Guidance & Control Implementation with Spiking Neural Networks

A feasibility study

by

Tudor-Mihai Avarvarei

Guidance & Control Implementation with Spiking Neural Networks

A feasibility study

by

Tudor-Mihai Avarvarei

To obtain the degree of Master of Science,
at the Delft University of Technology,
to be defended publicly on Thursday July 18, 2024 at 9:30 AM.

Student number:	4822242
Thesis supervisors:	Dr. Ir. C. de Wagter
	Ir. S. Stroobants
	Ir. R. Ferede
Project Duration:	August, 2023 - July, 2024
Faculty:	Faculty of Aerospace Engineering, Delft

Preface

Over the last decades, humanity has slowly oriented towards an artificial intelligence (AI) revolution with recent advancements promising to change the shape of modern world. This was deeply felt in various job sectors as well as in education with more and more AI tools becoming increasingly available to general public. However, these tools are extremely power hungry requiring great amounts of energy for training and inference. This fact coupled with the recent sustainability concerns begs the question: ”How can we improve the general efficiency of AI?”. Inspiration can be taken from the human brain which is incredible efficient consuming infinitesimal amounts of energy, leading the creation of neuromorphic computing field.

My MSc thesis inspiration started from the previous ideas and wondered how I can combine the neuromorphic computing field with my passion for flight. A missing link was discovered which led to the following topic: ”Perform a feasibility study focused on rapid guidance and control of quadrotors with the help of spiking neural networks”. Even though this work is intended to share my knowledge, it also led to many interesting discoveries on my part. It represented an entertaining journey where I learn numerous facts about human brain and how learning is being performed, facts that I try to sprinkle as often as possible through the report.

Even though I am the sole author of the report, the work realised throughout the 10 months could not have been possible without the help of several people from TU Delft. First and foremost, my thesis supervisors deserve my deepest gratitude for their immeasurable help and guidance. Even though sometimes I felt discouraged by the slow advancement, Christophe and Stein were a constant source of knowledge and incredible suggestions for further work and improvements. Special thanks to Robin who inspired my thesis topic and continuously facilitated my research allowing me to use most of his work such as the dataset as well as the simulation and experiment implementations. Further, I want to send my sincere appreciation to Stavrow and Erik who were always available to answer my endless questions and problems. A big thank you also to Nils and Korneel who were always a source of help during experiments.

But beyond its academic value, the current work also represents the end of an incredible chapter of my life. In 2018, I have moved to Netherlands aspiring to become an aerospace engineer at TU Delft. 6 years and many incredible experiences later, this report marks the transformation. And this could not be possible without the many inspiring friends who travelled continuously with me through this academic journey. To my Romanian aerospace mafioso community, Alex, Andrada, Crina, Nico and Tavi, thank you for providing a home far away from Romania, I could not be here without all your support and motivation throughout these 6 years filled with eternal ups and down. Further, I am extremely grateful to Marek who discovered me in my introvert shell 5 years ago and provided me with unbelievable memories and great laugh ever since. Thank you for all the support and I am looking forward to having new adventures with you wherever that will be.

Even though the last, I want to use the current paragraph to thank the most important people of my life, my family. Vă mulțumesc din tot sufletul pentru suportul nemărginit de-a lungul studiilor. Ați fost continuu alături de mine și nu pot să exprim cât vă sunt de recunoscător pentru asta. Vă iubesc și sper că v-am făcut mândri!

Tudor-Mihai Avarvarei
Delft, July 2024

Contents

List of Figures

List of Tables

Nomenclature

List of Symbols

α	Height of spike activation function	[-]
$\bar{u}_i$	SNN's control output command estimation	[-]
β	Slope of spike activation function	[-]
β_j	Value of the trainable parameter j	[-]
Δt	Timestep duration	$[s]$
λ	L2 regularisation term	[]
λ	Attitude vector	$[rad]$
$\mathbf{\Omega}$	Angular velocity vector	$[rad/s]$
ω	Propeller velocity vector	[RPM]
$\mathbf{F}$	Force vector	$[N]$
$\mathbf{g}$	Gravitational acceleration vector	$[m/s^2]$
$\mathbf{M}$	Moment vector	$[Nm]$
$\mathbf{p}$	Position vector	$[m]$
$\mathbf{u}$	Output command vector	[RPM]
$\mathbf{v}$	Velocity vector	$[m/s]$
$\mathbf{x_0}$	Initial state vector	[-]
$\mathbf{x}$	States vector	[-]
ω_1	Angular velocity of propeller 1	[RPM]
ω_2	Angular velocity of propeller 2	[RPM]
ω_3	Angular velocity of propeller 3	[RPM]
ω_4	Angular velocity of propeller 4	[RPM]
ω_{max}	Maximum angular velocity achievable by propellers	[RPM]
ω_{min}	Minimum angular velocity achievable by propellers	[RPM]
ϕ	Roll angle	$[rad]$
ψ	Yaw angle	$[rad]$
σ	Spike activation function	[-]
τ	Time constant	$[s]$
τ_i	Time constant of membrane current	$[s]$
τ_v	Time constant of membrane potential	$[s]$
θ	Pitch angle	$[rad]$
E	Energy cost function	[RPM]
F_x	Force on x-axis	$[N]$
F_y	Force on y-axis	$[N]$
F_z	Force on z-axis	$[N]$
$F_{ext,z}$	Disturbance force on z-axis	$[N]$
g	Gravitational acceleration	$[m/s^2]$
$gate_x$	Gate's x-position	$[m]$
$gate_y$	Gate's y-position	$[m]$
$gate_z$	Gate's z-position	$[m]$
$gate_{yaw}$	Gate's yaw angle	$[rad]$
I	Moment of inertia vector	$[kgm^2]$
$I_i^{(l)}[t]$	Membrane current of neuron i of layer l at time step t	[-]
I_x	X-axis moment of inertia	$[kgm^2]$
I_y	Y-axis moment of inertia	$[kgm^2]$
I_z	Z-axis moment of inertia	$[kgm^2]$
M_x	Moment around x-axis	$[Nm]$
M_y	Moment around y-axis	$[Nm]$
M_z	Moment around z-axis	$[Nm]$
$M_{ext,x}$	Disturbance moment on x-axis	$[Nm]$
$M_{ext,y}$	Disturbance moment on y-axis	$[Nm]$
$M_{ext,z}$	Disturbance moment on z-axis	$[Nm]$
M_{ext}	Disturbance moment	$[Nm]$
N	Number of timesteps	[-]
N_{neuro}	Number of neurons	[-]
N_{param}	Number of trainable parameters	[-]
p	Roll rate	$[rad/s]$
Q	Transformation matrix between angular velocities and Euler angles	[-]
q	Pitch rate	$[rad/s]$
R	Rotation matrix between world and body reference frames	[-]
r	Yaw rate	$[rad/s]$
S	Target state space	[-]
$S_j^{(l)}[t]$	Membrane potential of neuron j of layer l at time step t	[-]
T	Total simulation time	$[s]$
t_{Bebop}	Time to run a forward function on Bebop quadrotor	$[ms]$

t_{PC}	Time to run a forward function on personal computer [ms]
U	Input state space [-]
u_1	Output command of propeller 1 [RPM]
u_2	Output command of propeller 2 [RPM]
u_3	Output command of propeller 3 [RPM]
u_4	Output command of propeller 4 [RPM]
u_i	Target control output command [-]
$V_i^{(l)}[t]$	Membrane potential of neuron i of layer l at time step t [-]
V_i^{th}	Membrane potential threshold value of neuron i [-]
v_x	Velocity in x-axis [m/s]
v_x^B	Velocity in x-axis in body reference system [m/s]
v_y	Velocity in y-axis [m/s]
v_y^B	Velocity in y-axis in body reference system [m/s]
v_z	Velocity in z-axis [m/s]
v_z^B	Velocity in z-axis in body reference system [m/s]
$W_{ij}^{(l)}$	Weight connecting neurons i and j of layer l [-]
X	Output command space [-]
x	x-position [m]
y	y-position [m]
z	z-position [m]

List of Abbreviations

AdEx	Adaptive Exponential
AI	Artificial Intelligence
ANN	Spiking Neural Network
BCM	Bienenstock-Cooper-Munro
CNN	Convolutional Neural Network
CPG	Central Pattern Generator
CPU	Central Processing Unit
CuBa LIF	Current Based Leaky Integrate-and-Fire
DFBC	Differential-Flatness-Based-Controller
DNN	Deep Neural Network
DOF	Degree Of Freedom
DRAM	Dynamic Random Access Memory
DVS	Dynamic Vision Sensor
ESN	Echo State Network
FOLLOW	Feedback-based Online Local Learning Of Weights
G&CNET	Guidance and Control Network
GAN	Generative Adversarial Network
GNC	Guidance, Navigation and Control
H	Hypothesis
h2g	Hover to Gate
h2h	Hover to Hover
HM2-BP	Hybrid Macro-Micro Level Back-Propagation
IF-SFA	Integrate-and-Fire with Spiking Frequency Adaptation
IMU	Inertial Measurement Unit
INDI	Incremental Non-linear Dynamic Inversion
LIDAR	Light Detection and Ranging
LIF	Leaky Integrate-and-Fire
LSM	Liquid State Machine
LSTM	Long Short-Term Memory
MAC	Multiplier-Accumulator
MAE	Mean Absolute Error
MAV	Micro Air Vehicle
MLP	Multi-Layer Perceptron
MSE	Mean Squared Error
NCD	Normalised Colour Difference
NLIF	Non-Linear Integrate-and-Fire
ODE	Ordinary Differential Equation
PBSNLR	Perceptron-Based Spiking Neuron Learning Rule
PES	Prescribed Error Sensitivity
PI	Proportional–Integral
PID	Proportional–Integral–Derivative
PSNR	Peak Signal Noise Ratio
QIF	Quadratic Integrate-and-Fire
RBF	Radial Basis Function
RC	Resistor-Capacitor
ReLU	Rectified Linear Unit
RL	Reinforcement Learning

