peter U. Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2015.

[33] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine*, 36(6):51–63, 2019.

[34] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[35] Christian Mayr, Sebastian Hoeppner, and Steve Furber. Spinnaker 2: A 10 million core processor system for brain simulation and machine learning, 2019.

[36] Andreas Grübl, Sebastian Billaudelle, Benjamin Cramer, Vitali Karasenko, and Johannes Schemmel. Verification and design methods for the brainscales neuromorphic hardware system, 2020.

[37] Dingbang Liu, Hao Yu, and Yang Chai. Low-power computing with neuromorphic engineering. *Advanced Intelligent Systems*, 3(2):2000150, 2021.

[38] N.T. Carnevale and M.L. Hines. *The NEURON Book*. Cambridge University Press, 2006.

[39] J.M. Bower and D. Beeman. *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*. Springer New York, 2012.

[40] Marcel Stimberg, Romain Brette, and Dan FM Goodman. Brian 2, an intuitive and efficient neural simulator. *eLife*, 8:e47314, aug 2019.

[41] M. Gewaltig and M. Diesmann. NEST (NEural Simulation Tool). *Scholarpedia*, 2(4):1430, 2007. revision #130182.

[42] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. Nengo: a python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7:48, 2014.

[43] C. Eliasmith and C.H. Anderson. *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. A Bradford book. MIT Press, 2003.

[44] E Oja. A simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15(3):267—273, 1982.

[45] EL Bienenstock, LN Cooper, and PW Munro. Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. *Journal of Neuroscience*, 2(1):32–48, 1982.

[46] Trevor Bekolay, Carter Kolbeck, and Chris Eliasmith. Simultaneous unsupervised and supervised learning of cognitive functions in biologically plausible spiking neural networks. pages 169–174, 01 2013.

[47] Guillaume Chevalier. Spiking neural network (snn) with pytorch: towards bridging the gap between deep learning and the human brain. `https://guillaume-chevalier.com/spiking-neural-network-snn-with-pytorch-where-backpropagation-engenders-stdp-hebbia` 07 2019.

[48] Daniel Rasmussen. Nengodl: Combining deep learning and neuromorphic modelling methods, 2019.

[49] Hananel Hazan, Daniel J. Saunders, Hassaan Khan, Devdhar Patel, Darpan T. Sanghavi, Hava T. Siegelmann, and Robert Kozma. Bindsnet: A machine learning-oriented spiking neural networks library in python. *Frontiers in Neuroinformatics*, 12:89, 2018.

[50] Nengo. `https://www.nengo.ai/`.

[51] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[52] Daniel Rasmussen, Aaron Voelker, and Chris Eliasmith. A neural model of hierarchical reinforcement learning. *PLOS ONE*, 12(7):1–39, 07 2017.