VALENCIAN INTERNATIONAL UNIVERSITY

MASTER'S THESIS

Spiking Neural Networks: A Review

Author:
Unai Garay Maestre

DNI:
48770527Q

Supervisor:
Carlos Fernández Musoles

A thesis submitted in fulfilment of the requirements
for the degree of Master in Artificial Intelligence

February 4, 2021

Academic Course 2018-2019

Convocatory: 2ª

Credits: 12 ECTS

viu Universidad Internacional de Valencia

"*We have not yet learned what the brain has to teach us"*

Chris Eliasmith

VALENCIAN INTERNATIONAL UNIVERSITY

Abstract

Master in Artificial Intelligence

Spiking Neural Networks: A Review

by Unai Garay Maestre

This report follows the research and development of a final master thesis on artificial intelligence. The purpose of this research is to review Spiking Neural Networks (SNNs) in every aspect, from a mid-level perspective to a high level abstraction. Spiking Neural Networks promise a series of improvements over conventional Artificial Neural Networks (ANNs): energy efficiency, faster inference, biological plausibility and hardware friendly among others. These neural networks work using spikes which are discrete events that also occur in the human brain, and because they are discrete the algorithms used for training conventional ANNs are ineffective. This work tries to sum up some algorithms for training SNNs that are trying to overcome this problem and to give some practical examples using the Nengo framework and its variants in order to put those concepts into practice.

Acknowledgements

First of all, I have to thank the Valencian International University for giving me the opportunity to develop this project and for providing me with some of the knowledge that has allowed me to finish the work. Secondly, I want to thank my director Carlos Fernández Musoles who has supported me conceptually and who has given the motivation for starting the thesis on this topic. Next, to my friends who have given me the strength to keep up the good work while not forgetting to enjoy the journey. And last but no least, I have to give most of the credit to my family who has been supporting me from the beginning and has not let me down, encouraging me to keep going and never to give up.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1
1.1 Motivation . . . 1
1.2 Objectives . . . 2

2 Spiking Neural Networks 3
2.1 Introduction . . . 3
2.1.1 Artificial Neural Networks . . . 3
2.1.2 Spiking Neural Networks . . . 4
2.1.3 SNNs vs ANNs . . . 5
2.2 Models . . . 6
2.2.1 Hodgkin-Huxley . . . 6
2.2.2 Integrate and Fire . . . 7
2.2.3 Spike Response Model . . . 8
2.2.4 Izhikevich . . . 9
2.3 Learning . . . 9
2.3.1 Hebbian Rule . . . 9
2.3.2 Spike-timing dependent plasticity (STDP) . . . 10
2.3.3 Unsupervised Learning . . . 11
2.3.4 Supervised Learning . . . 11
Backpropagation Issues . . . 12
Some supervised learning methods for SNNs . . . 12
2.3.5 Reinforcement Learning . . . 14
2.4 Deep Learning in Spiking Neural Networks . . . 14
2.4.1 Deep Fully Connected SNNs . . . 16
2.5 Deep Convolutional Spiking Neural Networks . . . 18
2.6 Spike Coding . . . 20
2.6.1 Neural Coding . . . 21
Rate Coding . . . 22
Temporal Coding . . . 23
2.6.2 Neuromorphic Hardware . . . 23
2.6.3 Event-Based Datasets . . . 25

3 Technologies 27
3.1 Nengo . . . 27
3.1.1 The Nengo Ecosystem . . . 28
Nengo . . . 28
NengoDL . . . 29
Nengo GUI . . . 29
Nengo SPA . . . 30

Simulation Backends 31

4 Experiments and Results **33**
4.1 Experimenting with Nengo 33
4.1.1 Basis 33
Principle 1: Representation 33
Principle 2: Transformation 36
Principle 3: Dynamics 37
4.1.2 Learning in Nengo GUI 38
4.2 Classification with NengoDL and Tensorflow 40
4.2.1 Dataset: Fashion MNIST 41
4.2.2 SNN Architecture 42
4.2.3 Results 43
4.3 Classification using native Nengo and Nengo GUI 45
4.3.1 Nengo GUI SNN Architecture 45
4.3.2 Results 45

5 Conclusions **47**
5.1 Project Summary 47
5.2 Evaluation 47
5.3 Further Work 48
5.4 Final Thoughts 48

Bibliography **49**

List of Figures

2.1 Human brain structure and ANNs structure. Image downloaded from Medium, 2015 . . . 3
2.2 Spiking neuron model and its connections. Image downloaded from Frontiers in Neuroscience, 2016 . . . 4
2.3 Hodgkin-Huxley model's basic components. Image downloaded from *Hodgkin–Huxley model* . . . 6
2.4 Asymetric Spike-Timing Dependent Plasticity's learning window. Image downloaded from *Spike-timing dependent plasticity* . . . 11
2.5 Fully Connected Neural Network. Created using *Publication-ready NN-architecture schematics* . . . 15
2.6 Deep Convolutional Neural Network. Created using *Publication-ready NN-architecture schematics* . . . 16
2.7 Process of convolution in a Convolutional Neural Network. Retrieved from *freeCodeCamp* . . . 16
2.8 Fully Connected Spiking Neural Network. Downloaded from the paper *A brain-inspired spiking neural network model with temporal encoding and learning* . . . 17
2.9 Fully Connected SNN with backpropagation. Downloaded from the paper *Deep Learning in Spiking Neural Networks* . . . 18
2.10 Representation Learning for layer-wise unsupervised learning of a spiking CNN (*Multi-layer unsupervised learning in a spiking convolutional neural network*) . . . 19
2.11 Usage of Difference-of-Gaussian (DoG) filter (*Deep Convolutional Spiking Neural Networks for Image Classification*) . . . 19
2.12 Convolutional SNN architecture converted from a normal CNN (*Spiking Deep Convolutional Neural Networks for Energy-Efficient Object Recognition*) . . . 20
2.13 Spike Trains. Image downloaded from *Simulating neural spike trains* . 21
2.14 Spike Trains Encoding and Decoding. Created using the framework **nengodl** . . . 21
2.15 Spike-Count Rate. Downloaded from *Variability of Spike Trains and Neural Codes* . . . 22
2.16 Neuromorphic chips. Left is SpiNNaker and right is DARPA SyNAPSE with TrueNorth chips from IBM . . . 24
2.17 Event Based Vision Hardware. By *Prophesee Event Based Vision Explanation* . . . 25
2.18 Event-Based data of a building captured using a Dynamic Vision Sensor. Snapshot taken from the video *The Event-Camera Dataset and Simulator* . . . 26

3.1 The Nengo Ecosystem. Downloaded from *Nengo* . . . 28
3.2 Nengo GUI. Main components . . . 30

3.3 Semantic Pointer Architecture Unified Network by University of Waterloo Centre for Theoretical Neuroscience 30

4.1 Encoding signal using Nengo 34
4.2 Tuning curves 34
4.3 Tuning curves 35
4.4 Temporal filter 35
4.5 Estimated Output with different number of neurons 36
4.6 Transformation between populations 37
4.7 Dynamics. Oscillator 38
4.8 Nengo GUI. No learning applied 39
4.9 Nengo GUI. Learning white signal by using PES rule 41
4.10 Zalando's Fashion MNIST dataset examples 41
4.11 Comparison of the accuracy of the three models for different time-steps 44
4.12 Image prediction neuron spike comparison. Ankle Boot 44
4.13 SNN architecture for Fashion MNIST in Nengo GUI 45
4.14 SNN inference on Fashion MNIST test set in Nengo GUI 46

List of Tables

2.1 Comparison between conventional Artificial Neural Networks and Spiking Neural Networks. Red background indicates a disadvantage and green background indicates an advantage 6

4.1 Zalando's Fashion MNIST dataset label correlations 42
4.2 Spiking Neural Network Architecture for Fashion MNIST dataset . . . 42
4.3 SNNs results comparison for Fashion MNIST 43

Dedicated to my lovely family, who are behind all my achievements

Chapter 1

Introduction

Artificial Intelligence (AI) is a computer science field that has been around for more than half a century (1956, first introduced by John McCarthy Yang, 2006) and within the last decade it has started to grow very quickly.

Deep Learning (DL) and Artificial Neural Networks (ANNs) have become very popular recently and have been used for many tasks where they have performed very well, even outcoming humans on solving specific problems. However, one of the downsides of these type of neural networks is that they are very energy expensive and have slow inference. This creates a problem when when they are used in hardware where power efficiency and fast run-time is really needed. This is were Spiking Neural Networks (SNNs) enter to stage.

The aim of this Master's Thesis is to bring together all the knowledge and new research about Spiking Neural Networks that has been gathered to date, with a specific focus on Artificial Vision, and make some experiments as examples by using the latest and more updated libraries for SNNs. At the end, a conclusion is made with the results of the experiments and the next possible steps for this field are stated for future researchers to follow.

1.1 Motivation

Artificial Intelligence (AI) is a field which basically aims to give a computer program the ability to learn and to thing for itself. AI has gained popularity over the past years and, generally speaking, there are three main reasons: 1) A huge increase of available data, 2) More effective algorithms, 3) Increase of computing power and the use of GPUs for parallelising mathematical operations.

Machine Learning (ML) is a subset of AI that tries to learn directly from data instead of setting fixed rules for the program to follow, creating in such a way a model of knowledge which is then able to predict an outcome from new incoming data. The three most used learning methodologies for this discipline are:

- **Supervised Learning**: Input and output data is provided
- **Unsupervised Learning**: Only input data is provided
- **Reinforcement Learning**: Environment context as input and feedback from the environment as to determine ideal output

Nowadays, there is a discipline that is continually rising called Deep Learning (DL). DL is a subset of Machine Learning that aims to achieve the same goal as ML but by trying to mimic the human brain. In order to do that, Artificial Neural Networks (ANNs) are developed as a model of the human brain and by using the

same strategy as in ML (learning from data) they are able to obtain a much more complex understanding than a normal ML algorithm would do.

ANNs are said to be similar to the human brain, or at least to be a neural model inspired by human brain. Nevertheless, the human brain is much more efficient than these in every single way and also are able to generalise any task surprisingly well. Not the same can be said about ANNs. These are not efficient at all and the reason why is explained in this work further on.

Spiking Neural Networks (SNNs) come to solve the problem of efficiency, also introducing many improvements such as being able to learn spatio-temporal data. However, there are drawbacks for implementing these type of networks and the main one is that there are not many algorithms that are able to train a SNN.

1.2 Objectives

This work is going to focus on Spiking Neural Networks. Not only they are introduced and analysed in depth, but also some examples are implemented at the end.

The main objectives of this work are:

- **Explain and explore Spiking Neural Networks in depth** SNNs are introduced: explanation of how they work, the existing different models, learning methods and topologies (emphasising in Artificial Vision) and types of conversions of data to spike trains.
- **Implementation using Nengo library:**
  - Introductory implementations using Nengo
  - Develop Deep Learning Spiking Neural Networks using NengoDL and Tensorflow
  - Implement Spiking Neural Networks using native Nengo and Nengo GUI

Chapter 2

Spiking Neural Networks

In this section, Spiking Neural Networks are going to be explained in depth by starting from the very basis. They are compared against the current ANNs, the main models of SNNs, the learning methodologies and the topologies are explained, putting the focus on Artificial Vision. At the end, how neural encoding for Spiking Neural Networks input is done and a brief explanation of what neuromorphic hardware and event-based datasets mean will close this section.

2.1 Introduction

2.1.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are brain based models that are fed with data and are capable of learning very complex functions from that data. These are not only able to complete different types of tasks such as classifications, regression, clustering, etc. but also take very different kind of data as input such as images, sound, words, etc.

ANNs are inspired on the human brain. Figure 2.1 shows how.

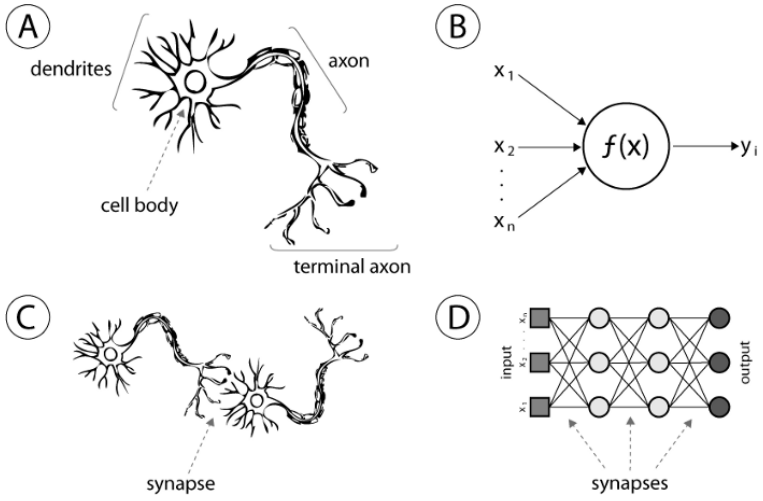


FIGURE 2.1: Human brain structure and ANNs structure. Image downloaded from Medium, 2015

In the first illustration (A), the model of a biological neuron can be seen whereas in the second illustration (B) an artificial neuron is shown. In (A) the dendrites are from where the cell receives data (impulses) and in (B) the Xs are the input data which, in the case of ANNs, are numbers. Then, the core of the neuron (cell body for the biological model) takes these numbers/impulses and performs a calculation

(function), giving an output number. In models (C) and (D), the connection between neurons is presented as the synapse, which is composed of weights that give more or less importance to every input from every neuron depending on whether they help giving the correct answer or not (if an image contains a dog and it is actually true).

In order to update every weight of every branch in an Artificial Neural Network, an algorithm called Back-propagation, Cun, 1988, has to be executed. Every time the ANN predicts the result wrong, a cost/error function is calculated to determine how wrong the answer was and, by using Back-propagation, this error is propagated backwards updating and adjusting every weight in the network towards the correct solution. Once the network is trained (the error is relatively low), it can be used to predict new samples of data.

Nevertheless, ANNs are very inefficient. They are very hard to train, not power efficient and their inference at run-time is very slow and thus, for some real use cases, they are not suitable. This is were Spiking Neural Networks (SNNs) become a very good alternative.

2.1.2 Spiking Neural Networks

In 1952, Alan Lloyd Hodgkinand Andrew Huxley produced the first model of a spiking neuron. Spiking Neural Networks (SNNs) are a much more inspired version of the human brain which mimic the biological neural codes, dynamics, and circuitry. These communicate by sequences of spikes (spikes trains) between neurons, and the presence of these spikes causes the information to be encoded over time, which allows SNNs to process spatio-temporal data. These spikes increase the membrane potential of contiguous neurons, which are fired in case they exceed certain threshold.

The next figure 2.2, shows an example of a Spiking Neuron model and its connections.

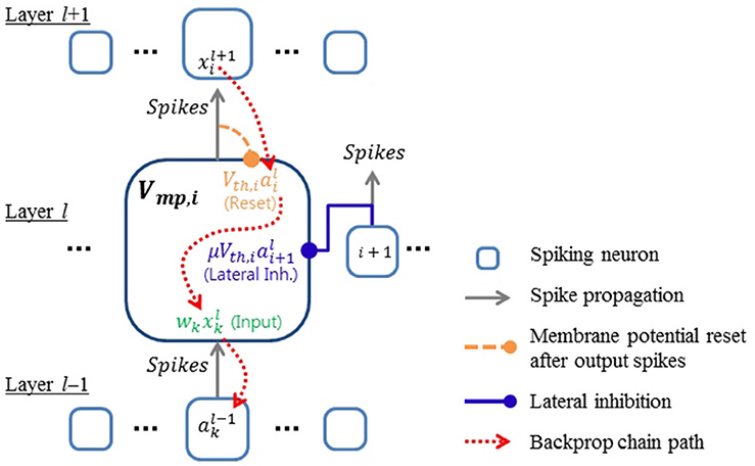


FIGURE 2.2: Spiking neuron model and its connections. Image downloaded from Frontiers in Neuroscience, 2016

There are different kinds of models, but this one has the main parts of a typical Spiking Neuron Model. In the middle of the illustration, it is allocated an example of a neuron. Neurons are typically aligned in layers, which are connected between each other calling these connections synapses. In these synapses is where the spikes flow as spike trains from the previous layer (pre-synaptic neurons) to the actual layer and from there to the next layer (post-synaptic neurons) and so on. The neuron takes

the spike train multiplied by the weight in the synapse as input which increases the membrane potential $V_{mp,i}$. If the membrane potential reaches a threshold then the neuron fires a spike to the post-synaptic neurons and the membrane potential of the neuron resets to a certain value (commonly to 0). Furthermore, if the neuron has fired a spike, there is a connection called lateral inhibition that prevents the rest of the neurons of the same layer to be fired. Once the error is calculated at the end of the network, backpropagation propagates the error backwards throw the layers, updating the synapses on its way.

It is very important to mention that backpropagation does not work as it worked with ANNs. This backpropagation rule has been developed by Frontiers in Neuroscience, 2016 and it is only an approach, there are plenty of other algorithms that try to accomplish this task by using, for example, unsupervised learning, among others. The reason why Artificial Neural Networks are able to implement backpropagation is because their activation functions are continuous and in Spiking Neural Networks due to the complex and discontinuous dynamics of spiking neurons backpropagation cannot be applied.

2.1.3 SNNs vs ANNs

There are several differences between Spiking Neural Networks and Artificial Neural Networks. In this subsection, they are going to be explained briefly and advantages and disadvantages will be stated.