RNN	Recursive Neural Network
RPM	Revolutions Per Minute
RQ	Research Question
SL	Supervised Learning
SLAM	Simultaneuous Localisation and Mapping
SNN	Spiking Neural Network
SPAN	Spike Pattern Association Neuron
SPTT	SpikeProp Through Time
STBP	Spatio-Temporal Back-Propagation
STDP	Spike-Timing-Dependent Plasticity
STKLR	Spike Train Kernel Learning Rule
SWAT	Synaptic Weight Association Training
UAV	Unmanned Aerial Vehicles
UL	Unsupervised Learning
VLSI	Very Large Scale Integration

Part I

Scientific Paper

Guidance & Control Implementation with Spiking Neural Networks

Tudor-Mihai Avarvarei[1]

***Abstract*—Quadrotors have continuously leveraged the use of artificial intelligence for navigation and decision-making. Moreover, neuromorphic computing, specifically Spiking Neural Networks (SNNs), is considered as an energy-efficient solution during inference. The current study will analyse the effects of implementing SNNs for mimicking energy optimal guidance and control. To achieve this, population encoding is used and an equivalent of 7-8 spiking neurons per conventional neuron is found to preserve most of the information. The equivalent controller prefers fast adaptation which requires small spiking threshold values and minimal reliance on past information. To improve the controller performance, dataset selection is of utmost importance with a careful trade-off between excessive race track customisation and generalisability being required. The results show that learning is feasible and SNN performance approaches conventional state-of-the-art models trained with multi-layer perceptrons. The current analysis represent an important step towards the rapid guidance and control of ultra-small energy efficient quadrotors.**

***Index Terms*—G&CNET, supervised learning (SL), energy optimal, neuromorphic computing, spiking neural networks (SNN), unmanned aerial vehicles (UAVs)**

I. Introduction

THE rapid evolution of Artificial Intelligence (AI) technologies has brought humanity in a new era characterised by machines enabled with unparalleled data processing capabilities. This surge in the AI sector has profoundly transformed various industries and disciplines. Likewise, Unmanned Aerial Vehicles (UAVs), by their very nature, require a high degree of autonomy, relying on AI for autonomous navigation and decision-making due to the general absence of human operators. This pursuit of autonomy presents a blend of challenges and opportunities. On one hand, it drives the need for sophisticated AI algorithms to ensure the safety and reliability of UAVs operations. On the other hand, the lack of human presence within quadrotors unlocks novel possibilities, granting access to hazardous or otherwise inaccessible environments such as radioactive or confined spaces. The agility and autonomy of quadrotors have facilitated efficient goods transportation, including delivery services [1], [2] and rapid deployment of emergency supplies [3]. This confluence of AI and UAVs promises transformative potential in enhancing operational efficiency, safety, and access to critical areas.

In the pursuit of autonomy, quadrotors have conventionally relied on energy-demanding artificial neural networks (ANNs). These ANNs present challenges for compact UAV designs, necessitating larger batteries that add weight and constrain acceleration and speed. Consequently, energy efficiency has emerged as a critical priority for the quadrotor industry. To address this challenge, the industry is transitioning towards neuromorphic computing, an innovative AI approach that mimics the human brain's efficiency in computation. Spiking neural networks (SNNs), a core component of neuromorphic computing, offer an alternative to conventional ANNs. SNNs encode information not based on signal intensity but through binary events known as spikes and their relative timing or combination, resembling biological neuronal activity. Another advancement within neuromorphic computing is the adoption of event-based cameras, which transmit data only upon changes in pixel intensity. This approach minimises redundant information transfer, aligning with the energy-efficient objectives crucial for small quadrotors.

In recent times, there has been substantial research activity on the convergence of UAVs and neuromorphic computing domains. Neuromorphic computing, known for its efficient computational abilities, has been particularly applied to very small aerial vehicles performing constrained tasks such as hovering, landing, and emulating agile manoeuvres akin to flies [4]–[6]. Advancements in event-based cameras and neuromorphic chips at larger scales have now enabled the feasibility of developing fully neuromorphic quadrotors as well. Early investigations primarily focused on addressing specific challenges. For instance, Pflaum et al. [7] devised a quadrotor stabilisation algorithm, and Mitrokhin et al. [8] demonstrated simulated flight control using data from embedded dynamic vision sensors (DVS), a sub-type of event-based camera technology. Additionally, Paredes-Vallés et al. [9] utilised algorithms derived from optic flow estimation for precision landings, while Landgraf et al. [10] extended these approaches to learning 6-dimensional ego-motion. Furthermore, researchers successfully tackled quadrotor control tasks using neuromorphic computing methodologies, including the use of neuromorphic PID controllers to manage one or multiple degrees of freedom [11]–[14] and employing neuromorphic reinforcement learning techniques, as demonstrated by Jiang et al. [15] for navigation.

However, while the predominant focus of prior research endeavours has been on addressing specific challenges, efforts have also been directed towards the realisation of a fully neuromorphic quadrotor. Notably, several researchers have successfully developed such autonomous UAVs capable of executing tasks such as hovering, landing, and adhering to predefined flight paths [16], [17]. The outcomes have demonstrated significant promise, exhibiting low power consumption and latency, while performing online learning and exceeding

[1]Faculty of Aerospace Engineering, Technological University of Delft, the Netherlands

conventional architectures in certain scenarios. Additionally, in the context of obstacle avoidance, investigations have indicated that fully neuromorphic quadrotors achieve superior energy efficiency (up to 6 times less power consumption) and demonstrate enhanced performance compared to conventional methodologies [18], [19].

In the realm of UAV applications, high-speed drone competitions serve as dynamic environments where AI technologies emerge. Within this area, innovation takes precedence over safety, providing researchers with the latitude to explore cutting-edge technology. Historically, quadrotor guidance, navigation and control (GNC) have predominantly relied on ANNs, achieving remarkable performance that surpasses human capabilities [20]. However, the rationale behind transitioning from traditional ANNs to SNNs holds substantial promise for high-speed drones, driven by two primary factors: the energy demands and the computation time might be substantially curtailed which will alleviate both the battery system and the processing delay. Capitalising on these factors, the quadrotor will benefit from a size reduction. Other theoretical advantages of SNNs include a more dynamic behaviour (leading to concurrent training and performing operational tasks) and a heightened resilience to noise and sensory perturbations (due to the temporal presentation of information) [21].

However, comprehensive research into a fully neuromorphic high-speed drone has not been undertaken. Developing such a quadrotor typically entails two essential components: first, the ability to sense the environment and process its information, and second, the quadrotor GNC. Sensing capabilities have predominantly relied on event-based cameras, extensively studied thus far with research spanning optic flow analysis [9], [22] to high-speed divergence estimation [23]. Conversely, as presented above, the neuromorphic implementation of GNC in quadrotors has primarily utilised simple Proportional-Integral-Derivative (PID) controllers or Spiking Neural Networks (SNNs) capable of executing basic flight manoeuvres and ego-motion, with an emphasis on robustness. However, achieving rapid flight demands a faster operational approach, sacrificing some robustness in favour of strategies closer to optimal control. To the best of the author's knowledge, the implementation of rapid guidance, control and navigation using neuromorphic computing methods remains unexplored to the present day.

Given the novelty of this research field, the present paper will primarily focus on feasibility, specifically examining the potential and methods for implementing rapid GNC using neuromorphic computing in high-speed quadrotor operations. To streamline this investigation, the problem scope has been narrowed by excluding navigation elements, concentrating solely on the guidance and control aspects of high-speed quadrotor flight. Additionally, although the integration of SNNs typically requires specialised neuromorphic hardware to maximise performance, conventional von Neumann hardware will be employed for training and testing the control network. This decision was made to mitigate potential risks and enhance comprehension of the neuromorphic learning process.

With these practicalities in mind, the primary objective of this paper is framed as follows: **to evaluate whether a Spiking Neural Network (SNN) architecture can maintain state-of-the-art performance in guiding and controlling rapid quadrotors**. The benchmark model under consideration was examined by Ferede et al. [24], [25], who utilised artificial neural networks to guide and control a high-speed quadrotor using both supervised and reinforcement learning methods.

In section II, the methodology employed to fulfil the research objective will be elaborated, covering the specifics of the controller training and simulation methodologies, along with the approach for comparing performance against traditional ANNs. Subsequently, the findings obtained will be expounded upon in section III. The results of this research will be critically analysed in section IV, where additional recommendations will be explored. Finally, the conclusions drawn from this study will be summarised and presented in section V.

II. Methodology

A. General SNN structure

A supervised learning approach was chosen to learn information with neural networks, due to its simplicity and ease of understanding. As during the training of the model a trade-off between stable and rapid flight will have to be performed while considering feasibility, a clear understanding of the learning process is required which discards the unsupervised learning method. On the other hand, an interesting approach that was considered by Ferede et al. [25] was the usage of reinforcement learning. However, very few researchers considered applying neuromorphic reinforcement learning as the inherent recurrence found in the neuromorphic neurons could hardly be implemented with a reward mechanism. For this reason, the reinforcement learning algorithm was mostly applied to solving relatively easy problems such as a UAV flying through a window [26] or simple control problems [27]. However, due to the focus of the research on proving the feasibility, this approach will be discarded. Starting from the work of Lu et al. [26], it is nevertheless worth considering reinforcement learning for future research, as it might still achieve better rapid guidance and control performance with neuromorphic computing akin to the artificial neural networks studies performed by Ferede et al. [24], [25].

However, the choice of supervised learning imposes novel problems too. First of all, in artificial neural networks, learning is generally done with the help of back-propagation. The same approach cannot be used with SNNs as the spikes required to transfer information are not continuous and thus non-differentiable. To solve this issue, researchers have proposed several methods, one of them being the Spike-based Back-Propagation algorithm [28], an easy-to-understand approach that achieves satisfactory performance [29]. This algorithm treats the membrane potential of spiking neurons as a continuous differentiable signal, where the discontinuity in the spiking time is regarded as noise. In other words, this method follows the error back-propagation mechanism of traditional ANNs but acts directly on spikes and membrane potentials. In order to perform back-propagation as mentioned previously, gradient descent becomes surrogate gradient descent [30], an algorithm that uses surrogate derivatives to define the derivative of the threshold-triggered firing mechanism. The function chosen to approximate the spike is the arctangent due to its gradient adaptability around the spike coordinates, an approach used successfully by Fang et al. [31]. Thus the derivative used to approximate the spike activation function is described by:

$$\sigma'(x) = \frac{\alpha}{1+\beta * x^2} \tag{1}$$

where α controls the height of the derivative's peak and β the slope of the derivative function.

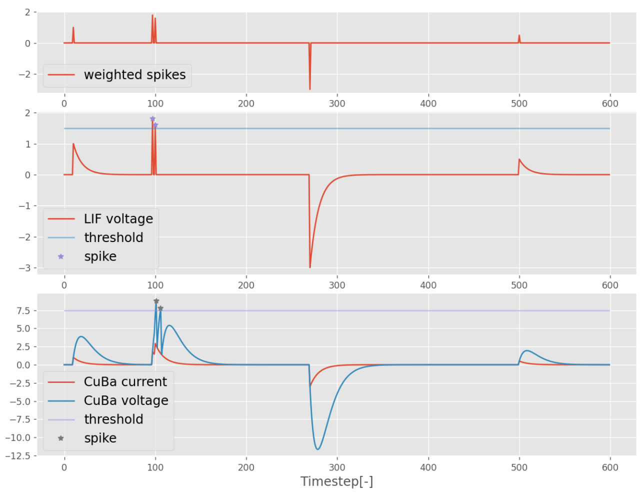


Fig. 1. Visualisation of the LIF and CuBa LIF neuron models and their dynamics when

For the neuron, the literature study found that the Leaky Integrate-and-Fire (LIF) family of neuron models is the fittest for the current research due to their relative simplicity and low computational requirements while greatly mimicking the neuromorphic behaviour [32], [33]. Among the numerous options available, the Current Based Leaky Integrate-and-Fire (CuBa LIF) neuron model was selected for its superior ability to retain past information. Unlike the standard LIF neuron model, the CuBa LIF neuron model does not completely discard past information after a spike is generated [30]. The working principles of this type of neuron can be visualised in Figure 1 where the spiking behaviour is controlled by the membrane potential and current. The discretised system of equations describing the CuBa LIF neuron dynamics are described by [34]:

$$I_i^{(l)}[t] = \tau_i I_i^{(l)}[t-1] + \sum_j W_{ij}^{(l)} S_j^{(l-1)}[t-1] \tag{2}$$

$$V_i^{(l)}[t] = \tau_v V_i^{(l)}[t-1] \cdot (1 - S_i^{(l)}[t-1]) + I_i^{(l)}[t] \tag{3}$$

The presented equations describe how the membrane current and potential of neuron i of layer l get updated at every time step t as a function of previous time steps parameters. In this system of equations, τ_i and τ_v are the time constant of the membrane current and potential respectively which controls how fast the neuron current and potential leaks after receiving information. $V_i^{(l)}[t]$ represents the membrane potential while $I_i^{(l)}[t]$ the membrane current. Otherwise, the membrane potential $V_i^{(l)}[t]$ integrates the spikes received from the presynaptic spike train by summing the product between every previous layer's neuron spike (written in the equation by j) $S_j^{(l-1)}[t-1]$ and their respective weight $W_{ij}^{(l)}$. When the membrane potential $V_i^{(l)}[t]$ exceeds a certain threshold V_i^{th}, the neuron i will cause a spike $S_i^{(l)}[t]$. For the next time step, the membrane potential $V_i^{(l)}[t]$ resets to a value that depends only on the membrane current value $I_i^{(l)}[t]$. After the spike is generated and the membrane potential is reset, the neuron generally enters a refractory period during which new incoming spikes do not affect the membrane potential. However, during the current study, due to the inherent need for fast processing of information, the refractory period was discarded. In this neuron model, multiple parameters are learnable during the back-propagation process. Firstly, controlling how fast the membrane potential leaks, the time constants are bounded between 0 and 1 with the help of the sigmoid function. Similarly, the threshold that causes the neuron to generate a spike is also learnable and bounded only to positive values by the ReLU function.

B. Dataset generation

For training an algorithm with supervised learning, a dataset must be created which, to assure a better comparison with traditional ANN approaches, will be adopted from the works of Ferede et al. [24], [25]. Thus the state and control inputs required for describing the system dynamics will be defined as follows:

$$\boldsymbol{x} = [\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{\lambda}, \boldsymbol{\Omega}, \boldsymbol{\omega}, \boldsymbol{M_{ext}}]^T \quad \boldsymbol{u} = [u_1, u_2, u_3, u_4]^T \tag{4}$$

where $\boldsymbol{p} = [x, y, z]$ and $\boldsymbol{v} = [v_x, v_y, v_z]$ represent the 3-dimensional position and velocity of the drone expressed in the world reference frame. Instead, using the body reference frame, the angular velocity $\boldsymbol{\Omega} = [p, q, r]$ and the Euler angles $\boldsymbol{\lambda} = [\phi, \theta, \psi]$ that describe the drone's orientation are expressed. Lastly, $\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3, \omega_4]$ represents the angular velocity of each of the propellers in RPM while $\boldsymbol{M_{ext}} = [M_{ext,x}, M_{ext,y}, M_{ext,z}]$ is a disturbance moment added to compensate for unmodeled dynamics. The control input $\boldsymbol{u} = [u_1, u_2, u_3, u_4]$ contains the target RPM commands normalised between 0 and 1. During the previous enumeration, body and world reference frames were mentioned which can be visualised in Figure 2. The key difference is that the SNN model learns to process parameters in the body reference frame as it is also expressed in the system dynamics below, while the world reference frame is primarily used for drone flight simulation.

The equations of motion describing the drone's dynamics are described by:

$$\begin{cases} \dot{\boldsymbol{p}} = \boldsymbol{v} \dot{\boldsymbol{v}} = \boldsymbol{g} + R(\boldsymbol{\lambda})\boldsymbol{F} \\ \dot{\boldsymbol{\lambda}} = Q(\boldsymbol{\lambda})\boldsymbol{\Omega} I\dot{\boldsymbol{\Omega}} = -\boldsymbol{\Omega} \times I\boldsymbol{\Omega} + \boldsymbol{M} + \boldsymbol{M_{ext}} \\ \dot{\boldsymbol{\omega}} = ((\omega_{max} - \omega_{min})\boldsymbol{u} + \omega_{min} - \boldsymbol{\omega})/\tau \\ \dot{\boldsymbol{M_{ext}}} = 0\dot{\boldsymbol{\omega}} \end{cases} \tag{5}$$

where $\boldsymbol{g} = [0, 0, g]$ represents the gravitational acceleration, I is the 3-dimensional moment of inertia matrix given by diag(I_x, I_y, I_z), w_{min} and w_{max} are the minimum and

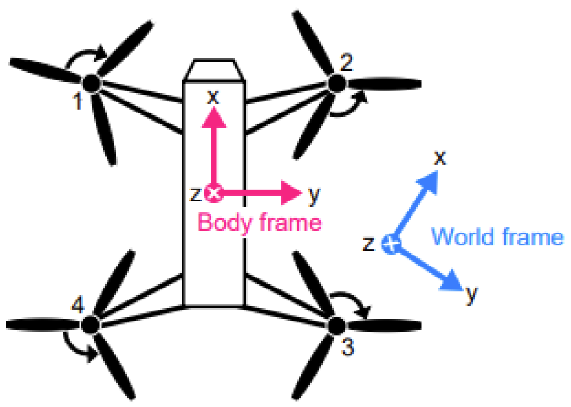


Fig. 2. The reference systems of the quadrotor used throughout the research as well as the numbering order of the propellers [24]

maximum propeller RPM limits and τ is the first order delay parameter of the actuator model. Moreover, $R(\boldsymbol{\lambda})$ is the rotation matrix between world and body reference frames while $Q(\boldsymbol{\lambda})$ represents a transformation matrix between angular velocities and Euler angles. Lastly, $\boldsymbol{F} = [F_x, F_y, F_z]$ is the specific force acting on the quadrotor in the body frame which is modelled using a thrust and drag model based on Svacha et al. [35]:

$$\begin{cases} F_x = -k_x v_x^B \sum_{i=1}^{4} \omega_i \\ F_y = -k_y v_y^B \sum_{i=1}^{4} \omega_i \\ F_z = -k_\omega \sum_{i=1}^{4} \omega_i^2 - k_z v_z^B \sum_{i=1}^{4} \omega_i - k_h ({v_x^B}^2 + {v_y^B}^2) \end{cases} \tag{6}$$

where v_x^B, v_y^B and v_z^B represent the drone's velocities in x, y and z directions of the body reference frame as expressed in Figure 2. Similarly, $\boldsymbol{M} = [M_x, M_y, M_z]$ is the specific moment in the body reference frame acting on the quadrotor modelled, can be expressed using a similar approach as a function of body velocity $\boldsymbol{v}$ and propeller RPM $\boldsymbol{\omega}$:

$$\begin{cases} M_x = -k_p(\omega_1^2 - \omega_2^2 - \omega_3^2 + \omega_4^2) + k_{pv} v_y^B \\ M_y = -k_q(\omega_1^2 + \omega_2^2 - \omega_3^2 - \omega_4^2) + k_{qv} v_x^B \\ M_z = k_{r1}(-\omega_1 + \omega_2 - \omega_3 + \omega_4) + k_{r2}(-\dot{\omega}_1 + \dot{\omega}_2 - \dot{\omega}_3 + \\ \qquad \dot{\omega}_4) - k_{rr} r \end{cases} \tag{7}$$

After comparing the measured moments and specific forces with modelled moments and specific forces, Ferede et al. [24] found significant differences within the quadrotor's moment estimation. To combat the discrepancies, the adaptive environment, considered in the current paper, was developed including disturbance moments M_{ext}. These were generated randomly from a predefined interval as follows: $M_{ext,x}$ and $M_{ext,y}$ between -0.04 and 0.04 and $M_{ext,z}$ between -0.01 and 0.01. Adding these parameters to the guidance and control network showed increased performance during ANN analysis. The constants used throughout the dynamic model expressed above for the Parrot Bebop 1 quadrotor were taken from the work of Ferede et al. [24] and can be seen in Table I.