The first difference, is that Spiking Neural Networks are capable of capturing temporal data. While ANNs only handle static data, binary data, such as the pixels within an image, SNNs can also deal with temporal-related factors, as for example spiking rate, spiking rank and spiking intervals, allowing them to increase their processing capacity. All these concepts will be explained later.

ANNs use backpropagation to adjust the weights of the network but they cannot be predicting incoming inputs at the same time. This means that they are not hardware friendly, they cannot learn on the go, while being used, and thus are not able to scale-up. SNNs on the other hand, are capable of learning locally, in-situ, on the go, thanks to the temporal factor of the spikes. Nevertheless, the learning methodologies used at this moment cannot made full use of them. This would be the second main difference.

SNNs only process spikes when they occur, the rest of the time are consuming almost no energy. Instead, ANNs consume a lot of energy which makes them a very less desirable option. In this work Michmizos, 2019 energy efficiency using Spiking Neural Networks is evaluated on the Loihi neuromorphic processor achieving 100 times less energy than in a normal CPU.

They have very high similarity to the human brain. This has been demonstrated by how their neurons work, their connections (synapse), the membrane potential, etc. and also by the learning methodologies that are used for their training, among many other things. This is a step forward to mimic the human brain.

The rapidness of the ANN inference at run-time is not comparable to the SNN inference, which is faster.

SNNs are computationally more powerful than conventional ANNs, as they can be applied to all problems solvable by non-spiking neural networks and more.

In the following table 2.1 these advantages and disadvantages are put together as a summary.

Artificial Neural Networks	Spiking Neural Networks
Many algorithms and frameworks	Few algorithms and frameworks
Good accuracy for static datasets	Not as good accuracy for static datasets
Less complex models	Models more complex
Not able to capture temporal codes	Process temporal data
Offline learning	In-situ learning
Energy inefficient	Very energy efficient
Not as similar to a human brain	Human brain similarity
Computationally less powerful	Computationally more powerful
Slower inference	Faster inference

TABLE 2.1: Comparison between conventional Artificial Neural Networks and Spiking Neural Networks. Red background indicates a disadvantage and green background indicates an advantage

2.2 Models

In this section, Spiking Neural Models will be introduced, as there are many different models. After that, the different learning methodologies and network topologies will be explained.

As it has been explained before, SNNs are strongly based in biological neural models, they try to be as similar as possible to them. There are different Spiking Neural Models that try to mimic real neural models as biologically plausible as possible. They differ in how well they reproduce biological model firing patterns and their computation efficiency.

2.2.1 Hodgkin-Huxley

The Hodgkin-Husley (HH) model, created by A.L Hodgkin and A.F. Huxley, is an important computational model for the neuroscience field because it models all physical aspects of a biological neuron. Due to its mathematical complexity it is impossible to create a Spiking Neural Network from it, but the essential concept is that it describes the dynamics of a biological neuron as a circuit with currents which are activated and deactivated at different timescales and at different voltages. Most artificial spiking neural models are based in the Hodgkin-Husley model.

In the following figure 2.3, the basic components of the HH model are shown.

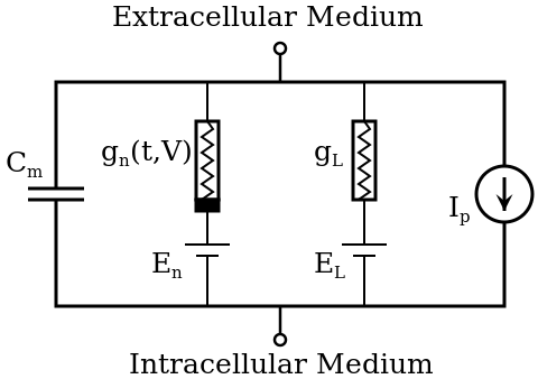


FIGURE 2.3: Hodgkin-Huxley model's basic components. Image downloaded from *Hodgkin–Huxley model*

There are three currents that take place in this model: I_{leak}, I_{Na} and I_K. These influence the membrane potential in different ways:

- I_{leak} is not gated and thus Chloride ions slowly leak across the membrane
- I_K is gated by $n[0,1]$ and it represents the activation of Potassium ion voltage gated channels (increasing the flow of potassium)
- I_{Na} current is gated by $m[0,1]$ which represents the activation of Sodium ion voltage gated channels (increasing the flow of sodium) and $h[0,1]$ which represents the inactivation of Sodium ion channels (stopping flow of sodium)

These three currents are represented in the following formula 2.1.

$$Cm\frac{dV}{dt} = \overbrace{-g_L(V-E_L)}^{I_{leak}} - \overbrace{g_{Na}m^3h(V-E_{Na})}^{I_{Na}} - \overbrace{g_Kn(V-E_K)}^{I_K} \tag{2.1}$$

Cm: membrane capacitance, V: membrane potential, E: membrane equilibrium potential, g: membrane conductance ($\frac{1}{R}$), (n, m, h): dimensionless quantities between 0 and 1 associated with potassium channel activation, sodium channel activation, and sodium channel inactivation.

m, h and n vary over time. They have their own formula that corresponds to the derivative over time 2.2.

$$\begin{aligned} \frac{dn}{dt} &= \alpha_n(V_m)(1-n) - \beta_n(V_m)n \\ \frac{dm}{dt} &= \alpha_m(V_m)(1-m) - \beta_m(V_m)m \\ \frac{dh}{dt} &= \alpha_h(V_m)(1-h) - \beta_h(V_m)h \end{aligned} \tag{2.2}$$

α_i, β_i: constants that model the rate at which the i-th ion channel gate begins to open and to close, respectively, and depend on the voltage.

For $p = (n,m,h)$, α_p and β_p take the form of 2.3.

$$\begin{aligned} \alpha_p(V_m) &= \frac{p_\infty(V_m)}{\tau_p} \\ \beta_p(V_m) &= \frac{(1-p_\infty(V_m))}{\tau_p} \end{aligned} \tag{2.3}$$

p_∞ and $(1-p_\infty)$ are the steady state values for activation and inactivation, respectively, and are usually represented by Boltzmann equations as functions of V_m.

The Hodgkin-Huxley model is the most accepted model. It is also the most complex model that will be considered in this thesis. The following models are more abstract and will lack some features of the Hodgkin-Huxley model, however they are more easy to analyse and simulate.

2.2.2 Integrate and Fire

Integrate and Fire Model (IF), by Lapicque in 1907, is one of the simplest models and it is based on the Hodgkin-Huxley Model. Its simplicity makes it one of the best

candidate for research exploration. The model works as follows: the membrane of the neuron has currents flowing into it and when the membrane potential reaches a certain threshold it fires a spike and the membrane potential is reset to zero. That is one feature from biological neural behaviour, but there are two more: there is a refractory period during which the neuron cannot fire and there is a state which represents the membrane potential evolving over time. The mathematical equations employed here are 2.4 and 2.5.

$$C_m \frac{dV}{dt} = \frac{V - E}{R} + I \tag{2.4}$$

$$\text{if} \quad V \geq V_{threshold} \quad \text{then} \quad V = E \tag{2.5}$$

V: membrane potential, Cm: membrane capacitance, E: membrane equilibrium potential, R: membrane resistance, I: the total currents flowing into the neuron cell.

Nevertheless, these models lack of a couple of features that biological ones do implement: the membrane threshold and refractory period do not change depending on the state of the neuron, they are absolute, and they are not capable of resonating (tendency for certain neural firing patterns to persist due to how their frequency related properties interact with the features of the brain and each other).

There are many improved I&F models: I&F with adaptation, Integrate and Fire or Burst, Resonate and Fire, Quadratic IF and IF-FHN, each one of them specialised on a specific neuron behaviour.

The most famous I&F model is called Leaky Integrate and Fire. Leaky refers to that the accumulator of the membrane potential decreases exponentially over time, which is more biologically realistic.

2.2.3 Spike Response Model

The Spike Response Model (SRM), Renaud Jolivet and Gerstner, 2003, is a generalisation of the Leaky Integrate and Fire Model. In contrast, the SRM includes a refractory period that limits the firing frequency of a neuron by preventing it from firing during that period. Moreover, I&F models are formulated using differential equations for the voltage, whereas the Spike Response Model is formulated using kernel functions (κ and η). These functions describe the effect of spike reception and emission on the membrane potential of the neuron.

The equations that represent this model are as in the equation 2.6

$$V_i(t) = \eta(t - t_i) + \int_{-\infty}^{\infty} \kappa(t - t_i, s) I^{ext}(t - s) ds \tag{2.6}$$

t is the firing time of the last spike of the neuron, n describes the form of the action potential (when the membrane potential reaches the spike threshold) and its spike after potential, k the linear response to an input pulse (the neuron in which the membrane potential is being calculated) and $I^{ext}(t)$ a stimulating external current.

In the second equation 2.7, if the membrane potential reaches certain threshold from below then t is updated.

$$\text{if} \quad V_i(t) \geq \theta \quad \text{and} \quad V_j(t) > 0, \quad \text{then} \quad t_i = t \tag{2.7}$$

The name of Spike Response stems from the fact that k describes the Response of the neuron to an incoming spike and n describes the Response of the membrane to its own spike.

2.2.4 Izhikevich

The Izhikevich Model, Izhikevich, 2003, is a simplified HH model biologically plausible and computationally efficient. This model is capable of reproducing spiking and bursting behaviour of known types of cortical neurons, Hippocampus neurons can be described by this mode and it is more suitable for large-scale simulations. It is represented by two differential equations 2.8 and 2.9.

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{2.8}$$

$$\frac{du}{dt} = a(bv - u) \tag{2.9}$$

This third equation 2.10 is used to adjust the membrane voltage v and the recovery variable u.

$$\text{if} \quad v \geq 30mV \quad \text{then} \quad \begin{cases} v = c \\ u = u + d \end{cases} \tag{2.10}$$

v represents the membrane potential and u is the recovery variable that describes the activation of Potassium ionic currents and the inactivation of Sodium ionic currents, and it provides negative feedback to v. I would be synaptic currents or injected dc-currents. a, b, c and d are just dimensionless parameters.

Izhikevich Model is widely used in benchmarking and simulations because it exhibits all neuron behaviours.

2.3 Learning

Learning in Neural Networks is the process of acquiring new knowledge and strengthening knowledge that you already have by following an iterative loop. In biology, this is represented by the connections between brain cells, called synapses, which can be created, modified or erased. It is the synapse plasticity that regulates this process depending on their activity. The changes on the connections have two types depending on the duration:

- Short Term Plasticity: Short-term synaptic plasticity acts on a timescale of tens of milliseconds to a few minutes unlike long-term plasticity, which lasts from minutes to hours. Short term plasticity can either strengthen (Synaptic enhancement) or weaken (Synaptic depression) a synapse.
- Long Term Plasticity: Long-term depression (LTD) and long-term potentiation (LTP) are two forms of long-term plasticity, lasting minutes or more, that occur at excitatory synapses.

2.3.1 Hebbian Rule

The Hebbian Rule (Donald Hebb, 1949) stipulates that if two neighbour neurons activate and deactivate at the same time, then the weight of the synapse connecting

these neurons should be enhanced. For neurons operating in the opposite phase, the weight between them should decrease. If there is no signal correlation, the weight should not change. The famous quote by Donald Hebb "Neurons that fire together, wire together" states it, and the following formula verifies it 2.11.

$$w_{ij} = x_i * x_j \tag{2.11}$$

Where w_{ij} is the weight of the connection from the pre-synaptic neuron j to the post-synaptic neuron i and x_i or x_j the input for each neuron. This is pattern learning (weights updated after every training example). For the methodology learning by epoch the following formula is stated 2.12.

$$w_{ij} = \frac{1}{p} \sum_{k=1}^{p} x_i^k * x_j^k \tag{2.12}$$

Where w_{ij} is the weight of the connection from the pre-synaptic neuron j to the post-synaptic neuron i, p is the number of training patterns, and x_i^k or x_j^k the kth input for each neuron.

There are many rules that derive from the Hebbian Rule and all of them share the same properties:

- Time-dependence: The synaptic weight change depends on the exact fire time of pre- and post-synaptic neuron
- Locality: the synaptic efficacy variation derives from some local variables (like pre- and post-synaptic activity and synaptic weight)
- Interactivity: the magnitude of the change depends on the activity of the two cells

2.3.2 Spike-timing dependent plasticity (STDP)

According to the Hebbian rule, synapses increase their efficiency if the synapse persistently takes part in firing the post-synaptic target neuron. However, if two neurons fire exactly at the same time, then one cannot have caused, or taken part in firing the other. This is where Spike-timing Dependent Plasticity (STDP) comes to place.

STDP states that to take part in firing the post-synaptic neuron, the pre-synaptic neuron needs to fire, on average, just before the postsynaptic neuron, and not at the same time nor long time before. If this occurs, the input is made somewhat stronger. But if an input spike tends, on average, to occur immediately after an output spike, then that particular input is made somewhat weaker. This way, neurons that might have caused a post-synaptic neuron to fire will probably do so in the future and vice versa.

The basic STDP model states that the total weight change Δw_j of the pre-synaptic neuron is calculated as in the formula 2.13.

$$\Delta w_j = \sum_{f=1}^{N} \sum_{n=1}^{N} W(t_i^n - t_j^f) \tag{2.13}$$

Where t_j^f is the presynaptic spike arrival times at pre-synapse j, being f a counter of the pre-synaptic spikes, and where $W(x)$ denotes one of the STDP functions (also called learning window). This learning window is well illustrated in the following figure 2.4

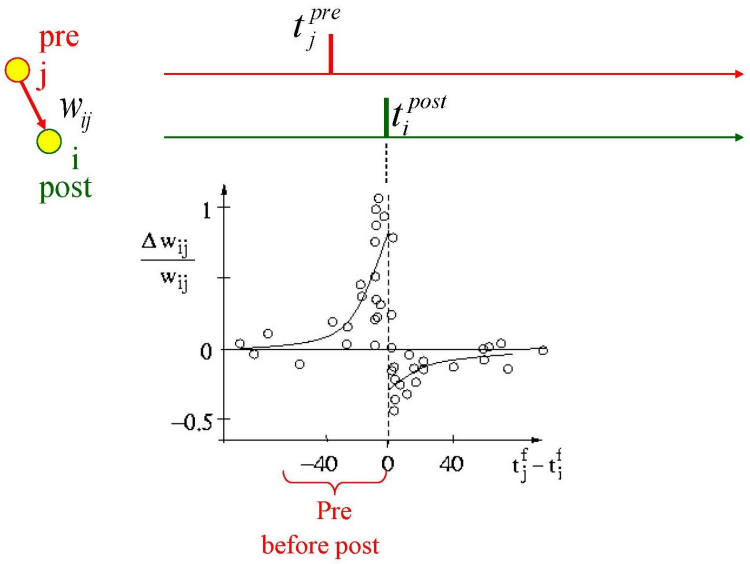


FIGURE 2.4: Asymetric Spike-Timing Dependent Plasticity's learning window. Image downloaded from *Spike-timing dependent plasticity*

The closer it is the time difference between the pre-synaptic spike and the post-synaptic spike the exponentially greater is the update of the synaptic weight, being either positive, long-term potentiation (LTP), if the pre-synaptic spike occurs before the post-synaptic spike or negative, long-term depression (LTD), otherwise.

The equation for the $W(x)$ that represents the figure 2.4 is written as in equation 2.14.

$$\begin{cases} W(x) = A_+ exp(-x/\tau_+) & \text{if} \quad x > 0 \\ W(x) = -A_- exp(x/\tau_-) & \text{if} \quad x < 0 \end{cases} \tag{2.14}$$

Where A_+ and A_- determine the maximum amounts of synaptic modification and the parameters τ_+ and τ_- determine the ranges of pre–to–post synaptic inter-spike intervals over which synaptic change occur.

2.3.3 Unsupervised Learning

Unsupervised Learning is a learning methodology where the input dataset has no labels at all. Thus, generally speaking, this approach tries to create clusters or subgroups from the data where the data within those clusters will have features in common.

Unsupervised Learning in Spiking Neural Networks often involves STDP as part of the learning mechanism.

Nevertheless, because of its primary dependency on the local neuronal activities without global supervisor, these cannot get a really high performance.

2.3.4 Supervised Learning

Supervised Learning, on the contrary, aims to learn a function that maps an input to an output based on example input-output pairs. This means that this methodology is well suited when the data is labelled. If these labels are not well predicted by the

model, then a cost function will determine how wrong the algorithm was and adjust its parameters (neuron's weights in case of a neural networks) in order to predict it right the next time.

Backpropagation Issues

There has been issues regarding the training of Spiking Neural Networks using Supervised Learning. The two main problems can be extracted from the following formula 2.15.

$$\delta_j^\mu = g'(a_j^\mu) \sum_k w_{kj} \delta_k^\mu \tag{2.15}$$

This is the core formula that occurs in all variants of backpropagation, which is being highly used by Supervised Learning approaches. In the formula, δ_j^μ and δ_k^μ act as the partial derivative of the cost function for input pattern μ with respect to the net input to some arbitrary unit j or k. j projects direct feedforward connections to the set of units indexed by k. $g()$ is the activation function to the net input of unit j, written as a_j^μ. w_{kj} are the feedforward weights projecting from unit j to the set of units indexed by k.

Both parts of the formula 2.15 denote problems regarding the bio-plausibility of Spiking Neural Networks when using backpropagation.

The first main problem is that $g'()$ needs $g()$ with respect to w_{kj}, and because $g()$ applies to a spiking neuron, it is probably calculated as a sum of Dirac delta functions and thus there is no derivative.