TABLE I
PARROT BEBOP 1 QUADROTOR PARAMETERS USED BY FEREDE ET AL. [24]

k_x $[RPM^{-1}s^{-1}]$	k_y $[RPM^{-1}s^{-1}]$	k_ω $[RPM^{-2}s^{-2}]$	k_z $[RPM^{-1}s^{-1}]$
1.08e-05	9.65e-06	4.36e-08	2.79e-05
k_h $[m^{-1}]$	I_x $[kgm^2]$	I_y $[kgm^2]$	I_z $[kgm^2]$
6.26e-02	9.06e-04	1.24e-03	2.05e-03
k_p $[RPM^{-2}Nm]$	k_{pv} $[Ns]$	k_q $[RPM^{-2}Nm]$	k_{qv} $[Ns]$
1.41e-09	-7.97e-03	1.22e-09	1.29e-02
k_{r1} $[RPM^{-1}Nm]$	k_{r2} $[RPM^{-1}Nm]$	k_{rr} $[Nms]$	τ $[s]$
2.57e-06	4.11e-07	8.13e-04	0.06

With the model's dynamics described, the dataset was generated using an energy-optimal control problem. This was formulated by Ferede et al. [24] as follows: Given a state space X and output space of admissible control commands U, the goal is to find a control trajectory $\mathbf{u} : [0, T] \to U$ that steers the quadrotor from an initial state $\mathbf{x_0}$ to some target state $S \subset X$ in time T while minimising an energy cost function. This can be formulated as:

$$\begin{cases} \underset{\mathbf{u},T}{\text{minimise}} \;\; E(\mathbf{u}, T) = \int_0^T \|\mathbf{u}(t)\|^2 dt \\ \text{subject to} \;\; \dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u}) \;\; \mathbf{x}(0) - \mathbf{x_0} \;\; \mathbf{x}(T) \in S \end{cases} \tag{8}$$

Minimising the final time T transforms the current problem into a time-optimisation problem and this will be confirmed by a parallel analysis of time and energy parameters below. Moreover, the energy-optimal side of the problem is rooted in the minimisation of the motor command vector $\mathbf{u}$. This leads to the solution $\mathbf{u}$ to exhibit oscillations around the hover thrust approximately halfway between minimum and maximum propeller commands. Using the AMPL [36] modelling language with the SNOPT NLP solver [37], the optimal trajectory $\mathbf{x}(t)$, $\mathbf{u}(t)$ can be computed. This is later discretised with a timestep $\Delta t = T/N$ where N is set to 200.

With these settings, 2 datasets were created where the initial conditions $\mathbf{x_0}$ and final conditions $\mathbf{x}(T)$ are different. The first dataset (hover to hover (h2h)) tries to simulate trajectories where the target is the hover state defined by $\mathbf{x}(T)$, $\mathbf{v}(T)$, $\lambda(T)$, $\boldsymbol{\Omega}(T)$, $\dot{\mathbf{v}}(T)$, $\dot{\boldsymbol{\Omega}}(T)$, $\dot{\omega}(T) = 0$. Moreover to ensure robustness the initial conditions are set randomly around the map following a uniform sampling from the following intervals:

$$\begin{cases} x \in [-5, 5] \;\; y \in [-5, 5] \;\; z \in [-1, 1] \\ v_x \in [-0.5, 0.5] \;\; v_y \in [-0.5, 0.5] \;\; v_z \in [-0.5, 0.5] \\ \phi \in [-2\pi/9, 2\pi/9] \;\; \theta \in [-2\pi/9, 2\pi/9] \;\; \psi \in [-\pi, \pi] \\ p \in [-1, 1] \;\; q \in [-1, 1] \;\; r \in [-1, 1] \\ \omega \in [\omega_{min}, \omega_{max}]^4 \end{cases} \tag{9}$$

The second dataset (hover to gate (h2g)) follows a similar structure but simulates trajectories where the target is simulated as a gate. To do this, a circular racing map was chosen that should be flown in clockwise direction as will be detailed in section III. Thus the target conditions are imposed to be 0 with the exception of the roll angle ψ which is set to $\pi/4$ and the y and x velocity ratio v_y/v_x which is set to $tan(\pi/4)$.

Moreover, the initial conditions are generated similarly, but the intervals were slightly altered to represent the circular racing map more accurately as follows:

$$\begin{cases} x \in [-5,-2] \;\; y \in [-1,1] \;\; z \in [-0.5,0.5] \\ v_x \in [-0.5,5] \;\; v_y \in [-3,3] \;\; v_z \in [-1,1] \\ \phi \in [-2\pi/9, 2\pi/9] \;\; \theta \in [-2\pi/9, 2\pi/9] \;\; \psi \in [-\pi/3, \pi/3] \\ p \in [-1,1] \;\; q \in [-1,1] \;\; r \in [-1,1] \\ \omega \in [\omega_{min}, \omega_{max}]^4 \end{cases} \tag{10}$$

It is important to mention that a second approach was analysed for generating the dataset using the reinforcement learning algorithm of Ferede et al. [25]. However, the resulting dataset striving for time optimal flight was deemed to be too noisy to be learned accurately by an SNN which led to the rejection of this approach. The analysis and the rejection argumentation can be found in appendix B.

C. Controller network

The controller tasked to fly the drone around the circuit will be a spiking neural network. To check the feasibility of such an innovative approach, a trivial network and learning algorithm were chosen. Thus, a feed-forward fully connected structure for the spiking neural network as well as a supervised learning approach were considered for the current study. However, it is important to state that even though a feed-forward architecture is used, recursion exists in the network incorporated in the spiking neurons which save past information. Moreover, to simplify the analysis and have a better ANN-to-SNN comparison, the 3-layered neural network structure created by Ferede et al. [24], [25] was adopted in the current study and can be visualised in Figure 3. The network's training parameters were initialised as follows. The optimiser was set to ADAM [38] due to its fast and accurate search for regression problems, the learning rate was first initialised to 1e-4 with a scheduler that decreases the value when the learning plateau is reached and the loss function considered was the mean squared error calculated as follows:

$$\text{MSE} = \sum_{i=1}^{4} (u_i - \bar{u_i})^2 \tag{11}$$

where u_i is the target control output command and $\bar{u_i}$ is the SNN's control output command prediction.

To operate an SNN, the inputs and outputs have to be encoded as spikes. Various encoding techniques exist, inspired by the human brain, such as rate, temporal or population encoding. The first 2 methods are not desirable as they require long times to process information [32], something unwanted for rapid guidance and control. Instead, population encoding can process instant information and be considered as a first step to transforming the inputs into spikes. The biggest disadvantage of such an approach is that multiple spiking neurons are required to capture the information of one value which might become prohibitive on von Neumann architectures used in the current study. For this reason, a preliminary analysis was performed to quantify the number of spiking neurons necessary to capture most of the information on input parameters which can be found in appendix A. Its conclusion found that 7-8 spiking neurons for each input value can capture the input space information accurately enough, a result that will be confirmed later by the sensitivity analysis performed on the controller's number of spiking neurons per layer of neural network.

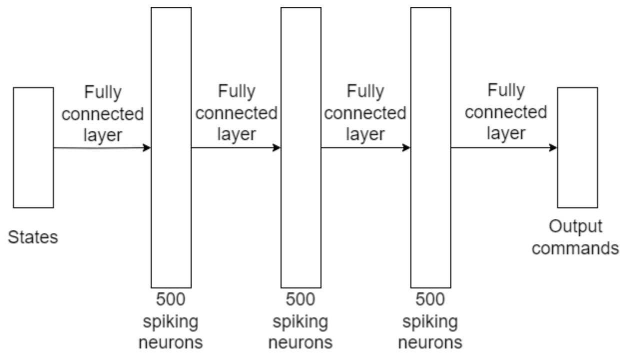


Fig. 3. General structure of the SNN network used throughout the research

However, adding an extra layer for only encoding the dataset adds just another source of error to the controller and it consumes additional computational resources. Leaving the encoding phase directly to the controller allows it to adjust the parameters accordingly such that it can decrease the encoding error directly. Thus, it was decided that the encoding should be performed directly by the controller's SNN which makes the current approach use population encoding as well. In this manner, the weights between the input layer and the first hidden layer as well as the weights between the last hidden layer and the output layer can be learned to summarise the possible encoding and decoding layers respectively. An important contribution that allows for the dismissal of these 2 steps is also represented by the ability of the neuron thresholds to be learned and thus be correlated to the weights such that accurate spiking behaviour of the neurons is achieved.