The second main problem applies to all networks, not only spike ones. The summatory in the formula 2.15 is using the feedforward weights in a feedback fashion, which means that symmetric feedback weights must exist and project accurately to the correct neurons so the equation is useable.

The first problem is theoretically solved by using substitute or approximate derivatives, whereas the second problem has gain some progress by stating that some tasks could still use backpropagation if random feedback weights were used, although more complex problems needed symmetric feedback.

Some supervised learning methods for SNNs

There are two types of Supervised Learning methods for SNNs:

- Indirect Supervised Learning (Conversion): This methodology involves training first an Artificial Neural Network (ANN) and then it is transform into a Spiking Neural Network with the same network structure where the rate of SNN neurons acts as the analog activity of ANN neurons.
- Direct Supervised Learning: This methodology focuses on training Spiking Neural Networks on their own, by trying to mimic the backpropagation process from conventional ANNs.

In **Indirect Supervised Learning**, to substitute for the floating-point activation values in ANNs, rate-based coding is generally used in which higher activations are replaced by higher spike rates. For ANN to SNN conversion, the well known activation function ReLU is tipically used when training the ANN. This is because they work apparently equivalent to Integrate-Fire Spiking Neurons, without any leak

and refractory period. For a neuron receiving inputs x_i through synaptic weights w_i, the ReLU neuron output y is given by the formula 2.16.

$$y = max(0, \sum_i w_i \cdot x_i) \tag{2.16}$$

Let us consider the ANN inputs x_i encoded in time as a spike train $\mathbb{X}(t)$, where the average value of $\mathbb{X}(t)$, $E[\mathbb{X}(t)] \propto x_i$. The IF Spiking Neuron keeps track of its membrane potential, V, which integrates incoming spikes and generates an output spike whenever the membrane potential cross a particular threshold $V_{threshold}$. The membrane potential is reset to zero at the generation of an output spike. All neurons are reset whenever a spike train corresponding to a new image/pattern in presented. The IF Spiking Neuron dynamics as a function of time-step, t, can be described by the following equation 2.17.

$$V(t+1) = V(t) + \sum_i w_i \cdot \mathbb{X}(t) \tag{2.17}$$

The higher the ratio of the threshold with respect to the weight, the more time is required for the neuron to spike, thereby reducing the neuron spiking rate, $E[Y(t)]$, or equivalently increasing the time-delay for the neuron to generate a spike. A relatively high firing threshold can cause a huge delay for neurons to generate output spikes. For deep architectures, such a delay can quickly accumulate and cause the network to not produce any spiking outputs for relatively long periods of time. On the other hand, a relatively low threshold causes the SNN to lose any ability to distinguish between different magnitudes of the spike inputs being accumulated to the membrane potential ($\sum_i w_i \cdot \mathbb{X}(t)$ from 2.17) of the Spiking Neuron, causing it to lose evidence during the membrane potential integration process. This, in turn, results in accuracy degradation of the converted network.

Consequently, most of the research work in this field has been concentrated on outlining appropriate algorithms for threshold-balancing, or equivalently, weight normalising different layers of a network to achieve near-lossless ANN-SNN conversion.

Direct Supervised Learning, on the other hand, is based on backpropagation and gradient descent although, as it has already been said, spiking neurons cannot be derived due to its discrete nature of spikes.

SpikeProp (Neurocomputing, 2002) to be the first algorithm to train SNNs by using backpropagation. Their cost function took into account spike timing and SpikeProp was able to classify non-linearly separable data for a temporally encoded XOR problem using a 3-layer architecture. The spike model the used was the Spike Response Model (SRM). Using the SRM model, the issue of taking derivatives on the output spikes of the hidden units was avoided because those units' responses could be directly modeled as continuous-valued applying to the output synapses that they projected to. However, this algorithm had its downsides

- Each output unit was constrained to discharge exactly one spike
- Continuous variable values had to be encoded as spike-time delays which could be quite long
- The synaptic weight is updated only if the post-synaptic neuron spikes, so if this cell never fires the junction weight (efficacy) never change. Thus, the accuracy of the networks train with this algorithm has a strong dependency to initial weights

Later advanced versions of SpikeProp, MultiSpikeProp, were applicable in multiple spike coding. Arxiv, 2016 by Jun Haen Lee, introduced a novel technique which treats the membrane potentials of spiking neurons as differentiable signals, where discontinuities at spike times are only considered as noise, enabling an error backpropagation mechanism. The most recent implementation of backpropagation in SNNs has been proposed by Yujie Wu Arxiv, 2017 who developed spatio-temporal gradient descent in multi-layer SNNs.

In Arxiv, 2018, a supervised learning method was proposed (BP-STDP) where the backpropagation update rules were converted to temporally local STDP rules for multilayer SNNs. This model achieved accuracies comparable to equal-sized conventional and spiking networks for the Modified National Institute of Standards and Technology (MNIST) dataset *The MNIST database of handwritten digits*.

Deep Spiking Neural Networks and specially Deep Convolutional Spiking Neural Networks will be covered in more detail below.

2.3.5 Reinforcement Learning

There are many studies that suggest the brain's reward system holds a very important part in making decisions and forming behaviours. Reinforcement Learning (RL) stands as learning by trial and error, where an agent is exploring virtual environments and rewards or punishments (negative rewards) are given to that agent depending on whether the agent is making actions that are considered right ones for that specific environment or wrong ones, respectively.

One way to adapt the learning of an agent in Reinforcement Learning for biological purposes, it is to modulate or even reverse the weight change calculated by STDP ,called Reward STDP (R-STDP) (PubMed, 2007). RSTDP stores the trace of synapses that are eligible for STDP and applies the modulated weight changes at the time of receiving a modulatory signal. Izhikevich proposed a R-STDP rule to solve the distal reward problem, where the reward is not immediately received.

Nonetheless, R-STDP is shown to be problematic since for each stimulus class the expected reward must be 0. This is due to the fact that the integral over the STDP curve (2.4) tends to deviate from and thus learning with $R \neq 0$ would cause a weight drift.

2.4 Deep Learning in Spiking Neural Networks

Deep Learning has shown very good results in research and applied scenarios. Deep Neural Networks (DNNs) have been around for many years, but not until a few have been useful. This is because now there is better hardware (GPUs) and much more data. This way, Deep Neural Networks are capable of gaining a very deep understanding of the task at issue.

Deep Neural Networks are somehow inspired in the human brain by having stacked layers with interconnected neurons that become deeper as more layers are added sequentially. DNNs extract complex features through sequential layers of neurons equipped by nonlinear, differentiable activation functions to provide an appropriate platform for the back-propagation algorithm. Back-propagation is used for calculating the loss of the neurons within the network and gradient descent would be the algorithm for updating correctly the weight of those neurons (towards the global minimum).

There are several types of Deep Neural Networks: Fully Connected Neural Networks (FCNN), Deep Convolutional Neural Networks (DCNN), Deep Belief Networks (DBN) and Recurrent Neural Networks (RNN).

Fully Connected Neural Networks (FCNN) consist of a series of fully connected layers. Each output dimension depends on each input dimension. Figure 2.5 shows a standard Fully Connected Neural Network architecture with input, hidden, and output layers.

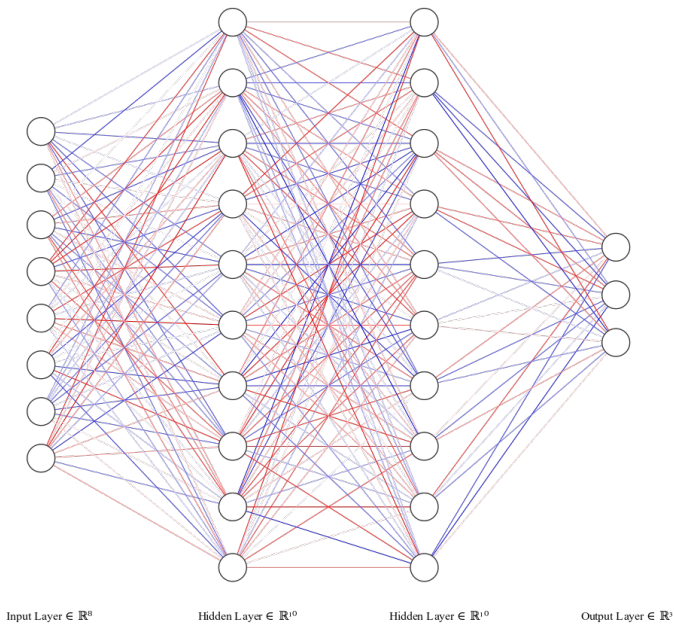


FIGURE 2.5: Fully Connected Neural Network. Created using *Publication-ready NN-architecture schematics*

In a FCNN, information arrives to the first layer (the input layer) where the network learns how to pre-process the data or to extract the more general features of the data. The hidden layers, which can vary in quantity and number of neurons, are capable of extracting the more complex features out of the data. Towards the end of the architecture the output layer is found. This layer is in charge of performing the classification task in order to match properly every input data to the correct label.

Deep Convolutional Neural Networks (DCNN) consist of a series of convolutional layers interspersed with max pooling layers, and fully connected layers at the end. The following figure 2.6 shows the architecture of a DCNN.

Convolutional layers apply a series of mathematical operations to the input (dot product) aiming to learn features in a much efficient way than FCNNs do. DCNNs are able to successfully capture the Spatial and Temporal dependencies in an image (or any other similar type of data such as audio waves) through the application of relevant filters. If we take a RGB image as an example, where every channel is a matrix of pixel values, it would be very computationally inefficient to flatten the matrices and to train a Fully Connected Neural Network.

In order to learn spatial features in an image, a DCNN has to apply several filters to it. Figure 2.7 shows how a convolution is performed.

These filters are capable of learning features by applying mathematical operations to the image. First layers learn more general features and as deeper the neural network becomes more specific characteristics are captured. Here the Pooling layer

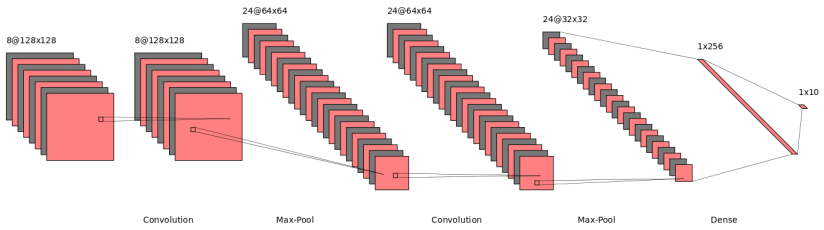


FIGURE 2.6: Deep Convolutional Neural Network. Created using *Publication-ready NN-architecture schematics*

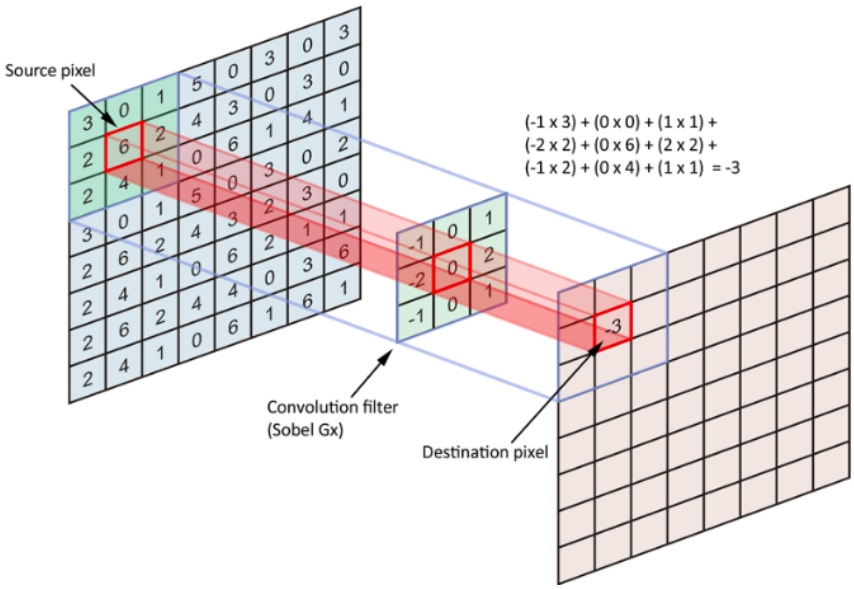


FIGURE 2.7: Process of convolution in a Convolutional Neural Network. Retrieved from *freeCodeCamp*

takes an important part: its main purpose is to reduce the spatial size of the input data and allowing for assumptions to be made about features contained in the sub-regions binned, that is how deeper filters learn more specific features.

Deep Belief Networks and Recurrent Neural Networks are not going to be explained in this work to keep it briefer.

2.4.1 Deep Fully Connected SNNs

Typically, a basic and simple Fully Connected Spiking Neural Network architecture would look like the one in the Figure 2.8.

There is always a first spike conversion step (the different types of conversions will be explained down below) in any SNN. Learning neurons are placed after conversion and in this case, as it is a FCSNN, only Dense layers are applied. On the right side of the figure, an coding neuron model can be seen, which its entire purpose is to convert the data to spike based information.

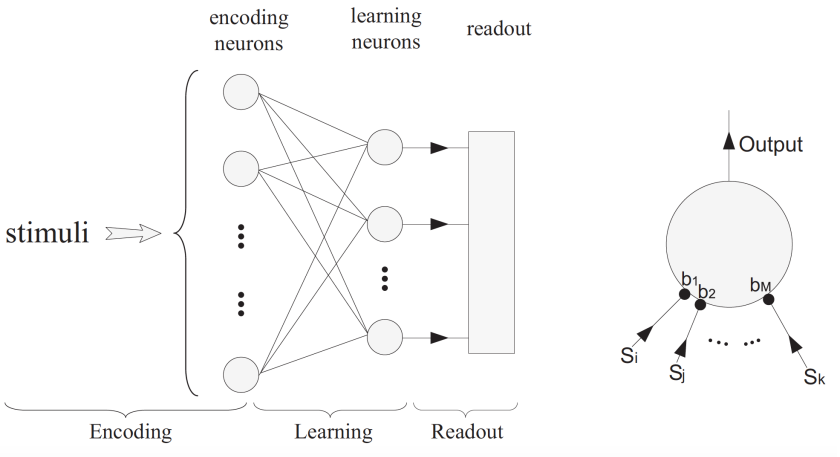


FIGURE 2.8: Fully Connected Spiking Neural Network. Downloaded from the paper *A brain-inspired spiking neural network model with temporal encoding and learning*

There is no much difference from the original Fully Connected Neural Networks in terms of architecture, but internally the neurons interact between each other very differently: organic connections are created or destroyed depending on how frequently the are used and there can be connections between neurons from within a population and between populations. Different types of neurons have been shown and explained and combining those with different learning rules and approaches such as Unsupervised Learning, Supervised Learning using backpropagation, conversion from ANN to SNN and so on.

Fully Connected Spiking Neural Network started being implemented using STDP and Stochastic Gradient Descent. In 2015, Diehl and Cook, by using Unsupervised Learning, demonstrated that extracting important features and patterns from stimuli was plausible by using STDP in a two layer fully connected SNN, achieving 95% of accuracy in the MNIST dataset *Unsupervised learning of digit recognition using spike-timing-dependent plasticity*.

A tendency to implement biologically more plausible Spiking Neural Networks led Jun Haeng Lee to propose a backpropagation algorithm where the membrane potential was treated as a differentiable signal trying to mimic the continuous non-linear activation functions in ANNs in 2016 *Training Deep Spiking Neural Networks using Backpropagation*, achieving 98,88% of accuracy on the MNIST and also being much more computationally efficient than conventional ANNs. Figure 2.9 shows the backpropagation technique applied by Lee in that paper.

In Figure 2.9, a_l, i, neuron's activation value, is determined by taking into account the neuron's membrane potential and it is used as input for the next layer in the backpropagation algorithm. In order to make backpropagation work, the neuron's input, the lateral inhibition and the threshold are used to calculate the activation function in order to be differentiable and then the chain rule is applied.

As it is has also been said, another way to implement Fully Connected SNNs is to convert it from an offline ANN. The main reason to do this conversion is to implement these models in hardware in an efficient way. These conversions are based on different kinds of spike codings as floating-point values have to be encoded as spikes. However, this topic will be discussed later in the Spike Coding section.

Due the fact that this work is more focused on Spiking Neural Networks applied to artificial vision, Deep Convolutional Spiking Neural Networks will be explained more

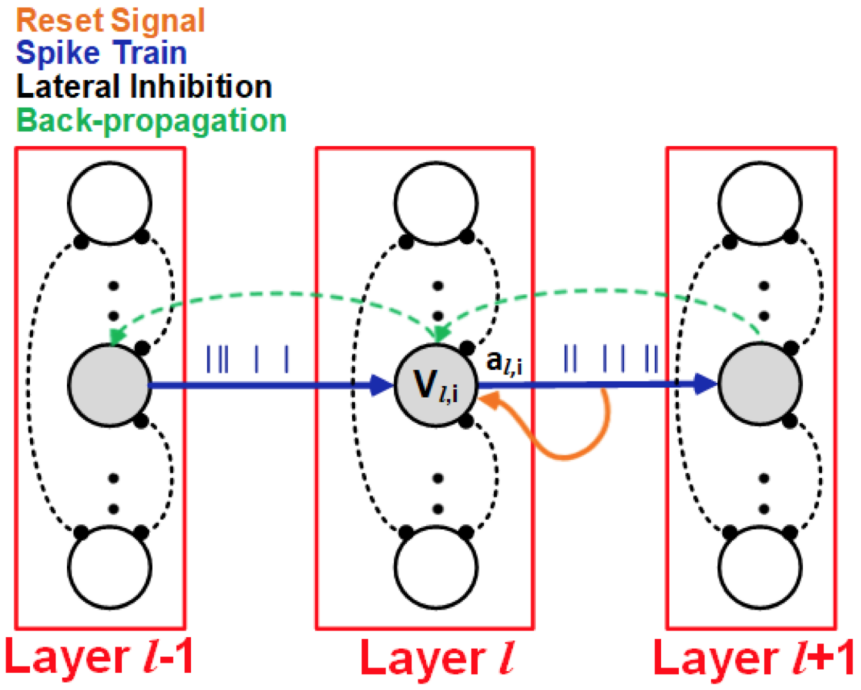


FIGURE 2.9: Fully Connected SNN with backpropagation. Downloaded from the paper *Deep Learning in Spiking Neural Networks*

in depth in the next section.

2.5 Deep Convolutional Spiking Neural Networks

The main purpose of using Deep Convolutional Spiking Neural Networks is to process image data rather than anything else. However, these have been used on types of data very different from images, such as sound or even text.

The topology of these kind of networks usually consist of stacked sequences of repeated convolutional layers followed by pooling layers and connected at the end with dense layers or any feed-forward classifier. These type of networks have shown outstanding performance in all kind of tasks and are considered to be the state of the art in this domain.

The main question here is how a Convolutional Spiking Neural Network can be trained while preserving CNNs feature extraction properties. It is known that the primary visual cortex, which is the first cortical area in the visual hierarchy of the primate's brain, is able to extract general visual properties in the area of imagery and thus that is what the first layers of a CNN do. Subsequent layers are able to extract features and attributes of particular visual content and later layers will continue to extract more detailed properties.

One way to combine CNNs with Spikes in order to be able to build Deep Convolutional Spiking Neural Networks is to use Representation Filters and STDP learning rules. There are two types:

- Trained: Representation Learning Layers are filters that learn to encode input data into spike trains. In the figure 2.10 Representation learning for layer-wise unsupervised learning of a spiking CNN has been utilised.

- Hand-Crafted: Hand-crafted convolutional kernels have been used in the first layer of a number of spiking CNNs that have obtained high classification performance. The most common choice is to use the Difference-of-Gaussian (DoG) filter to extract features in the first layers. An example of this layer is shown in the following figure 2.11.

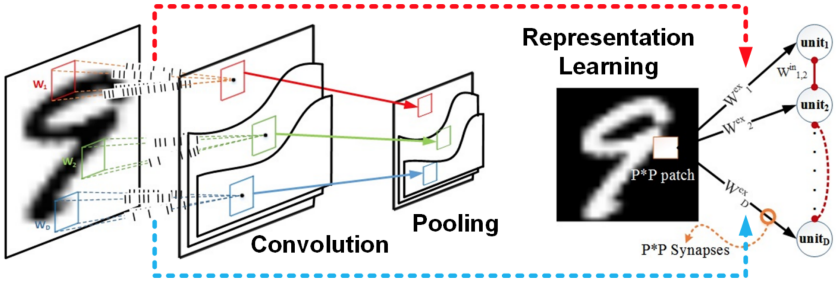


FIGURE 2.10: Representation Learning for layer-wise unsupervised learning of a spiking CNN (*Multi-layer unsupervised learning in a spiking convolutional neural network*)

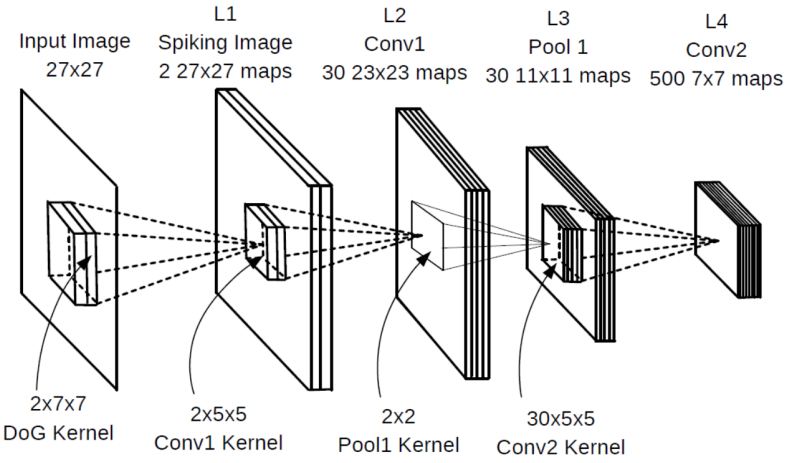


FIGURE 2.11: Usage of Difference-of-Gaussian (DoG) filter (*Deep Convolutional Spiking Neural Networks for Image Classification*)

Convolutional Neural Networks use backpropagation in order to adjust the weights of the network in a way that tries to minimise the loss function. Convolutional SNNs now too.

As it has been explained, Jun Haeng Lee already proposed a backpropagation algorithm based on a differentiable membrane potential that worked on Fully Connected Neural Networks, but it also does on CSNNs. He achieved an accuracy percentage of 99.31% on the MNIST dataset using Convolutional SNNs.

Spiking Convolutional Auto-Encoders have also been trained using backpropagation *Unsupervised Regenerative Learning of Hierarchical Features in Spiking Deep Networks for Object Recognition* where they used local, layer-wise learning of convolutional layers but also using the membrane potential as replacements for differentiable activation functions, although it did not achieved the same performance as Lee et. al.

Another approach for building Deep Convolutional Spiking Neural Networks is to convert an existing trained CNN to a DCSNN architecture by using the trained synaptic weights, similar to the ANN-to-SNN conversion method. This methodology has brought high performance DCSNN in task where normal CNNs work well but with fewer operations and less consume of energy. This has unlocked CNNs to be implemented on hardware efficiently. In the following Figure 2.12, a CSNN can be seen, where the layers of the SNN use the weights of the CNN where it has been trained. It can be noted that at the beginning of the network, the image is converted to spike trains based on the pixel intensity.

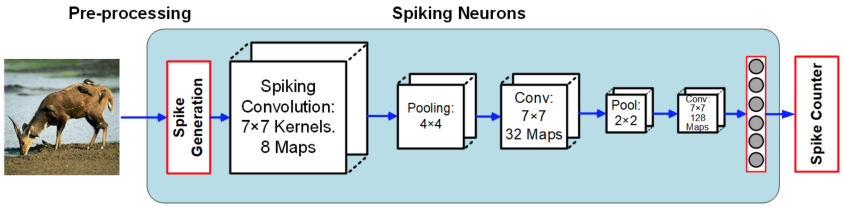


FIGURE 2.12: Convolutional SNN architecture converted from a normal CNN (*Spiking Deep Convolutional Neural Networks for Energy-Efficient Object Recognition*)

2.6 Spike Coding

Conventional Neural Networks work with analogue quantities, they receive analogue inputs and return analogue outputs. As opposite to these, Spiking Neural Networks internally operate using spike trains (discrete values).

One of the main purpose of using SNNs is because they are energy efficient. As a result, they are a good fit for problems that involve real-world live interactions, thus efficient inference on hardware is a must and that is where SNNs are good at. But, how is the data perceived from the world?

Conventional ANNs have been used for many purposes and many type of datasets have been employed in order to train them. If the main objective would be artificial vision, for example, raw static images would be the kind of data used for training these type of networks, getting very good results. Spiking Neural Networks have not yet achieved the same performance, they perform using data with a temporal variable and thus static images have to be encoded to incorporate this. Also, because they use discrete values, new algorithms for training had to be created and yet there is no consensus on which one is better.

Spiking Neural Networks were thought to be trained using data that depends on time, and that is why it is not fair to compare them against conventional ANNs using static datasets. Instead, neuromorphic datasets are the ones used for training SNNs which are obtained using neuromorphic hardware. In the scope of images, these datasets contain mainly the information that varies over time between frames, meaning less information is captured and hence much more efficient processes can be performed. This is where Spiking Neural Networks take the lead. Neuromorphic hardware and neuromorphic datasets (or event-based datasets) will be explained later.

The general approach for the Spiking Neural Networks to work with analogue data but internally still operate with spike trains, is to **Encode** the data that is being received into spike trains and **Decode** the output data to analogue data.

An example of spike trains is shown in figure 2.13.

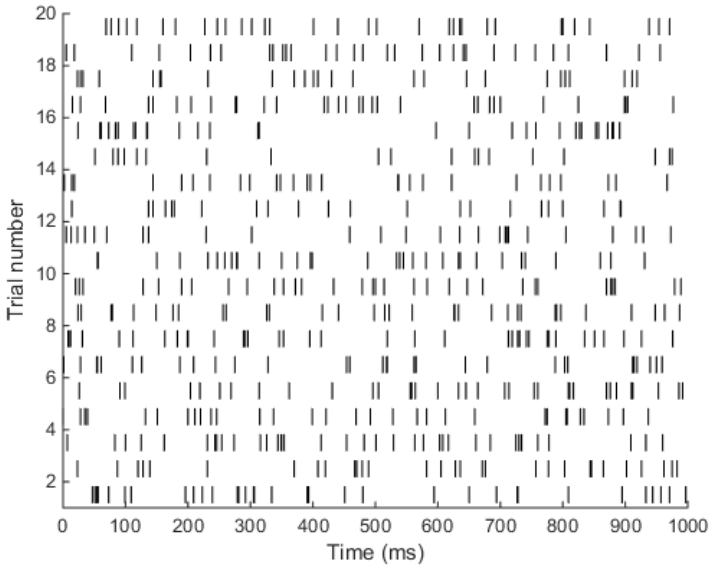


FIGURE 2.13: Spike Trains. Image downloaded from *Simulating neural spike trains*

Where each line corresponds to a neuron being fired (the number of the neuron is represented by the y axis) at a specific time in ms (x axis).

In the following figure 2.14, the process of Coding continuous data into Spike Trains (discrete values) as well as the inverse procedure (Decoding), where discrete values are converted to continuous data, is shown.

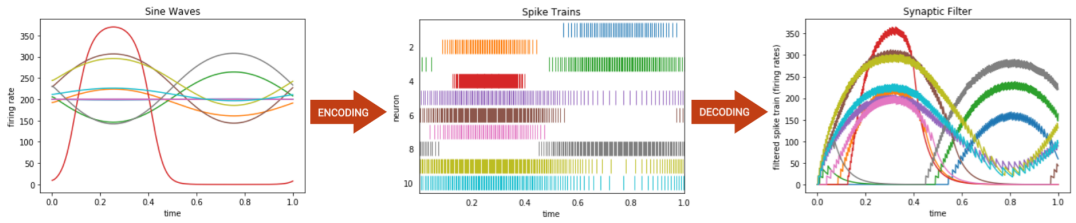


FIGURE 2.14: Spike Trains Encoding and Decoding. Created using the framework **nengodl**

In the following subsection, different types of Coding procedures will be explained.

2.6.1 Neural Coding

Neural coding is a neuroscience field concerned with the mechanisms by which the brain encodes sensory information, such as vision, hearing, taste, etc. The data that is sent to the brain is called stimulus and the purpose of Neural Encoding is to represent this information to the neurons of the brain in a specific way. It is said that there are neurons in the brain called sensory neurons which are able to encode any external sensory stimuli into action potentials, which are later processed by the rest of the brain. The sequence of action potentials is called spike train. Action potentials can vary in many ways, but scientist have treated them as identical in neural coding studies.

Depending on the type of stimuli, different coding schemes could be utilised. The main two Coding schemes in neuroscience are **Rate Coding** and **Temporal Coding**. Rate Coding is based on the firing rate, the average number of spikes per unit time, and Temporal Coding is based on the precise timing of single spikes. These both concepts are widely discussed in the neuroscience community and hence they and their variants will be explained in this work.

Rate Coding

Rate Coding, also called Frequency Coding, states that as higher the intensity or value of a stimulus is, the rate of action potentials increases as well.

This technique assumes that all the information of a stimulus is represented in the firing rate and thus temporal encoded data is ignored.

There are some variants of this scheme regarding to different averaging procedures, some of which will be explained down below.

Rate Coding has been treated as the standard methodology for coding stimuli because of its ease of use and capability to be analysed for further improvement of the research. However, its simplicity has recently raised a common concern because it does not fully represent brain activity, it is not even close of doing so.

Spike-Count Rate is one variant of Rate Coding and it is based on counting the number of spikes during a trial divided by the duration of that trial. It is also called Temporal Average. The time of the trial T is settable by a person and it is commonly selected between the range of $100 > T > 500$ ms. This procedure works well when stimulus is constant or slowly varying, meaning it does not support very fast changes of stimuli. Nevertheless, there are some real-world scenarios where the environment changes fast.

Figure 2.15 shows how Spike-Count Rate works: the mean of the firing rate by temporal average.

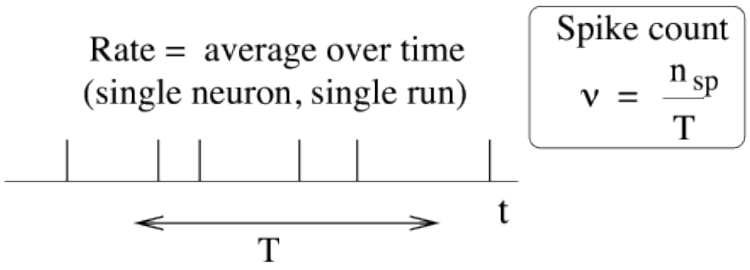


FIGURE 2.15: Spike-Count Rate. Downloaded from *Variability of Spike Trains and Neural Codes*

Time-Dependent Firing Rate is another variant of Rate Coding and it states that the firing rate should be defined by the average number of spikes (averaged over trials) during a short interval between times t and $t + \Delta t$, divided by the duration of the interval. The time t is measured with respect to the start of the stimulation sequence. Δt must be large enough so there are sufficient number of spikes to estimate the average properly (range of 1 to few milliseconds). It works for stationary as well as for time-dependent stimuli.

Formula 2.18 represents mathematically the Time-Dependent Firing Rate.

$$r(t) = \frac{\sum^{k} n_k(t; t + \Delta t)}{k} / \Delta t \tag{2.18}$$

Time-Dependent Firing Rate measure is a useful method to evaluate neuronal activity, in particular in the case of time-dependent stimuli. The obvious problem with this approach is that it can not be the coding scheme used by neurons in the brain. Neurons can not wait for the stimuli to repeatedly present in an exactly same manner before generating response.

Temporal Coding

The basic idea of Temporal Coding is as simply stated as that of the Rate Coding: information about the stimulus or action is contained in the relative timing of spikes, not just in, or instead of in, the rate of those spikes. The meaning, however, of "relative timing" is flexible and so there are actually multiple temporal codings, which range from merely extreme examples of rate coding to incompatible schemes.

Neurons exhibit high-frequency fluctuations of firing-rates which could be noise or could carry information. Rate Coding models suggest that these irregularities are noise, while Temporal Coding models suggest that they encode information.

The original scheme of Temporal Coding was thought as if neurons could encode information as binary data because of the nature of the discrete possible values of the action potential, 1 or 0, spike or not spike. Temporal coding allows the sequence 000111000111 to mean something different from 001100110011, even though the mean firing rate is the same for both sequences. Nevertheless, this scheme would mean that such codes would have to operate most efficiently when 1's and 0's appear equally frequently, which is very expensive.

Sparse Temporal Coding is variant of Temporal Coding which is the representation of items by the strong activation of a relatively small set of neurons. For each stimulus, this is a different subset of all available neurons.

Given a potentially large set of input patterns, sparse coding algorithms (e.g. sparse auto-encoder) attempt to automatically find a small number of representative patterns which, when combined in the right proportions, reproduce the original input patterns. The sparse coding for the input then consists of those representative patterns

Population Temporal Coding is another variant of Temporal Coding which instead of approaching the problem single-cell wise, each neuron has a distribution of responses over some set of inputs, and the responses of many neurons may be combined to determine some value about the inputs.

Population Temporal Coding grasps the essential features of neural coding and yet is simple enough for theoretic analysis.

2.6.2 Neuromorphic Hardware

Trying to mimic our brain structure and functioning is one of the most optimistic objectives of this century and although this field is developing at the speed of light, the purpose of building such thing is still very far in time. The functioning of the brain and its structure is very strong related and hence they must work properly not only on their own, but also combined together.

In terms of structure, research has brought new hardware capable of running brain-like algorithms: Neuromorphic Hardware. Neuromorphic Hardware involves any electrical device which mimics the natural biological structures of our nervous system. The goal is to impart cognitive abilities to a machine by implementing neurons in silicon. Due to its much better energy efficiency and parallelism it is being considered as an alternative over conventional architectures and very energy consuming GPUs.

The efficiency of Neuromorphic Hardware is achieved because of the asynchronous nature of the on-chip processing, like a human brain. Each neuron does not need to be updated at every time step. Only the ones which are in action require power. This is called event-driven processing and is the most important aspect for rendering neuromorphic systems viable as a suitable alternative for conventional architectures.

The two most famous neuromorphic chips are *SpiNNaker* (Spiking Neural Network Architecture) and *SyNAPSE* (Systems of Neuromorphic Adaptive Plastic Scalable Electronics). In the Figure 2.16 both chips are shown.