Regarding the dataset passed for training the network, 10000 different trajectories of $N = 200$ timesteps including state-motor command pairs were generated. 90 % of the dataset was used for training while the remaining 10 % was used for testing the network and thus helping choose the best model. It is also worth mentioning that as opposed to the ANN approach of Ferede et al. [24] that uses only one pair of state-motor commands, the SNN is required to receive sequences of information due to its inherent recurrent nature. For this reason, each trajectory was fed directly to the SNN. The recurrence found in the neurons allows for understanding the current timestep information as a function of the previous timesteps information. However, recurrence also requires an initialisation period when the neurons adapt their current and voltage potential from their resting values which do not allow for high spiking activity during the first timesteps.

III. Results

To verify the performance of the controller, several metrics need to be calculated. For this, a racing map simulator with similar dynamics to the ones presented in subsection II-B was created. The target map that has to be followed by the quadrotor was chosen to be the same as the one studied by Ferede et al. [24], [25] and can be visualised in Figure 4. A similar map was chosen in order to create a fair comparison between neuromorphic and non-neuromorphic controllers. The circuit has the trivial square shape as it is simple enough to showcase the feasibility of such an approach while simulating a high-speed environment. The simulator is created and the equivalent trajectory is obtained by solving an Ordinary Differential Equation (ODE). The timestep Δt is chosen such that the simulation is stable using the Runge-Kutta method. Finding a stable timestep is crucial for further analysis becoming a prerequisite for real experiments which are limited by the Bebop quadrotor's processing frequency of 500 Hz. In the end, to simulate a racing environment, the quadrotor starts at a random gate and runs for a time of 30 s. To adapt the training dataset to the current racing map, the simulator switches the reference frame by $\pi/2$ clockwise when a gate is passed and thus the target is reset. To ensure that a gate is passed, the Euclidean distance between the centre of the gate and the centre of the quadrotor should be smaller than 0.5 m.

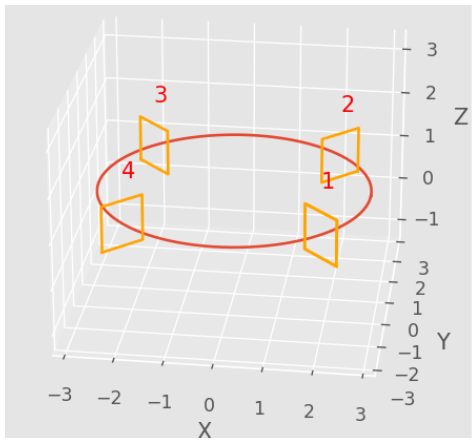


Fig. 4. Visualisation of the racing track used in simulation and experiments

A. *Sensitivity analysis*

Before presenting the results of the best model achieved, a sensitivity analysis was performed illustrating how the best model was chosen based on performance achieved during training. The analysis was done predominantly on the hover to hover (h2h) dataset and started with varying learning rates. The results found in Table II show that the best starting value lies at approximately $1e-3$ for the current training approach with an ADAM optimiser and a scheduler. The same table dictates that a learning rate of $1e-3$ achieves the maximum performance while reducing this value leads to the model converging to a local minimum and increasing it leads to a fast convergence to a sub-optimal solution. Thus the analysis continued with the current finding and analysed further the shape of the surrogate arctangent function used for back-propagation. In Table III it can be observed that the steeper the surrogate function becomes and resembling closer a spike, the better the learning performance becomes with a high value for slope and a low value for height being preferred as expected. However, a limit exists and if the value of the slope increases over the value of 100, the performance of the network starts to drop.

TABLE II
INFLUENCE OF LEARNING RATE ON MSE LOSS

Learning rate	1e-2	1e-3	1e-4
MSE loss [-]	5.397e-3	2.731e-3	6.290e-3

TABLE III
INFLUENCE OF ARCTANGENT SHAPE ON MSE LOSS

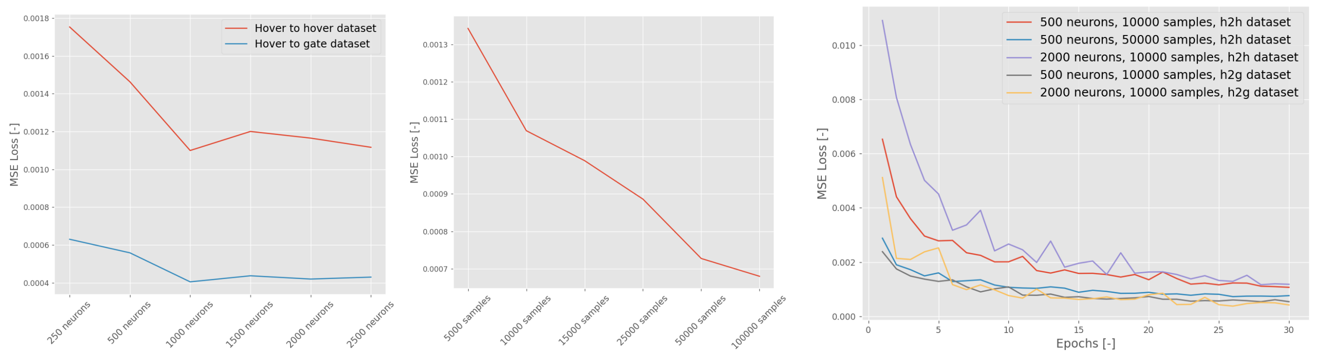
Height \ Slope	**1**	**10**	**100**	**1000**
1	7.934e-3	6.290e-3	5.743e-3	3.377e-2
5	3.167e-2	2.317e-2	9.614e-3	9.735e-3

The analysis continued by verifying the influence of the initial leak and threshold values on the training performance with the results summarised in Table IV. The initial leak and threshold values are generated randomly with the help of a normal distribution function where the mean and variance are passed. In the table below, the sensitivity analysis includes exclusively the mean values while the standard deviation was set directly to 0.5 for all parameters as it was found to have little influence upon the training performance. It is important to note that the leak values presented represent the value passed to the sigmoid function to be bounded. Thus a low leak value of -2, after passed through the sigmoid function is closer to 0 meaning that a great part of the past information is forgotten. This further translates, in a contradictory manner, into a large leak of information at every timestep. Similarly, the opposite happens when a large leak value is passed leading actually to a small leak of information at every timestep. With this in mind, it can be observed from the table that low leak values and thresholds are preferred leading to increased performance. This translates further to the fact that previous information is not crucial for the quadrotor's performance due to the low leak value but also that the quadrotor requires to have fast adaptability due to the low threshold value. In the end, for the simulations and experiments, the initial mean values for leaks and thresholds were set to -1 and -1 respectively.

TABLE IV
INFLUENCE OF THRESHOLD AND LEAK INITIAL VALUE SELECTION ON MSE LOSS

Mean Leak \ Mean Threshold	**0.5**	**1**	**3**
-2	2.453e-3	2.469e-3	6.530e-3
-1	2.297e-3	2.207e-3	9.667e-3
0	2.269e-3	2.416e-3	7.089e-3
1	3.554e-3	2.974e-3	1.189e-2
2	6.569e-3	1.197e-2	1.982e-2

Further, the sensitivity analysis looked at the influence of the number of neurons on the performance of the model. As illustrated in Figure 5a, 2 clear conclusions can be drawn. Firstly, for both datasets used in training, the loss seems to reach the plateau and, equivalently, the best performance at approximately 1000 neurons with a minimal performance



(a) The validation loss versus the number of neurons for the 2 datasets

(b) The validation loss versus the dataset size passed to the model trained with 500 neurons per layer on the hover to hover dataset

(c) The validation loss versus number of epochs for various trained models

Fig. 5. Sensitivity analysis

MSE loss of 1.0995e-3 for the hover to hover (h2h) dataset and of 4.0452e-4 for the hover to gate (h2g) dataset. As can be seen, the models trained on the hover to hover dataset perform worse which led to the discard of this approach. During the simulation phase, several plots were generated which show the clear decrease in performance achieved with the hover to hover dataset. The results and their analysis can be visualised in appendix C.

However, looking again at the plot, it can be observed that adding simply more neurons than the 1000 mark does not help the performance while fewer neurons lead to worse performance. The current result is consistent with the conclusions drawn from appendix A where it is discovered that 7-8 spiking neurons per non-spiking neurons conserve most of the information. Similarly, using the hover to hover dataset, the influence of the dataset size on training performance was analysed and the results can be visualised in Figure 5b. As expected, offering multiple input-output examples improves the performance of the network leading to an MSE loss decrease of the model trained on 500 neurons from 1.3430e-3 with 5000 samples to 6.8967e-4 with the whole dataset of 100000 samples.

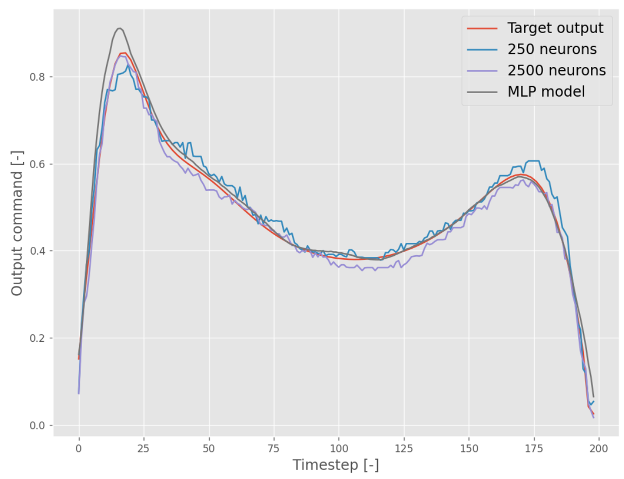
The sensitivity analysis also examined the influence of various hyperparameters on the training time represented in Figure 5c by the number of epochs required to flatten the performance. For most models, the plateau is reached relatively fast with 10 to 15 epochs being required. However, the performance loss sometimes differs based on the influence of the hyperparameter. For example, having more neurons requires a longer time for convergence as more parameters need to be optimised which also leads to a more irregular shape with MSE loss fluctuations during training. Secondly, having a bigger dataset with more variate examples leads to faster convergence and thus fewer epochs are required during training. Last but not least, confirming the results found above, the hover to gate dataset achieves a better performance including a faster convergence too.