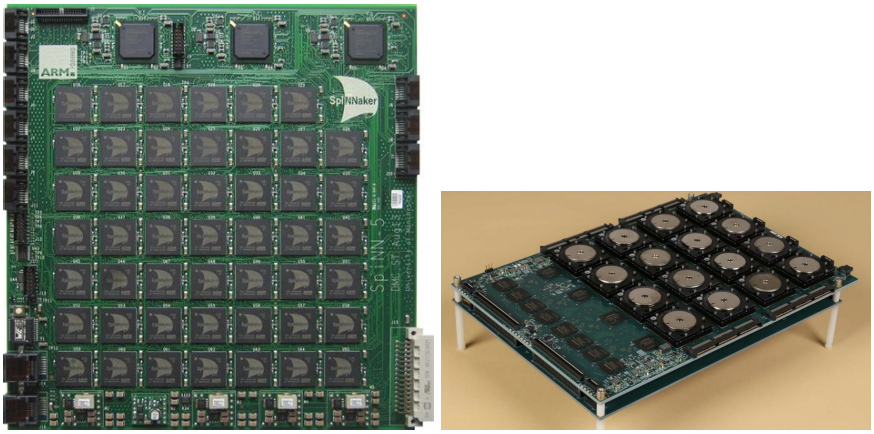


FIGURE 2.16: Neuromorphic chips. Left is SpiNNaker and right is DARPA SyNAPSE with TrueNorth chips from IBM

Recently, commercial neuromorphic chips have been developed by well known companies such as Google or Intel. Google has developed their famous *Tensor Processing Unit (TPU)* which are a perfect fit for their even more famous deep learning framework *Tensorflow*. Intel's neuromorphic research test chip *Loihi* is also available and it uses asynchronous spiking neural networks.

There are also other chips smartphone oriented developed by popular companies such as Qualcomm, Apple or Huawei that are already built in smartphones that are being used daily by ordinary people.

In addition, neuromorphic hardware for perceiving the environment is also required in order to take advantage of data that depends on time for training Spiking Neural Networks. This type of hardware is call Dynamic Sensor. These sensors are capable of capturing dynamic data, data that changes over time.

For vision purposes, Dynamic Vision Sensors (DVS) are the ones that are being utilised. These are bio-inspired vision sensors that output pixel-level brightness changes instead of standard intensity frames. They offer significant advantages over standard cameras: very high dynamic range, no motion blur and a latency in the order of microseconds.

The Figure 2.17 shows an example of a Dynamic Vision Sensor where it can be noted that frame-based cameras capture the entire frame with irrelevant information and data is lost between frames as it cannot capture all frames, while event-based cameras capture the entire movement of a body as a continuous stream of information where no information is lost between frames and only relevant data is captured.

FIGURE 2.17: Event Based Vision Hardware. By *Prophesee Event Based Vision Explanation*

Prophesee is the company inventor of the world's most advanced neuromorphic vision system which uses an Event-Based approach for artificial vision.

Spiking Neural Networks can take advantage of event-based hardware as they deliver asynchronous timestamped spikes which are called events, and there is no need to convert the data to spike trains because it is already that kind of data.

2.6.3 Event-Based Datasets

Conventional static dataset that are used for conventional ANNs do not unlock the potential of Spiking Neural Networks. ANNs perform well on those datasets, but SNNs are more than one step behind.

Because of the temporal nature of SNNs, proper dataset should be created in order to get fair results. These datasets are called Event-Based Datasets, and are usually created using neuromorphic sensors. For artificial vision, Dynamic Vision Sensors (DVS) are employed and as it has been explained, these record only motion, not entire frames. The Figure 2.18 shows how a DVS is capable of capturing data of a building.

One of the earliest and most famous event-based datasets is the *Neuromorphic-MNIST* (N-MNIST) dataset, which is a spiking version of the original frame-based MNIST dataset. This dataset was obtained using an ATIS sensor that was moving while viewing MNIST examples on a LCD monitor (this way the digits could be captured by the neuromorphic sensor, as it only catches moving objects). Although it achieved capturing temporal data, this way still fails to explore the true nature of SNNs.

Sound datasets could be interesting for training SNNs as their nature is temporal. A very recent publication, delivers *Two public domain datasets for spiking neural networks*. One of those datasets, SHD, consists of recordings of spoken digits from 0 to 9, just like MNIST. This dataset could be interesting for evaluating SNNs. The work is cited here Cramer, B., 2019.

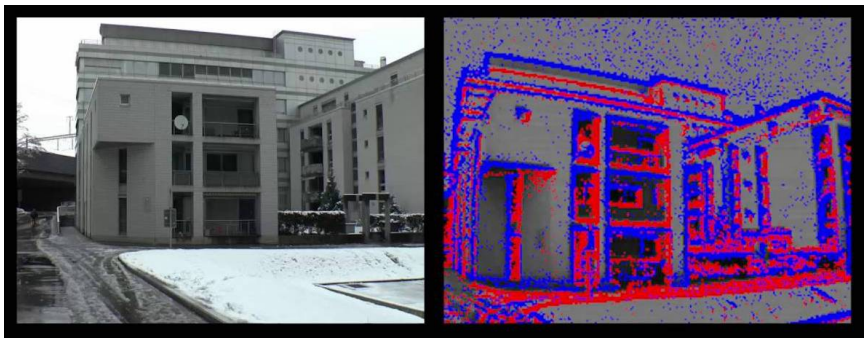


FIGURE 2.18: Event-Based data of a building captured using a Dynamic Vision Sensor. Snapshot taken from the video *The Event-Camera Dataset and Simulator*

Chapter 3

Technologies

Coding Spiking Neural Networks is not an easy task. SNNs do not work as conventional ANNs do and thus the usual deep learning frameworks would not work. The basis had to be rebuilt and recoded in order to match the functioning of SNNs.

Thankfully, although Spiking Neural Networks are not as explored as conventional ANNs, some libraries have been developed so implementing SNNs is not as difficult as it once was. Among these libraries are: *The Brian spiking neural network simulator*, *The Neural Simulation Technology Initiative*, *Nengo*, *BindsNET*, etc. And there are many others.

Of all, Nengo is one of the most complete ones and it will be the one that is going to be used here. Further information on Nengo is explained down below.

BindsNET has been released the past year 2018/2019 and although is not going to be used in this work, it is one to be taken into account. This library is written with *PyTorch* under the hood which it already means that is up to date, as this framework is one of the most utilised for research (even more than *Tensorflow*) and it is very easy to use. It has also been implemented focusing on Reinforcement Learning by using the *OpenAI Gym* toolkit, which elevates its potential to another level.

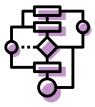
Because the purpose of this work is to focus on explaining SNNs, Nengo seems to be the best option.

3.1 Nengo

The Nengo Brain Maker is a Python package for building, testing, and deploying neural networks. The main features of this library are as shown down below:


Spiking or traditional non-spiking models


Fully scriptable or GUI-based development


Highly customisable or use available modules


Tackle dynamic information processing


Easily exploit the latest hardware

3.1.1 The Nengo Ecosystem

The Nengo ecosystem is made up of several interacting projects. The following image groups these projects into rough categories 3.1.

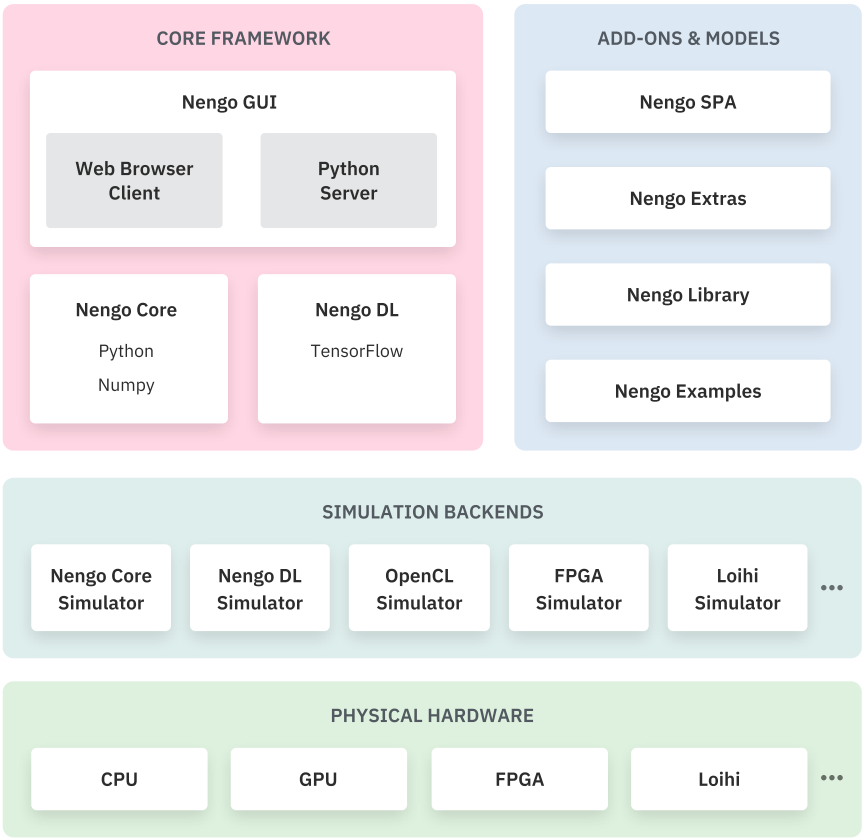


FIGURE 3.1: The Nengo Ecosystem. Downloaded from *Nengo*

Nengo

The core of the Nengo ecosystem is the Python library nengo, which is capable of building and simulating spiking and non-spiking large-scale neural models. Spiking neural networks in Nengo are only simulated, but they are not actual SNNs. This framework includes the five Nengo objects (Ensemble, Node, Connection, Probe, Network) and a NumPy-based simulator.

Nengo is built upon two theoretic underpinnings, the Neural Engineering Framework (NEF) and the Semantic Pointer Architecture (SPA). Nengo primarily works using the NEF but it has another module called NengoSPA built for dealing directly with semantic pointers.

Nengo differs primarily from other modelling software in the way it models connections between neurons and their strengths. Using the NEF, Nengo allows defining connection weights between populations of spiking neurons by specifying the function to be computed, instead of forcing the weights to be set manually, or use a learning rule to configure them from a random start.

Nengo objects allow building complex spiking neural networks using different types of neurons, learning rules, connections, etc. Also, it is extensible and flexible enough to let the user define its own neuron types and learning rules, get input directly from hardware, build and run deep neural networks, drive robots, and even

simulate the model on a completely different neural simulator or neuromorphic hardware.

NengoDL

With Nengo it is possible to train Spiking and non-spiking models, supporting Tensorflow deep learning framework with its latest in-built Keras library on its most recent version Tensorflow 2.0. Nengo has documentation with explained examples on how to use this tool for deep learning practitioners that come from Tensorflow/Keras by allowing to import deep learning models created outside Nengo for biological inference, to insert Tensorflow/Keras syntax in Nengo syntax and to build Tensorflow/Keras equivalents using Nengo's syntax.

Because there has been an increase of deep learning practitioners, Nengo's designers decided to create another section called NengoDL, which included all the features mentioned above to favour the transition to Nengo's platform.

NengoDL is a different simulator class that takes Nengo models as input, allowing this way to use some underlying computational framework like Tensorflow.

To sum up, NengoDL includes new extra features that Nengo Core does not:

- Optimising the parameters of a model through deep learning training methods (using the Keras API)
- Faster simulation speed, on both CPU and GPU
- Inserting networks defined using TensorFlow (such as deep learning architectures) directly into a Nengo model

Nengo GUI

Nengo GUI is an HTML5-based interactive visualiser for large-scale neural models created with Nengo. The GUI lets the user see the structure of a Nengo model, plots spiking activity and decoded representations, and enables the user to alter inputs in real time while the model is running.

The GUI looks like the one in the Figure 3.2.

As the figure shows, the Nengo GUI has 4 main components:

- **Netgraph related things:** These could include sub-components such as plots of the spike trains generated by the activation of neurons, plots of the values that can be decoded from that spiking activity change, etc.
- **Graph components:** These involve Ensemble objects of groups of neurons, different type of connections, input of stimuli, networks, etc.
- **Ace editor:** Coding editor with auto-completion and correction highlighting which allows the user to code neural models and it automatically synchronises with the visual model.
- **Simulation control:** UI where the simulation can be started by pressing the play button, stopped and replayed. It shows the speed at which the simulation is running (sometimes the specifications of the machine will restrain the maximum speed available) and the elapsed time of the simulation.

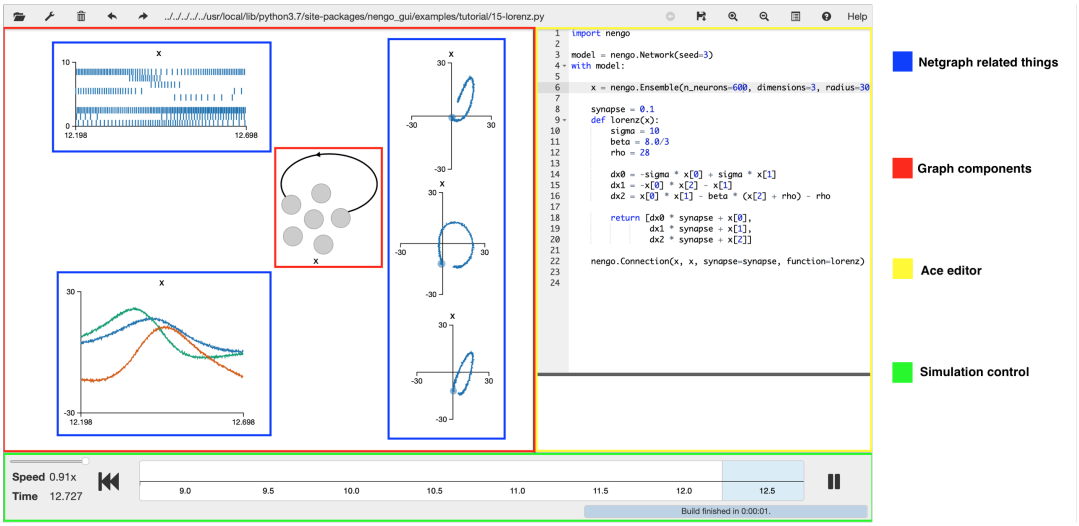


FIGURE 3.2: Nengo GUI. Main components

Nengo SPA

Nengo SPA is an implementation of the Semantic Pointer Architecture for Nengo. The Semantic Pointer Architecture provides an approach to building cognitive models implemented with large-scale spiking neural networks.

Briefly, the semantic pointer hypothesis states:

> Higher-level cognitive functions in biological systems are made possible by semantic pointers. Semantic pointers are neural representations that carry partial semantic content and are composable into the representational structures necessary to support complex cognition.

The figure 3.3 shows the architecture of SPAUN ("Semantic Pointer Architecture Unified Network"), a cognitive architecture pioneered by Chris Eliasmith of the University of Waterloo Centre for Theoretical Neuroscience.

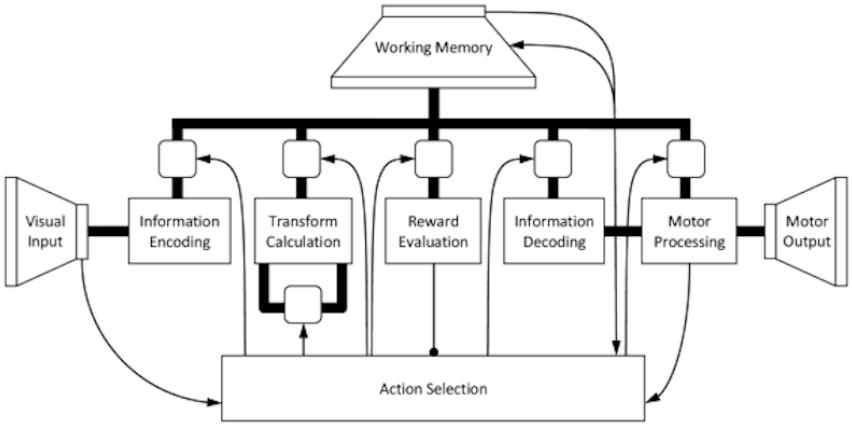


FIGURE 3.3: Semantic Pointer Architecture Unified Network by University of Waterloo Centre for Theoretical Neuroscience

Some of the features of this module listed by Nengo are:

- Write arbitrarily complex expressions with type checking involving neurally represented and static Semantic Pointers.
- Quickly implement action selection systems based on a biological plausible model of the basal ganglia and thalamus.
- Neural representations are optimised for representing Semantic Pointers.
- Support for using different binding methods with algebras. Nengo SPA ships with implementations of circular convolution (default) and vector-derived transformation binding (VTB), which is particularly suitable for deep structures. Different binding operations/algebras can be mixed in a single model.
- Seamless integration with non-SPA Nengo models.

Simulation Backends

Nengo is designed so that models created with the Nengo front-end API work on a variety of different simulators, or back-ends For example, back-ends have been created to take advantage of GPUs and neuromorphic hardware.

Nengo has several extensions of its main back-end:

- **Nengo FPGA:** It allows portions of a network to be run on an FPGA to improve performance and efficiency. A FPGA (field-programmable gate array) is a board with built-in logic gates where there are no fixed connections but programmable ones, allowing all possible connections between components and a very high performance.
- **Nengo Loihi:** It permits the user to build Nengo models for inference in Intel's Loihi architecture.
- **Nengo OpenCL:** It uses an OpenCL-based simulator for brain models built using Nengo. OpenCL (Open Computing Language) is an open framework for writing programs that execute across heterogeneous platforms such as CPUs, GPUs, DSPs, FPGAs, and others.
- **Nengo SpiNNaker:** It contains a SpiNNaker-based simulator for models built using Nengo. It allows real-time simulation of large-scale models.
- **Nengo MPI:** A C++/MPI back-end for the nengo neural simulation library. The message passing interface (MPI) is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory.

Chapter 4

Experiments and Results

The source of the code shown in this section is available in the github of the author for the reader to follow Unai Garay Maestre's github.

4.1 Experimenting with Nengo

4.1.1 Basis

Nengo is based in three main cornerstones:

- Representation
- Transformation
- Dynamics

These three principles are going to be explained in this notebook along with the basic components of Nengo.

Principle 1: Representation

Encoding: Neural populations represent time-varying signals through their spiking responses. A signal is a vector of real numbers of arbitrary length. This example is a 1D signal going from -1 to 1 in 1 second.

```
import nengo
model = nengo.Network(label="NET")
with model:
    input = nengo.Node(lambda t: t * 2 - 1)
    input_probe = nengo.Probe(input)
```

A class Network from nengo is created using the first line in the above cell snippet of code. In that network a single Node is created. Nodes provide non-neural inputs to Nengo objects and process outputs. Nodes can accept input, and perform arbitrary computations for the purpose of controlling a Nengo simulation. In this example, it only represents the function $t * 2 - 1$. A Probe object is added which is solely purpose is to collect data from the simulation.

In order to run this simulation, the following two lines are needed:

```
with nengo.Simulator(model) as sim:
    sim.run(1.0)
print(sim.data[input_probe])
```

This piece of code uses the Simulator back-end for referencing Nengo models which are passed as parameters. It builds and runs the simulation for the time specified: 1 second. Afterwards, the data recollected from the Probe can be visualised by accessing the *sim.data* Simulation Data structure.

The output that can be seen in the Figure 4.1 is achieved.

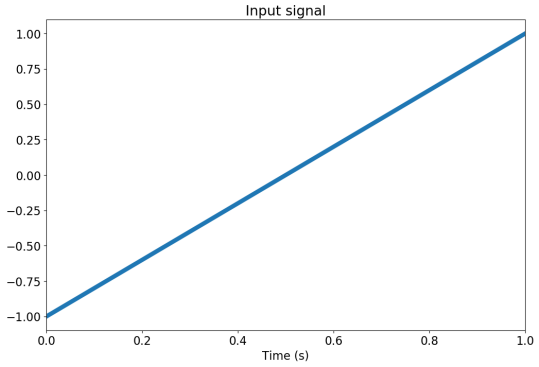


FIGURE 4.1: Encoding signal using Nengo

These signals drive neural populations based on each neuron's tuning curve. The tuning curve describes how much a particular neuron will fire as a function of the input signal.

```
intercepts, encoders = aligned(8)  # Makes evenly spaced intercepts
with model:
    A = nengo.Ensemble(
        8,
        dimensions=1,
        intercepts=intercepts,
        max_rates=Uniform(80, 100),
        encoders=encoders)
with nengo.Simulator(model) as sim:
    eval_points, activities = tuning_curves(A, sim)
```

An Ensemble is a group of neurons that collectively represent a vector and in this example it has been initialised with 8 neurons (first parameter) and only one dimension as input. This produces the following plot 4.2 which shows the firing of every neuron depending on the input, the tuning curve.

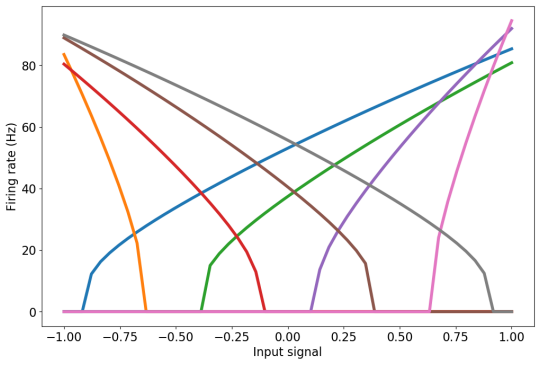


FIGURE 4.2: Tuning curves

These neurons can be driven with the input signal and their spiking activity can be observed over time 4.3.

```
with model:
    nengo.Connection(input, A)
    A_spikes = nengo.Probe(A.neurons)
```

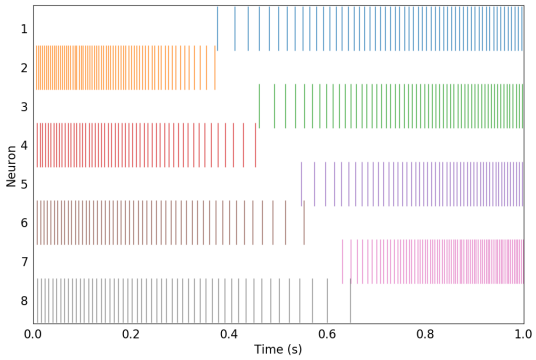


FIGURE 4.3: Tuning curves

Decoding: The originally encoded input signal can be estimated by decoding the pattern of spikes. To do this, the spike train is filtered with a temporal filter that accounts for postsynaptic current (PSC) activity 4.4.

```
model = nengo.Network(label="NET")
with model:
    input = nengo.Node(lambda t: t * 2 - 1)
    input_probe = nengo.Probe(input)
    intercepts, encoders = aligned(8)  # Makes evenly spaced intercepts
    A = nengo.Ensemble(8, dimensions=1,
                       intercepts=intercepts,
                       max_rates=Uniform(80, 100),
                       encoders=encoders)
    nengo.Connection(input, A)
    A_spikes = nengo.Probe(A.neurons, synapse=0.01)
```

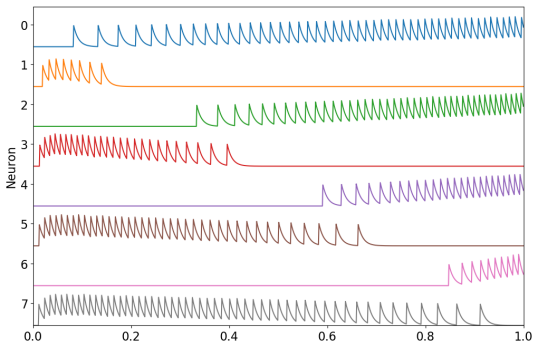


FIGURE 4.4: Temporal filter

Then those filtered spike trains are multiplied with decoding weights and sum them together to give an estimate of the input based on the spikes.

The decoding weights are determined by minimising the squared difference between the decoded estimate and the actual input signal.

The accuracy of the decoded estimate increases as the number of neurons increases as it can be seen in the figure 4.5

```
with model:
    A_probe = nengo.Probe(A, synapse=0.01)  # 10ms PSC filter
```

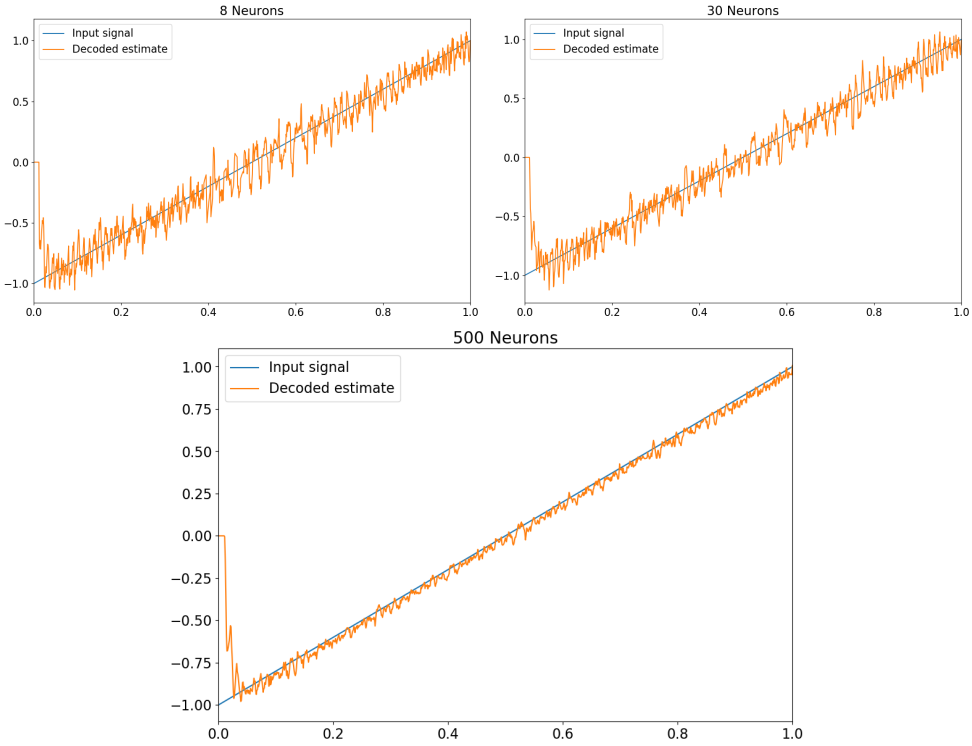


FIGURE 4.5: Estimated Output with different number of neurons

Principle 2: Transformation

In general, the transformation principle determines how we can decode spike trains to compute linear and nonlinear transformations of signals encoded in a population of neurons. Those transformed signals then can be projected into another population and the same process could be applied again. Essentially, this provides a means of computing the neural connection weights to compute an arbitrary function between populations.

The following example tries to represent a sine wave in the first population. The second population attempts to compute the inverse of that signal by passing the function lambda $x : -x$ into the connection between the previous population and the new one. Finally, the square of that second population (x^2) is calculated into the third population by using the function lambda $x : x * *2$. The represented signals and the spikes of their neurons are shown in the following Figure 4.6.

```
model = nengo.Network(label="NET")
with model:
    input = nengo.Node(lambda t: np.sin(np.pi * t))
    # Identity
```

```
        A = nengo.Ensemble(30, dimensions=1, max_rates=Uniform(80, 100))
        nengo.Connection(input, A)
        A_spikes = nengo.Probe(A.neurons)
        A_probe = nengo.Probe(A, synapse=0.01)

        # -A
        minusA = nengo.Ensemble(30, dimensions=1, max_rates=Uniform(80, 100))
        nengo.Connection(A, minusA, function=lambda x: -x)
        minusA_spikes = nengo.Probe(minusA.neurons)
        minusA_probe = nengo.Probe(minusA, synapse=0.01)

        # (-A)^2
        A_squared = nengo.Ensemble(30, dimensions=1, max_rates=Uniform(80, 100))
        nengo.Connection(minusA, A_squared, function=lambda x: x ** 2)
        A_squared_spikes = nengo.Probe(A_squared.neurons)
        A_squared_probe = nengo.Probe(A_squared, synapse=0.02)
    with nengo.Simulator(model) as sim:
        sim.run(6)
```

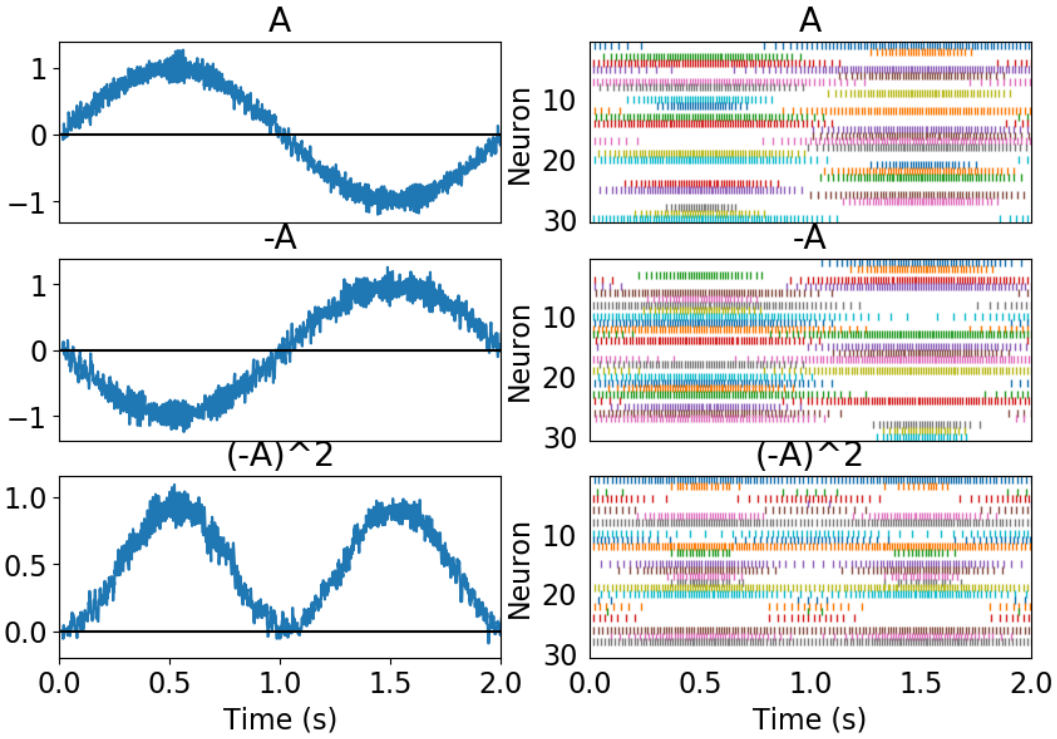


FIGURE 4.6: Transformation between populations

Principle 3: Dynamics

Until now, Ensembles were only representing the values as generic signals. Nevertheless, in a dynamical system they could be taken as state variables and thus methods of control theory or dynamic systems could be applied to brain models. Nengo automatically translates from standard dynamical systems descriptions to descriptions consistent with neural dynamics.

In order to tell Nengo to implement dynamics, populations have to be connected recurrently (i.e., to themselves). That is, a Connection that will feed a value into an Ensemble that is the same value that the Ensemble is currently representing is formed. This means data can be stored over time.

Below is a simple harmonic oscillator implemented using this third principle. It needs is a bit of input to get it started, meaning the input will have a value of [1, 0] (two inputs) for 0.1 seconds and then it will become both 0. The following Figure 4.7 shows the results.

```
model = nengo.Network(label="NET")
with model:
    input = nengo.Node(lambda t: [1, 0] if t < 0.1 else [0, 0])
    oscillator = nengo.Ensemble(200, dimensions=2)
    nengo.Connection(input, oscillator)
    nengo.Connection(
        oscillator, oscillator, transform=[[1, 1], [-1, 1]], synapse=0.1)
    oscillator_probe = nengo.Probe(oscillator, synapse=0.02)
```

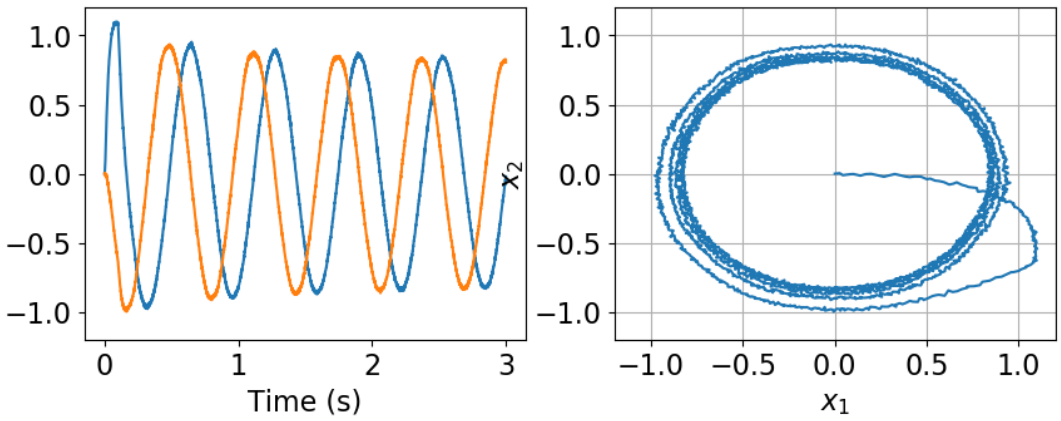


FIGURE 4.7: Dynamics. Oscillator

The term *transform* is a short way of expressing a transformation without the necessity of defining a function. For example, the transform being applied in this case is $[[1,1],[-1,1]]$ which for the term *function* is translated into the lambda x : $(x[0]+x[1], -x[0]+x[1])$.

4.1.2 Learning in Nengo GUI

The past example showed how to approximate a function across a connection by using the term *function* and passing in an actual function or by using the term *transform* and passing in weight vectors, and this was done on the building process of the model, offline. Nevertheless, there is another approach for online learning which is error-driven and it is going to be shown in this section.

This time Nengo GUI is going to be utilised as an alternative and in order to show graphically what error-driven learning is and looks like.

First, a model is created with an input node that feeds in a white signal or white noise (random signal with equal intensity at different frequencies) with two output signals coming out from that node. An ensemble (pre) and a connection between the input node and that ensemble is created (the more nodes are added the more faithful

the signal will be). Finally, another ensemble (post) with another connection between the last ensemble (pre) and this one (post) is created. This connection attempts to approximate a random function of the white signal and using offline learning is not possible.

```
import numpy as np
import nengo
from nengo.processes import WhiteSignal
np.random.seed(42)

model = nengo.Network()
with model:
    inp = nengo.Node(WhiteSignal(60, high=5), size_out=2)
    pre = nengo.Ensemble(60, dimensions=2)
    nengo.Connection(inp, pre)
    post = nengo.Ensemble(60, dimensions=2)
    conn = nengo.Connection(pre, post, function=lambda x: np.random.random(2))
```

This code in Nengo GUI produces the architecture and the output shown in the figure 4.8.