B. *Simulation performance*

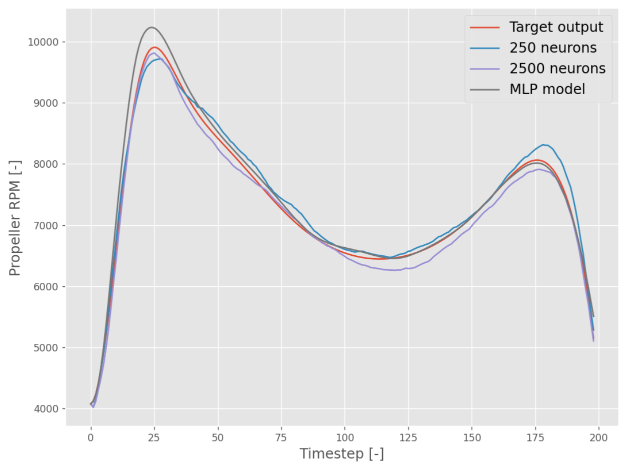
Using the outcomes of the sensitivity analysis presented above, several models could be trained and simulated in a similar environment and racing track as the ones used by Ferede et al. [24] for a fair comparison. As mentioned previously, the models presented throughout the remaining part of the report will focus on the results obtained using the hover to gate (h2g) dataset if not indicated otherwise.

Firstly, the direct output of 2 SNN models with 250 and 2500 neurons per layer on a separate testing dataset was analysed with respect to the target command output as presented in Figure 6a. Moreover, the figure also includes the output generated by the MLP model created by Ferede et al. [24] that uses 120 neurons per layer. Comparing the MLP to SNN, it can be seen that both neural networks types perform well with the latter having bigger error differences from the real output close to the end of the timestep range while the former having a bigger error close to the beginning due to overshooting. Moreover, it can be observed that the SNN model output is less smooth than the MLP model output which can be rooted in the inherent nature of spiking neurons that are not fully continuous and present output is affected by previous information. When comparing the influence of the number of neurons on the SNN model performance, it can be seen that having more neurons leads to a reduction in the error as well as a smoother output is generated. However, it was found that the noisy behaviour caused by the SNN models have little influence, in the end, upon the behaviour of the propeller rotational speed parameters. The generated propeller rotational speed in RPM after being passed through the system dynamics can be visualised in Figure 6b. The plot illustrates that the noisy behaviour specific to SNN models is filtered out by system dynamics and the errors of the propeller rotational speed with respect to the ideal values are consistent with the output command observations.

To quantify the performance of the SNN models for the task of quadrotor racing, it is important to register the lap times. Even though the models were trained on a dataset generated with an energy optimal policy, it was discovered that lap times and energy consumption are heavily correlated.



(a) Comparison of output command predictions



(b) Comparison of generated propeller RPM

Fig. 6. Comparison between 2 SNN (with 3 layers of 250/2500 neurons) and 1 MLP (3 layers of 120 neurons) model performances with respect to target dataset during a testing sample of 200 timesteps

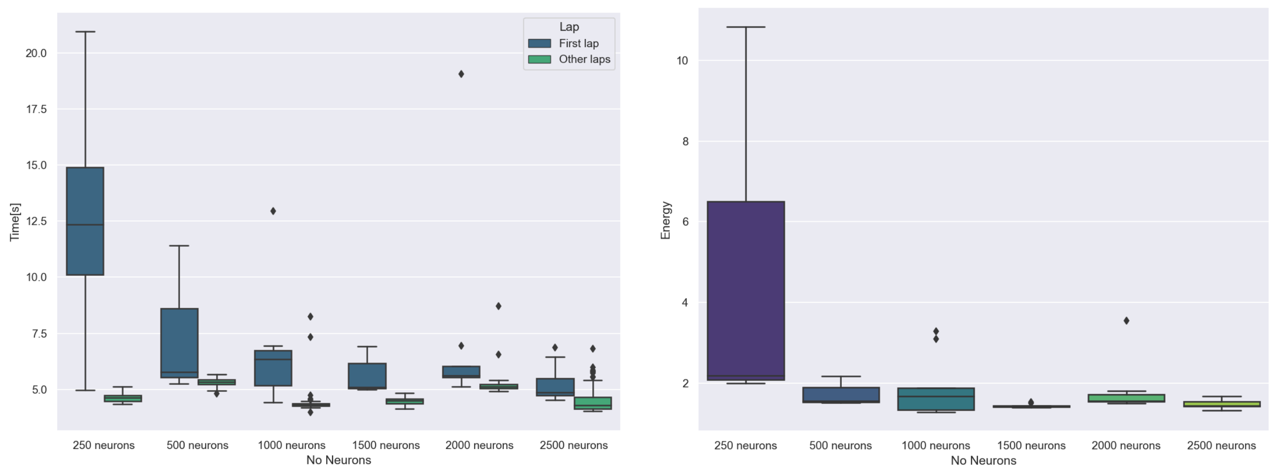
This was expected as to reach a gate with minimum energy consumption, it is important to fly there in the shortest time possible. To confirm the results, Figures 7a and 7b are presented and their clear resemblance is proved with the boxplots showing similar trends. In order to achieve the current results, the simulation involved the flight of the quadrotor across the racing track for 30 seconds for 10 times. With these, Table V was generated that includes the mean and variance of both first and other lap times for SNN and MLP models. Similarly, the mean and variance of the energy consumption is included to confirm the correlation. Moreover, the table includes the stable timestep required to simulate the flight of the quadrotor across the racing track. In order to calculate the energy consumption, the squared output command of all the propellers is summed up over all the timesteps. Then the value is multiplied by the time difference between consecutive timesteps and by the number of gates passed during the 30 seconds simulation for normalising it to the existing dataset.

One clear conclusion of the current analysis is that the SNN models perform generally worse than the MLP models trained by Ferede et al. [24], [25] whether the training method involved supervised learning (SL) or reinforcement learning (RL). However, the performance of the SNN models gets closer and closer to the performance of the SL MLP model as more neurons are being used. The best performance of the SNN model was attained with 1500 spiking neurons per layer that achieves the smallest energy consumption and the shortest time for other laps. The energy consumption of the SNN models flattens at approximately 500 neurons with fluctuations happening as the number of neurons increases. However, the big difference still stands in the times of the first lap which proves to be the most difficult task for the SNN model due to the starting point. Once the model enters the racing optimal track, the quadrotor seems to have fewer problems. One outlier exist when it comes to the influence of the number of neurons on the performance of the model. This is the model trained using 2000 neurons per layer which achieve longer lap times than expected for both first lap and other lap times as well as a higher value for energy consumption. Another important outcome is the variance of the lap times. This value shows how robust the model is to disturbances, a very important attribute of the network that proves the feasibility of this approach and possible implementation to real experiments. Again this decreases generally with the number of neurons in the model with a clear outlier being the model trained with 2000 neurons per layer that has very high variance. The maximum stable timestep serves as an indicator of both model stability and experiment update frequency requirement. A lower value suggests decreased stability, as observed in the model with 250 neurons, which necessitates a very high update frequency ($\approx 157Hz$) and thus a more powerful processor. Conversely, models with 500 or 2500 neurons demonstrate higher maximum stable timesteps, allowing for a reduced update rate ($\approx 49Hz$) while maintaining stability. This inverse relationship between neuron count and the required update frequency underscores the importance of optimising model parameters for both computational efficiency and stability.

Based on the lap times achieved, the racing tracks chosen to be followed by the model could also reveal important observations about the performance of the models. For this reason, several SNN models as well as the MLP model trained with SL were simulated in the environment and can be seen in Figures 8a-8f. The current results can explain the outcomes found in the previous paragraph. Thus, it can be firstly observed that in all SNN models the starting point causes the most problems with the model with 250 neurons starting the simulation clearly in the wrong direction, confirming the very high variance and mean values found in the previous analysis. The deficient starting of the model may be caused by the dataset whose samples do not start from a hovering condition and are created to fly directly through a future gate in a clockwise manner. Moreover, the SNN models are not clearly aiming to fly through the centre of the gate as it is being done by the MLP model. Similarly, the time of the laps can also be

TABLE V
INFLUENCE OF THE NUMBER OF NEURONS ON THE SNN MODEL'S PERFORMANCE QUANTIFIED THROUGH LAP TIMES AND ENERGY CONSUMPTION, COMPARED TO EXISTING LITERATURE MODELS

Model	**First lap times**		**Other laps times**		**Energy consumed**		**Maximum stable timestep** $[ms]$
	Mean$[s]$	**Variance**$[s^2]$	**Mean**$[s]$	**Variance**$[s^2]$	**Mean**$[s]$	**Variance**$[s^2]$	
SL model Ferede et al. [24]	4.334	2.54e-4	4.202	1.92e-3	1.316	2.03e-6	-
RL model Ferede et al. [25]	3.393	4.90e-2	2.749	1.67e-2	2.685	2.15e-2	-
250 neurons h2g dataset	12.661	42.951	4.648	3.35e-2	5.010	25.449	6.349
500 neurons h2g dataset	7.008	5.433	5.317	3.26e-2	1.720	7.41e-2	20.394
1000 neurons h2g dataset	6.608	5.757	4.541	0.731	1.929	0.560	13.340
1500 neurons h2g dataset	5.574	0.688	4.482	2.51e-2	1.444	1.96e-3	14.627
2000 neurons h2g dataset	7.399	22.475	5.300	0.501	1.851	0.484	12.386
2500 neurons h2g dataset	5.233	0.669	4.593	0.456	1.491	1.15e-2	20.505