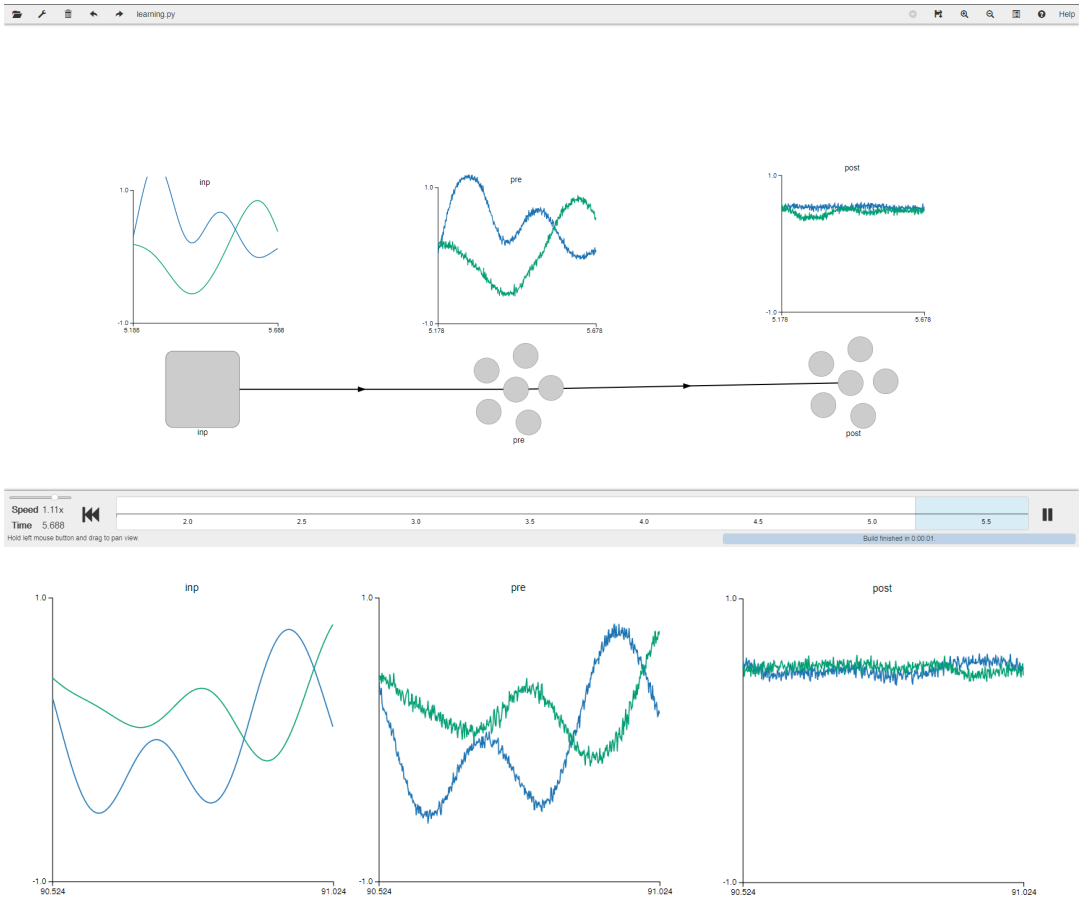


FIGURE 4.8: Nengo GUI. No learning applied

Looking at the results it can be noted that the connection between pre and post does not approximate the white signal at all.

In order to add online error-driven learning to this connection, an error must be computed and minimised. Nengo provides many learning rules and one supervised learning rule that could be utilised for this purpose is the PES (Abstract Prescribed Error Sensitivity). Therefore, a new ensemble for calculating the error and a connection from the pre ensemble and the post node to that ensemble have to be created. After that, a learning rule has to be added to the connection between the pre and post ensembles which is then connected by the error ensemble in order to minimised the error computed. This results in the following code to be added:

```
# Error ensemble
error = nengo.Ensemble(60, dimensions=2)

# Error = actual - target = post - pre
nengo.Connection(post, error)
nengo.Connection(pre, error, transform=-1)

# Add the learning rule to the connection
conn.learning_rule_type = nengo.PES()

# Connect the error into the learning rule
nengo.Connection(error, conn.learning_rule)
```

If the learning rule will keep being applied indefinitely, so an inhibit function should be applied to the error ensemble when the error has been reduced enough. In this example, the inhibit function will shut down the error computation after a a duration of 10 seconds.

The Figure 4.9 shows how this methodology is able to learn the white signal properly and how, after a period of time larger than 10 seconds, the error is inhibited.

```
# Outside the model
def inhibit(t):
    return 2.0 if t > 10.0 else 0.0

inhib = nengo.Node(inhibit)
nengo.Connection(inhib, error.neurons, transform=[[-1]] * error.n_neurons)
```

In the top image of the Figure 4.9, it can be seen how the network is trying to learn the white signal, the post ensemble plot shows that the signal is starting to resemble to the input signal and the error ensemble plot shows that there is still error in the approximation. In the bottom image, error signal has been suppressed after 10 seconds and pre and post plots now look very similar.

4.2 Classification with NengoDL and Tensorflow

In this section, a hybrid model between Nengo and Tensorflow is going to be developed. NengoDL permits the implementation of models using deep learning frameworks, such as Tensorflow and Keras, within Nengo's environment and architecture.

NengoDL has a class for inserting Tensorflow layers called nengo_dl.TensorNode and a class for inserting Keras layers called nengo_dl.Layer. Because Keras has become widely used, it is much more simpler than Tensorflow and in the latest update

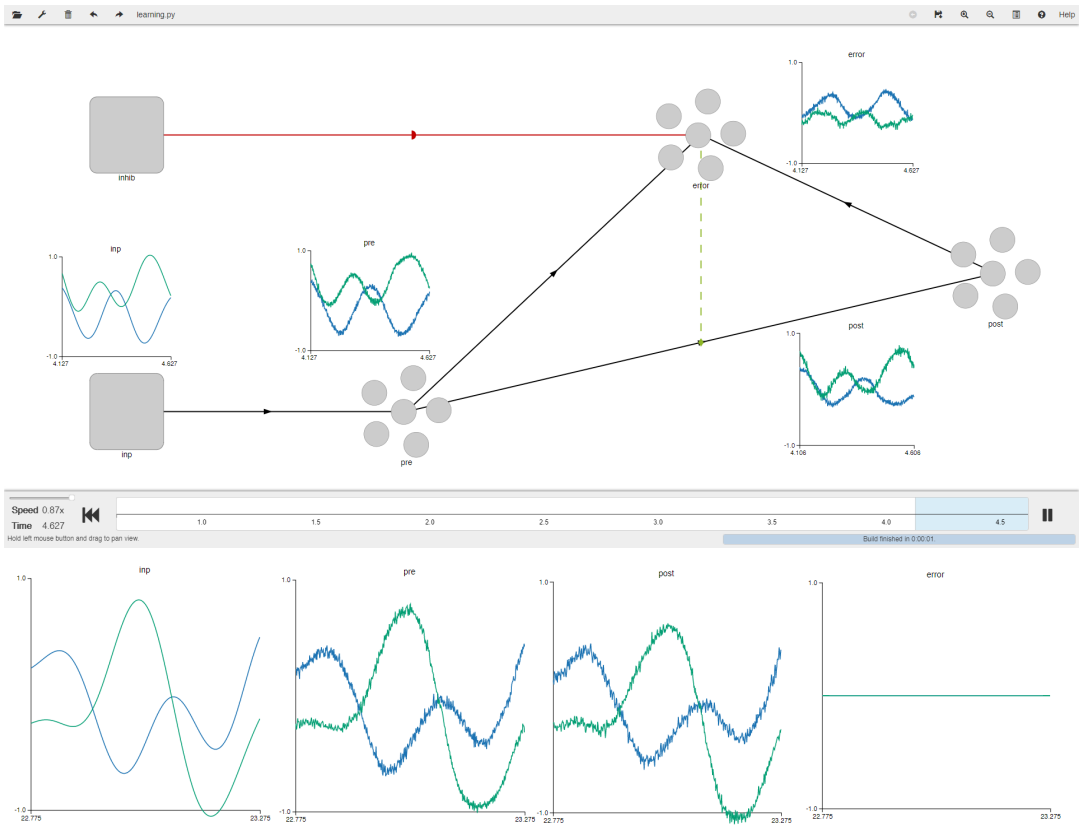


FIGURE 4.9: Nengo GUI. Learning white signal by using PES rule

of Tensorflow to version 2.0 Keras has been included as part of the API, Keras will be the selected option.

In order to follow the code for this implementation, the github url containing the Jupyter Notebook where it has been developed is fashion-mnist.ipynb.

4.2.1 Dataset: Fashion MNIST

The selected dataset for this task is the Fashion MNIST dataset. Fashion-MNIST is a dataset of Zalando's article images Xiao, Rasul, and Vollgraf, 2017 consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. The Figure 4.10.

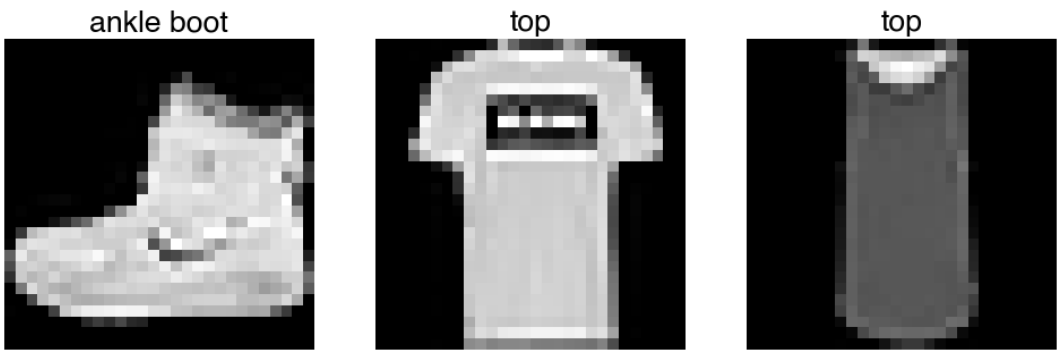


FIGURE 4.10: Zalando's Fashion MNIST dataset examples

The labels of the dataset correspond to the descriptions in the table 4.1.

Description	T-shirt/Top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle Boot
Label	0	1	2	3	4	5	6	7	8	9

TABLE 4.1: Zalando's Fashion MNIST dataset label correlations

In order to load this dataset, Tensorflow makes it easy with its API by only having to write:

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
```

4.2.2 SNN Architecture

As it has been mentioned, the Spiking Neural Network is going to be built by inserting Keras layers into the Layer class of NengoDL. Before achieving a good architecture, there has been some exploration. First, a Fully Connected Spiking Neural Network was built. Secondly, a Convolutional Spiking Neural Network was made. Finally, Dropout layers were added in order to make the network generallise better, and this approach was tested in both type of networks.

The final architecture is shown in the following table 4.2.

Layer Stack	Layer Type	Filters/Value	Kernel	Input
1	Input			(784, 1)
	Conv2D	32	3x3	(28, 28, 1)
	LIF			
	Dropout	0.25		
2	Conv2D	64	3x3	(26, 26, 32)
	LIF			
	Dropout	0.25		
3	Conv2D	64	3x3	(12, 12, 64)
	LIF			
	Dropout	0.25		
4	Dense	10		

TABLE 4.2: Spiking Neural Network Architecture for Fashion MNIST dataset

This architecture is based in 4 stacked layers. In the first layer stack, an input node has to be created by using standard Nengo API with an input shape of the size of the images 784. This input node is followed by a Conv2D layer from Keras, a Convolutional layer of two dimensions, with an input dimension of 28x28x1, 32 filters and a kernel size of 3x3. Consequently, a LIF neuron is inserted in the Layer class, creating a standard Nengo Ensemble with equal neurons as the multiplication of the input shape (784) and a LIF neuron as neuron type. Finally, this is followed by a Dropout layer from Keras with 25% of neurons being randomly disabled.

Two more stacks of layers with the same type of layers (not including the input node) but with varying parameters follow the first one. At the end, the fourth stack, a Dense layer with 10 units (same number as the number of classes) is implemented in order to decide which class the input belongs to.

After that, the simulation is run under NengoDL's back-end.

A new dimension for taking into account the time needs to be added for the training set. For the test set the same dimension is added too, but these samples need to be exposed to the network for a period of time and thus they need to be run for some steps, and depending on the number of steps it will get different accuracy. Generally, more steps mean more accuracy.

For the training process, the RMSprop optimiser from Tensorflow has been employed with a Learning Rate of 0.001. In order to understand the RMSprop optimiser the original source should be reviewed Hinton, 2012. Furthermore, the loss function Sparse Categorical Cross-entropy from Tensorflow is chosen for calculating the error.

Finally, the model is trained for 30 epochs. The number of epochs has been chosen given that the last 5 epochs there was no much improvement in the training loss.

4.2.3 Results

There is going to be two different sets of comparisons:

- Fully Connected SNN with Dropout VS Convolutional SNN without Dropout VS Convolutional SNN with Dropout
- Different values for the number of time-steps that the input data has been exposed to the network
- Output of the models when predicting a single image

For the first part, the following table shows the results 4.3 where the time-steps have been set to 100 so the results are less affected by the time exposed and rather they are evaluated by the actual architecture.

Model	Accuracy	Loss
Fully Connected SNN with Dropout	0.839	0.868
Convolutional SNN	0.875	**0.512**
Convolutional SNN with Dropout	**0.894**	**0.512**

TABLE 4.3: SNNs results comparison for Fashion MNIST

From the table 4.3 can be stated that the Convolutional SNN with Dropout performs better than the other two, although it is closely followed by the one without Dropout. The computational loss, however, looks the same for both with and without Dropout. Fully Connected SNN has very high loss, 0.868, which discards itself for competing against the other two models.

This work is not meant as an attempt to achieve state of the art results, but to show how to implement Spiking Neural Networks with NengoDL and trying out different architectures and combinations, hence other's work achievements are not taken into account.

Continuing with the results, the following Figure 4.11 can be shown as a measure of the performance of every model for different time-steps.

In this chart, the first thing that can be seen is that the accuracy improves directly proportional to the increase of time-steps. The second thing, is that within the range of 2 to 10 time-steps there is a considerable gap in accuracy. After 10 time-steps, little improvement is accomplish.

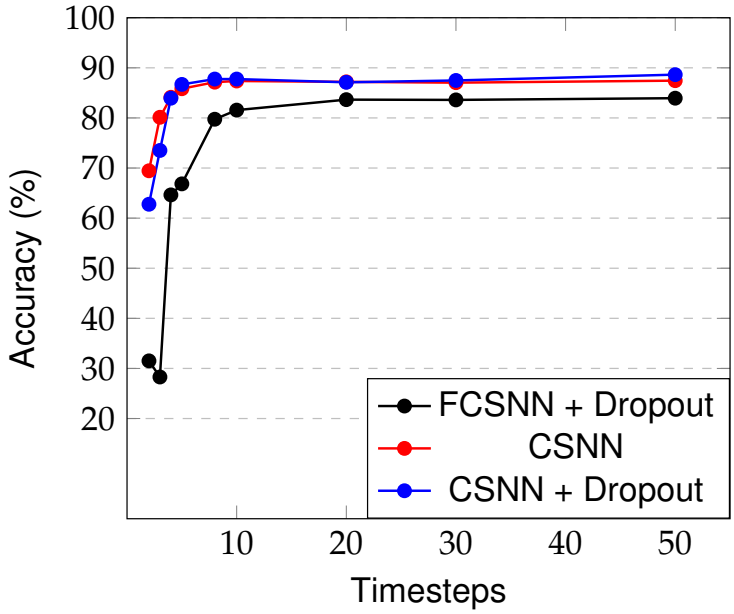


FIGURE 4.11: Comparison of the accuracy of the three models for different time-steps

Another thing to take into account is that in the very first time-steps the Convolutional SNN without Dropout performs better than the one with Dropout, although in time-step 5 the CSNN with Dropout is already obtaining higher accuracy.

For the final comparison, the models have been presented an image from the testing set for 100 time-steps. The Figure 4.12 shows the comparison of the prediction of an Ankle Boot for every model.

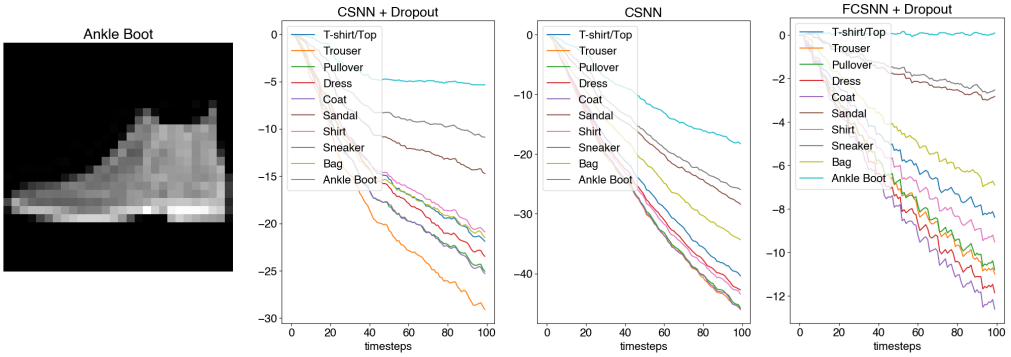


FIGURE 4.12: Image prediction neuron spike comparison. Ankle Boot

The figure shows different results for each model, but with a common higher prediction: Ankle Boot. One of the difference between them could be how the prediction spike differs from the other prediction spikes (the higher the better), where the FCSNN + Dropout model would be the one with higher difference and the CSNN model would be the opposite. Another think to take into account is how noissy the spikes look (the higher the worse), and here the FCSNN + Dropout model would be the one with higher noise whereas the CSNN model would be the less.

4.3 Classification using native Nengo and Nengo GUI

As stated before, Nengo GUI is a Nengo based visual interface for developing and simulating Spiking Neural Networks visually and intuitively. In the last section, classification of Fashion MNIST clothes images by using NengoDL and Tensorflow has been successfully employed, and in this section a similar network is going to be built by using purely Nengo within the Nengo GUI platform for visually results.

This example can be found in the following link fashion-mnist$_{v}ision_{n}etwork.py$.

4.3.1 Nengo GUI SNN Architecture

The architecture of the Spiking Neural Network built in Nengo GUI is represented in the Figure 4.13.

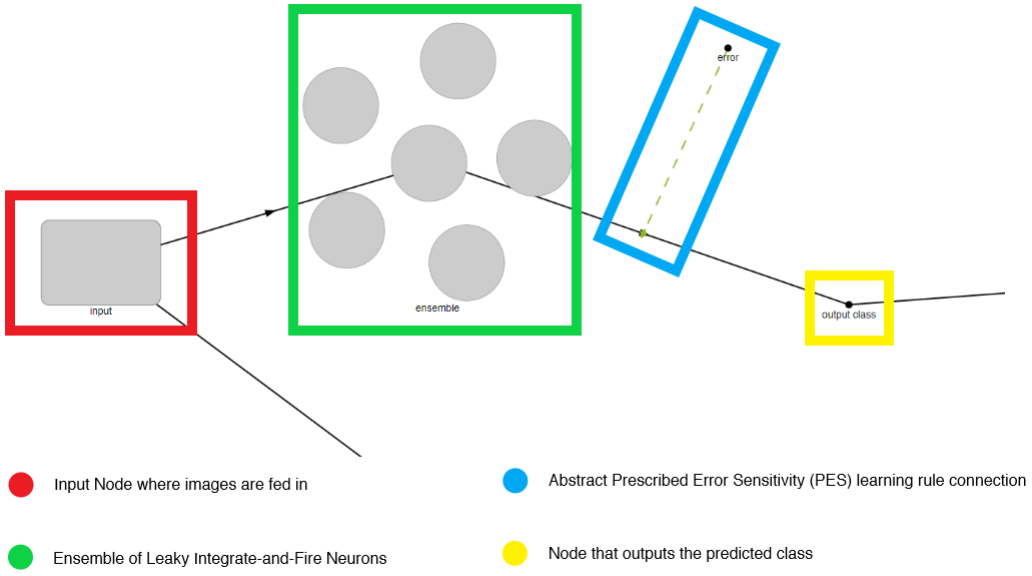


FIGURE 4.13: SNN architecture for Fashion MNIST in Nengo GUI

The Input Node is where the Fashion MNIST images of clothes are fed in. The Ensemble is a poblation of 1500 Leaky Integrate-and-Fire Neurons that try to approximate the expected input images to the expected output which are the classes of clothes. The Abstract Prescribed Error Sensitivity (PES) learning rule connection tries to minimise the error of the predicted classes against the expected classes. Finally, the output class node receives the predicted output of the Ensemble as input.

4.3.2 Results

The Figure 4.14 shows a screenshot of the Spiking Neural Network running inference on the test set over time, after training has been done in the building process of the network by using the Nengo back-end.

The Node *display_node* receives the Fashion MNIST images as input from the Input Node and shows them in an html display. The module *output_spa* is a minimal SPA (Semantic Pointer Architecture) network for passing through data and in this case it handles the representation of the spikes of the *output_class* Node which are represented by a vocabulary of the names of the classes and are drawn in a

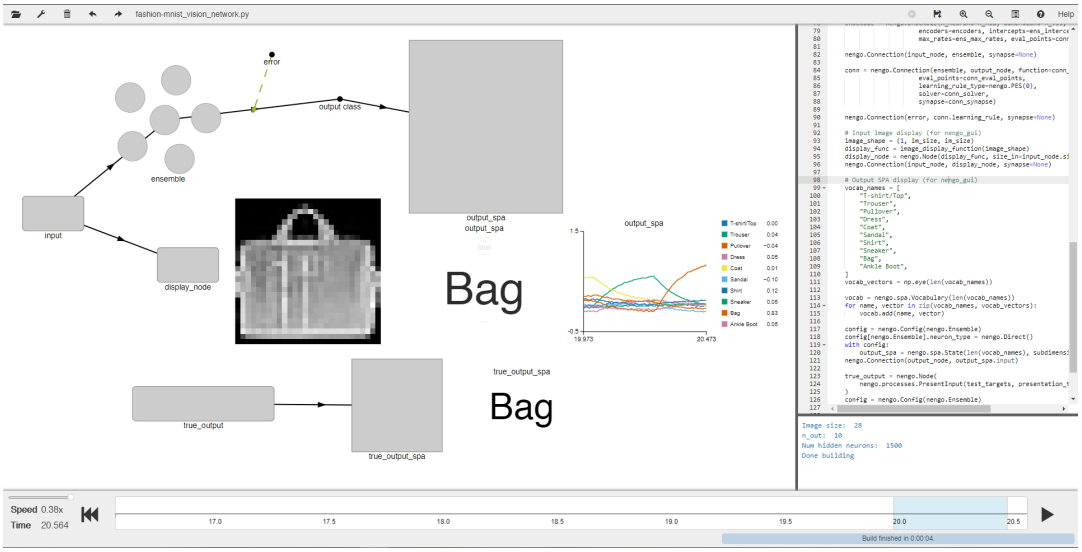


FIGURE 4.14: SNN inference on Fashion MNIST test set in Nengo GUI

semantic pointer cloud. Furthermore, the SPA outputs the actual value of the spikes of each neuron for each class and can be represented in a changing plot (right side of the architecture). Finally, a true label output for the test set is represented using another SPA module which is fed by the labels from the testing set. Ideally, the trained network actual output should match the true label output.

In the case of the Figure 4.14, the network is predicting properly the input image as a Bag with a very high spike confidence, but there are cases that do not match that well.

A demo has been recorded and it can be accessed directly from the link snn-nengo_gui.mp4 or by using the embedded link in the github page Demo SNN Nengo GUI.

Chapter 5

Conclusions

5.1 Project Summary

The purpose of this project is to review Spiking Neural Networks (SNNs): how they work, the different type of neurons, learning methods, how they are implemented in Deep Learning, conversion methods to Spike Trains and a brief explanation of neuromorphic hardware and event-based data. SNNs have proved to be a very good substitute for ANNs, at least on paper. They are more energy efficient, have faster inference, are able to process temporal data natively, have more brain similarity than ANNs and many other features. Nevertheless, because they work by firing spikes, discrete events, instead of continuous values like ANNs, they have to be trained differently as the spikes are not directly differentiable. Relatively new approaches have been developed with different learning methodologies (Unsupervised Learning, Supervised Learning and even Reinforcement Learning) and with bio-inspired learning rules, such that they have decrease the gap between SNNs and ANNs. In this work, a review of Spiking Neural Networks has been employed, from biological neuron level to Deep Spiking Neural architectures, and some implementation examples have been shown by using the Nengo library, which provides a flexible environment for training both SNNs and ANNs with the latest technologies available. At the end of the work, some conclusions are made and the next possible steps are mentioned along with some final thoughts.

5.2 Evaluation

After giving insights about Spiking Neural Networks, some of these concepts are put into practice by starting with very simple examples to training Deep Spiking Neural Networks using the Nengo library. The very first section of the implementation involves the explanation and coding examples of the 3 principles (Representation, Transformation, Dynamics) of NEF, the Neuromorphic Engineering Framework, implemented using Nengo. Also, in this section Nengo GUI, an HTML5-based interactive visualiser for large-scale neural models created with Nengo, is introduced and the learning of two continuous signals by using online learning on SNNs is achieved.

Subsequently, the next section already involves training a Deep Spiking Neural Network using the Deep Learning library from Nengo called NengoDL which implements the famous DL framework Tensorflow as back-end. Combining these two frameworks, a Convolutional SNN is trained for predicting the Fashion MNIST clothes dataset achieving a very good accuracy of nearly 90%. This model is directly compared with a Dense SNN which is not able to perform as good as the Convolutional version. Also, in this section different values of time-steps are tested in order to set an optimal value for the training of a SNN.

In the last section, Nengo GUI is again used for developing a similar network than the one from the last section, but using native Nengo objects. In this last experiment, the network is again trained using the Fashion MNIST dataset, the trained SNN is run for inference on the test set and, utilising the SPA module from Nengo, the predictions are shown in the GUI, thus achieving an excellent visually example which will give the reader well-established knowledge of Spiking Neural Networks and how they work in practice.

5.3 Further Work

In general, the main objective of this work is to provide some insights about Spiking Neural Networks theoretically and practically. Nevertheless, there has been some points that were not addressed and would be considered in future work:

- **Event-based datasets:** One of the most logical steps to take next is to train a SNNs using neuromorphic datasets, on which these algorithms can stand out and take advantage of their extra temporal dimension.
- **Hardware implementation:** Another possible step would be to try SNNs on neuromorphic hardware which is where these work better and their energy efficient inference would make a difference.
- **SNNs benchmarking:** Create a standard benchmark for evaluating SNNs performance instead of using ANNs bencharmks such as the MNIST dataset.

5.4 Final Thoughts

This has been a very challenging work because of the lack of biological background of the author. Although the author had artificial intelligence base knowledge, this type of networks with biological behaviour has led to countless hours of research and acquirement of knowledge. Regardless of the absence of expertise, the development of this thesis has been a very fascinating and enlightening experience.

Bibliography

Amirhossein Tavanaei. *Deep Learning in Spiking Neural Networks*. [Online; accessed November 10, 2019]. URL: https://arxiv.org/pdf/1804.08150.pdf.

— *Multi-layer unsupervised learning in a spiking convolutional neural network*. [Online; accessed November 17, 2019]. URL: https://ieeexplore.ieee.org/document/7966099.

Arxiv (2016). *Training Deep Spiking Neural Networks using Backpropagation*. [Online; accessed October 13, 2019]. URL: https://arxiv.org/pdf/1608.08782.pdf.

— (2017). *Spatio-Temporal Backpropagation for Training High-performance Spiking Neural Networks*. [Online; accessed October 13, 2019]. URL: https://arxiv.org/pdf/1706.02609.pdf.

— (2018). *BP-STDP: Approximating Backpropagation using Spike Timing Dependent Plasticity*. [Online; accessed October 13, 2019]. URL: https://arxiv.org/pdf/1711.04214v2.pdf.

BindsNET. [Online; accessed November 22, 2019]. URL: https://github.com/BindsNET/bindsnet.

Cambridge University Press. *Variability of Spike Trains and Neural Codes*. [Online; accessed November 23, 2019]. URL: https://neuronaldynamics.epfl.ch/online/Ch7.S2.html.

Cramer, B. Stradmann, Y., Schemmel, J. Zenke (2019). *The Heidelberg spiking datasets for the systematic evaluation of spiking neural networks*. [Online; accessed November 23, 2019]. URL: https://arxiv.org/abs/1910.07407.

Cun, Yann le (1988). *A Theoretical Framework for Back-Propagation*. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-88.pdf.

Diehl and Cook. *Unsupervised learning of digit recognition using spike-timing-dependent plasticity*. [Online; accessed November 10, 2019]. URL: http://clm.utexas.edu/compjclub/wp-content/uploads/2016/05/diehl2015.pdf.

freeCodeCamp. *freeCodeCamp*. [Online; accessed November 9, 2019]. URL: https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/.

Frontiers in Neuroscience (2016). *Training Deep Spiking Neural Networks Using Backpropagation*. [Online; accessed July 21, 2019]. URL: https://www.frontiersin.org/articles/10.3389/fnins.2016.00508/full.

Hinton, Geoffrey (2012). *Overview of mini-batch gradient descent*. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Izhikevich, Eugene M. (2003). *Simple Model of Spiking Neurons*. URL: https://link.springer.com/chapter/10.1007/3-540-44989-2_101.

Jun Haeng Lee. *Training Deep Spiking Neural Networks using Backpropagation*. [Online; accessed November 10, 2019]. URL: https://arxiv.org/pdf/1608.08782.pdf.

Loihi. [Online; accessed November 23, 2019]. URL: https://www.intel.es/content/www/es/es/research/neuromorphic-computing.html.

Medium (2015). *Everything You Need to Know About Artificial Neural Networks*. [Online; accessed July 21, 2019]. URL: https://medium.com/technology-invention-and-more/everything-you-need-to-know-about-artificial-neural-networks-57fac18245a1.

Michmizos, Guangzhi Tang Arpit Shah Konstantinos P. (2019). *Spiking Neural Network on Neuromorphic Hardware for Energy-Efficient Unidimensional SLAM*. URL: https://arxiv.org/pdf/1903.02504.pdf.

Nengo. [Online; accessed October 25, 2019]. URL: https://www.nengo.ai/.

Neurocomputing (2002). *Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons*. [Online; accessed October 13, 2019]. URL: https://homepages.cwi.nl/~sbohte/publication/backprop.pdf.

Neuromorphic-MNIST. [Online; accessed November 23, 2019]. URL: https://www.garrickorchard.com/datasets/n-mnist.

NN-SVG. *Publication-ready NN-architecture schematics*. [Online; accessed October 25, 2019]. URL: http://alexlenail.me/NN-SVG/index.html.

OpenAI Gym. [Online; accessed November 22, 2019]. URL: https://gym.openai.com/.

Praneeth Namburi. *Simulating neural spike trains*. [Online; accessed October 24, 2019]. URL: https://praneethnamburi.com/2015/02/05/simulating-neural-spike-trains/.

Priyadarshini Panda Kaushik Roy. *Unsupervised Regenerative Learning of Hierarchical Features in Spiking Deep Networks for Object Recognition*. [Online; accessed November 17, 2019]. URL: https://arxiv.org/pdf/1602.01510.pdf.

Prophesee. [Online; accessed November 23, 2019]. URL: https://www.prophesee.ai/.

Prophesee Event Based Vision Explanation. [Online; accessed November 23, 2019]. URL: https://www.prophesee.ai/2019/07/28/event-based-vision-2/.

PubMed (2007). *Solving the distal reward problem through linkage of STDP and dopamine signaling*. [Online; accessed October 16, 2019]. URL: https://www.ncbi.nlm.nih.gov/pubmed/17220510.

PyTorch. [Online; accessed November 22, 2019]. URL: https://pytorch.org/.

Qiang Yu. *A brain-inspired spiking neural network model with temporal encoding and learning*. [Online; accessed November 10, 2019]. URL: https://www.sciencedirect.com/science/article/pii/S0925231214003452.

Renaud Jolivet, Timothy J. and Wulfram Gerstner (2003). *The Spike Response Model*. URL: https://link.springer.com/chapter/10.1007/3-540-44989-2_101.

Ruthvik Vaila John Chiasson. *Deep Convolutional Spiking Neural Networks for Image Classification*. [Online; accessed November 17, 2019]. URL: https://arxiv.org/pdf/1903.12272.pdf.

Scholarpedia. *Spike-timing dependent plasticity*. [Online; accessed October 12, 2019]. URL: http://www.scholarpedia.org/article/Spike-timing_dependent_plasticity.

SpiNNaker. [Online; accessed November 23, 2019]. URL: http://apt.cs.manchester.ac.uk/projects/SpiNNaker/.

SyNAPSE. [Online; accessed November 23, 2019]. URL: https://www.darpa.mil/program/systems-of-neuromorphic-adaptive-plastic-scalable-electronics.

Tensor Processing Unit (TPU). [Online; accessed November 23, 2019]. URL: https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

Tensorflow. [Online; accessed November 22, 2019]. URL: https://www.tensorflow.org/.

The Brian spiking neural network simulator. [Online; accessed November 22, 2019]. URL: http://briansimulator.org/.

The Event-Camera Dataset and Simulator. [Online; accessed November 23, 2019]. URL: https://www.youtube.com/watch?v=bVVBTQ7l36I.

The Neural Simulation Technology Initiative. [Online; accessed November 22, 2019]. URL: https://www.nest-simulator.org/.

Two public domain datasets for spiking neural networks. [Online; accessed November 23, 2019]. URL: https://compneuro.net/posts/2019-spiking-heidelberg-digits/.

Wikipedia. *Hodgkin–Huxley model*. [Online; accessed September 15, 2019]. URL: https://en.wikipedia.org/wiki/HodgkinâĂŞHuxley_model.

Xiao, Han, Kashif Rasul, and Roland Vollgraf (2017). *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. URL: https://arxiv.org/abs/1708.07747.

Yang, Chris Smith Brian McGuire Ting Huang Gary (2006). *The History of Artificial Intelligence*. URL: https://courses.cs.washington.edu/courses/csep590/06au/projects/history-ai.pdf.

Yann LeCun. *The MNIST database of handwritten digits*. [Online; accessed November 10, 2019]. URL: http://yann.lecun.com/exdb/mnist/.

Yongqiang Cao Yang Chen. *Spiking Deep Convolutional Neural Networks for Energy-Efficient Object Recognition*. [Online; accessed November 17, 2019]. URL: https://link.springer.com/article/10.1007/s11263-014-0788-3.