(a) Boxplots of lap times registered in simulation

(b) Boxplots of energy consumed in simulation

Fig. 7. Statistical analysis of models trained with various numbers of neurons on the hover to gate (h2g) dataset

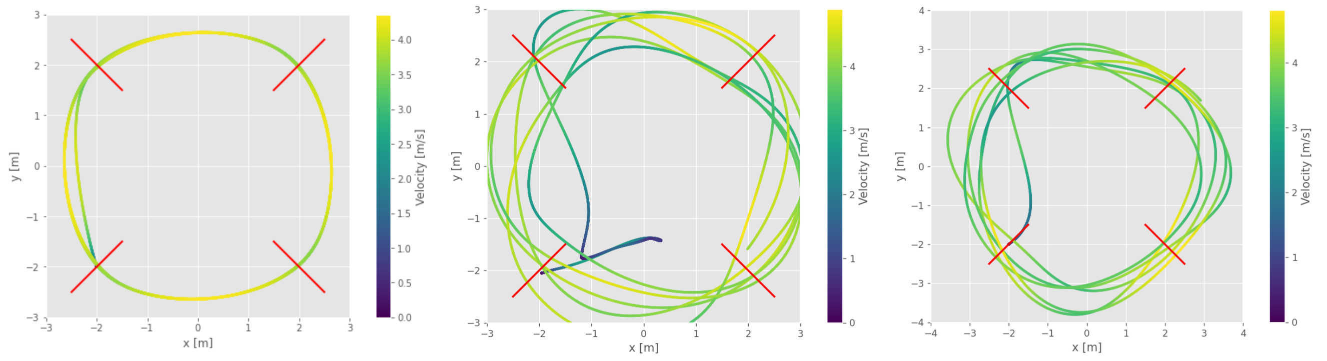
explained by looking at the general racing track patterns. Thus, the SNN model with 2000 neurons chooses to go consistently through the exterior part of the gate which, even though it uses a higher velocity, makes the quadrotor fly slower through the track. This confirms the high energy consumption as well as the the high values for lap times shown previously. On the other hand, one of the most energy and time optimal SNN model, namely the one trained on 2500 neurons learns to fly correct through the interior of the gate which helps it generally fly faster and with less energy. Another outcome is that the outlier lap times observed in Figure 7a are caused by models missing a gate and having to return to pass through it as seen in both Figures 8d and 8e. The best example is the track achieved by the model trained on 1000 neurons. Generally, this model flies through the interior of the circuit but the poor performance caused by missing the gate sometimes generates slower lap times and very high variance. Overall all the SNN models trained on the hover to gate dataset seem to show good adaptability with the quadrotor being able to recover in case it flies off the track.

The last metric analysed during the simulation is the time required to run the created models. Looking at Figure 9, 2 times were registered involving the initialisation and the forward function. The times were calculated on an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz processor. It is important to see that both the initialisation and forward function times show the same exponential behaviour as a function of the number of neurons. However, the forward function time is more crucial as this might make the quadrotor unstable if left too large. As a reference, the time of running the forward function on the MLP model trained with supervised learning is 8.6 ms slightly below the value of 11.2 ms of the SNN model with 250 neurons. Even though, the times presented here are irrelevant as the quadrotor uses a different, less powerful processor. The relation between the number of neurons and the running time could still prove beneficial during experiments. For this reason, this has been theorised in Equation 12 where the time required to run a forward function on PC in ms (t_{PC}) is expressed as a quadratic function of the number of spiking neurons in the model (N_{neuro}).

$$t_{PC} = 2.41\text{e}{-6} \cdot N_{neuro}^2 + 1.25\text{e}{-4} \cdot N_{neuro} + 4.68\text{e}{-2} \quad (12)$$

IV. DISCUSSION

The current study proposed an experiment that analysed the feasibility of building a guidance and control neural network for rapid quadrotors with neuromorphic computing techniques. The analysis involved mimicking the work of Ferede et al. [24], [25] that performed a similar analysis with the help of traditional artificial neural networks. Thus, various spiking neural networks were built and trained to fly around a simple


(a) The performance of the MLP model 500 neurons per layer (b) The performance of the SNN model with 250 neurons per layer (c) The performance of the SNN model with 500 neurons per layer

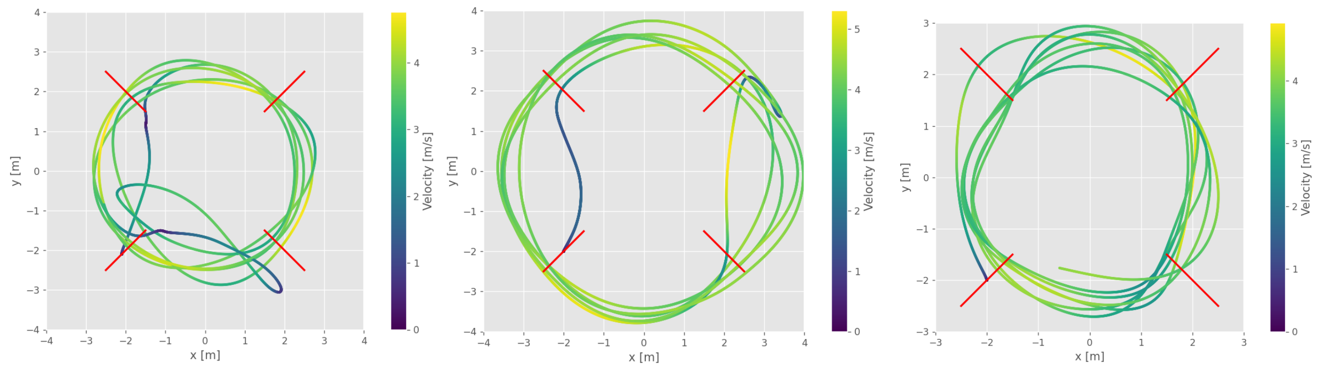

(d) The performance of the SNN model with 1000 neurons per layer (e) The performance of the SNN model with 2000 neurons per layer (f) The performance of the SNN model with 2500 neurons per layer

Fig. 8. Racing track performance of various models trained on the hover to gate dataset

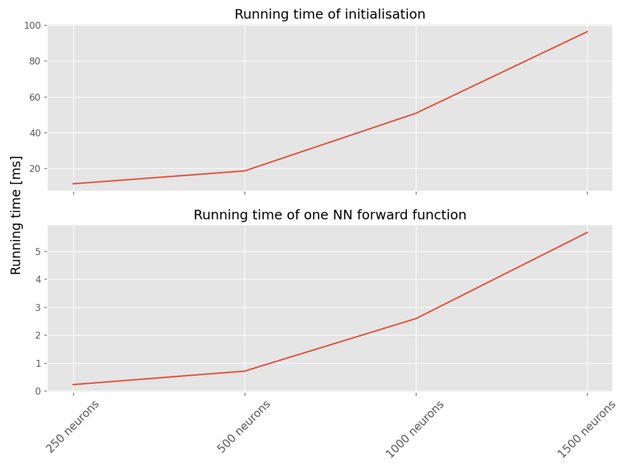


Fig. 9. Running time of both initialisation and forward function of the neural network (NN) models trained with various numbers of neurons on personal computer

track shown in Figure 8 and the results presented previously were obtained with the help of a quadrotor racing simulator environment. The focus of the performed analysis was to verify computationally if such an approach is feasible. However, as positive as the results look, the current study still lacks a clear answer as practical experiments could not be realised with the current quadrotor and SNN controllers. Even though, important conclusions can be drawn from the current study as it will result from the paragraphs below.

The sensitivity analysis performed on the initial values of leaks and thresholds in the model indicates a preference for rapid responses due to low threshold values, while the preference for low leak values implies minimal consideration for previous information. This suggests that recursion is not significantly more beneficial. Specifically, long-term memory proves less useful than just a few previous timesteps, necessitating a small leak value. This can also be confirmed by the good performance achieved by MLP models that learn directly from input-output command pairs without the need for temporal information. Moreover, the preference for low leak values means that fast adaptability is crucial. This can be rooted to the random initialisation done during the dataset generation which demands swift adjustments. Additionally, the need for both rapid control and the ability to respond to disturbances further supports the need for quick adaptation without heavily relying on past timesteps.

Furthermore, conclusions regarding the optimal number of neurons are evident from the encoding procedure described in appendix A but also from the sensitivity analysis performed on the SNN controllers trained with various numbers of spiking neurons in Figure 11. To effectively convert a 32-bit floating point value into spiking neurons, it was discovered that a ratio of 7-8 is necessary to preserve most of the information. This ratio accounts for the elimination of additional factors:

- The precision provided by floating points could potentially convey excessive information to the drone discarded by the usage of spiking neurons;
- The use of recursive relations, specific to spiking neural networks, aids in information preservation with a low number of spiking neurons, as the subsequent values remain close to the previous ones for most parameters;
- Grouping parameters to avoid redundant information transfer as done by population encoding also supports this neuron ratio. For example, velocity could be derived from position data and the timestep;

The quality of the dataset is critical for developing a highly accurate model. For example a dataset that is widespread and generalisable, such as the hover to hover dataset, is valuable as it can be used at later research stages for more complex tasks such as navigation. However, it has been observed that a more specialised dataset, like the hover-to-gate dataset, which is created in such a manner that it teaches the quadrotor how to fly in a circle in a clockwise direction, results in better learning outcomes. The general nature of the hover-to-hover dataset means that fewer specific behaviours are learned, leading to cautious simulation with slower speeds and longer times to travel between waypoints as can be seen in appendix C. Conversely, if the dataset is excessively specialised, as was observed when the reinforcement learning approach was used explained in appendix B, the SNN manages to learn accurately but crashes when minimal disturbances or deviations from ideal track are involved. Therefore, a balance between how adaptable the model has to be and how fast the quadrotor should fly is essential for optimising spiking computation. If the race track is known and navigation is not necessary, a dataset with more constraints imposed may be more suitable as total adaptability is not required, but for general navigation and guidance, a robust dataset is advantageous.

Notable outcomes are observed when analysing the dataset size as well which is done with the hover to hover (h2h) dataset in appendix C. As expected, increasing the number of samples enhances performance. However, the quadrotor's velocity does not significantly improve with a larger dataset. This might be due to the h2h dataset's design, which emphasises on stopping at the next gate rather than passing through at non-zero speeds. A positive of the dataset is that the starting point is not from a standstill, which explains why a model trained on a smaller dataset can still fly but with less precision in reaching the endpoint. This causes the model to be able to adapt to disturbances such as overshooting, undershooting or deviating laterally from the gates. With the current analysis, future research should explore augmenting the h2h dataset to make it easier to learn for complex tracks and navigation tasks. Currently, the only improvement is brought by the augmentation that transforms it into a hover to gate (h2g) dataset, optimised for right turns in a circular track. Reinforcement learning could also be a promising solution for developing a faster model, but achieving robustness and training an SNN through a recurrent RL approach might prove challenging. Another outcome of the analysis is that the SNN controller is more noisy than the MLP one. This result was expected as the output of the SNN controller is calculated as the sum of discrete values. However, it was found that this had no influence upon the smoothness of the propeller rotational speed signal being filtered by the system dynamics.

A practical experiment was also done with the current approach to verify if the SNN model can be flown outside simulation. Thus, the approach of Ferede et al. [24], [25] was chosen to be reproduced and for the experiments, a Parrot Bebop 1 quadrotor was flown whose parameters can be found in Table I. To include the controller, the onboard software of the quadrotor is replaced by the Paparazzi-UAV open-source autopilot [39]. Real-time computation was performed on a Parrot P7 dual-core CPU Cortex A9 processor. The Bebop quadrotor was equipped with an MPU6050 IMU sensor to measure specific force and angular velocity, as well as RPM measurements for each propeller, which were crucial for the control method. Flight tests were conducted in The Cyber-Zoo, a 10x10x7 meter flight arena at TU Delft's Aerospace Engineering faculty, featuring an OptiTrack motion capture system for real-time position and attitude data. An extended Kalman filter was employed to fuse the OptiTrack position and attitude measurements with accelerometer and gyro data from the IMU, enabling accurate estimations of position, velocity, attitude, and IMU biases. A schematic of the experiment can be visualised in Figure 10

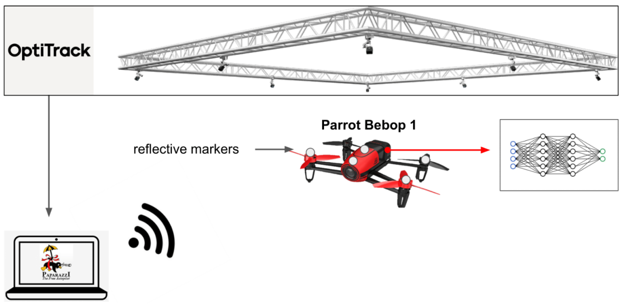


Fig. 10. A schematic of the experiment [25]

However, the experiment revealed that the number of spiking neurons is a great hindrance for the performance if implemented on the Bebop quadrotor. Thus no successful flights were obtained. Instead, the focus was on finding the minimum update frequency that SNN is capable to deliver when implemented on von Neumann architecture. Thus, several SNN models with various numbers of neurons were implemented on the actual quadrotor and the results were summarised in Table VI. With the current findings, implementing the SNN controller on a quadrotor cannot be performed. Looking again at the minimum timestep for stability imposed by the Runge-Kutta criterion during simulations shown in Table V, it is clear that for all 3 models this cannot be achieved in a real experiment. Another important outcome from the experiment is that the running time of the code as a function of the number of neurons is closely related to the finding shown in Figure Figure 9. Thus switching from an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz processor to a Parrot P7 dual-core CPU Cortex A9 processor of the Bebop quadrotor slows down the running time by a factor of approximately 50 - 60 times. The required time to run a forward function on the Bebop quadrotor in ms (t_{Bebop}) was also expressed as a function of the number of neurons (N_{neuro}) in Equation 13.

TABLE VI
MINIMUM UPDATE FREQUENCY OF SNN CONTROLLER WITH VARIOUS NUMBER OF NEURONS ON THE PARROT P7 DUAL-CORE CPU CORTEX A9 PROCESSOR

No neurons	250	500	1000
Minimum update frequency [Hz]	73.1	28.4	7.96
Running time [ms]	13.7	35.2	126

$$t_{Bebop} = 1.28\mathrm{e}-4 \cdot N_{neuro}^2 - 9.6\mathrm{e}-3 \cdot N_{neuro} + 8.13 \quad (13)$$

The current study showed that neuromorphic controllers used for rapid guidance and control are possible. However, future research is still required especially as the capability of such a controller in real environment has not been proven. Thus the following recommendations could be made for future research:

- Investigate better network structures, including different numbers of layers, various neuromorphic neuron types, and alternative back-propagation techniques. The current study did not focus on obtaining the best SNN controller but the focus was on verifying the feasibility of such an approach. This led to the choice of the safest approach with most of the inspiration being drawn from the work of Ferede et al. [24] while the neuromorphic approach was only aimed at the easiest and fastest implementation;
- Explore the possibility of learning the track through reinforcement learning with SNN. While this approach may prove difficult due to the need of creating a reinforcement learning approach to learn a recurrent neural network (given by SNN) [40], it showed very good results by achieving the fastest lap times along the racing track when implemented with traditional artificial neural networks [25];
- For the supervised learning approach, verify other dataset creation methods involving other constraints or adding more parameters, such as force disturbances or the next gate position and yaw. This approach might be the fastest method to increase the performance of the current SNN but the gain in performance is not expected to be great;
- Conduct real experiments with this model, either by moving to a more powerful drone (which necessitates a new training as a new drone model is required) or by analysing new methods to speed up the current network implementation on the Bebop quadrotor. This study will answer the research objective of the current paper possibly proving that controlling a rapid quadrotor with neuromorphic computing is feasible in a real experiment;
- Verify the implementation on neuromorphic hardware. Evaluate the potential time and energy gains, considering that fast information updates of rapid quadrotors may limit the advantages of neuromorphic hardware, which operates only when activated;

V. CONCLUSION

The aim of the current paper was to analyse if training a spiking neural network (SNN) for guidance and control of a rapid quadrotor across a racing track is feasible. Despite achieving a worse performance and the impossibility for implementation on a real experiment, the current study confirms through the simulation results the potential for successful implementation.

Several key findings emerged from the current research. Performance optimisation can be achieved by maintaining lower leak rates as there is minimal need for past information but also by preserving lower threshold values as rapid neuron adaptation is necessary at every timestep. The choice and size of the dataset proved critical: for general guidance and control (and possibly even navigation), a complete and diverse dataset such as the hover to hover dataset is essential, while improving performance can be done by imposing several racing circuit constraints on the dataset as it was performed through the hover to gate dataset. Generally, a larger dataset yields better outcomes. A notable trade-off was observed between maximising the general speed of the quadrotor and robustness (expressed through adaptability of the controller to disturbances), influenced mostly by the number of neurons used in the model. The optimal ratio from floating point to spiking neurons was identified to be 1:7-8 to preserve most of the information. While the output of the SNN remains noisy, increasing the dataset size or neuron count can reduce the noise interval. Running time was found to be exponential proportional to the number of neurons, with relatively proportional variations observed across different processors.

Practical experiment was hindered by the Bebop quadrotor's processor speed, which was insufficient for the current network requirements. But to address the research objective outlined at the beginning of this study, introducing neuromorphic computing generally resulted in poorer performance compared to state-of-the-art artificial neural network models across parameters such as loss, speed, and training/running time. This discrepancy may also be attributed to the usage of von Neumann architecture instead of neuromorphic hardware, which relies on floating-point operations rather than spiking neurons, thus not fully leveraging the advantages of SNNs. Additionally, the discrete calculations inherent in SNNs led to noisier outputs and thus higher loss. Overall, while SNNs show potential for quadrotor racing applications, further advancements in neuromorphic hardware and optimisation techniques are necessary to fully exploit their benefits and achieve stability and performance comparable to traditional guidance and control models.

APPENDIX A
ENCODING THE DATASET

Before the building process of the controller started, a preliminary analysis of the encoding phase was performed. As mentioned, a population encoding approach was chosen due to its fast processing of the information required for racing. However, this approach introduced a novel problem as multiple spiking neurons are needed to represent the information of one floating point neuron which will make the SNN too complex and large to be processed efficiently. For this reason, an analysis of the optimal ratio between spiking neurons and floating point neurons to achieve accurate representation was performed. The analysis involved the reproduction of the input series through an autoencoder structure where the hidden layer is formed out of a variable number of spiking neurons [41]. For analysing the performance, a new dataset was created based on the work of Ferede et al. [25] which learned how to fly the racing track using reinforcement learning. Using this approach a dataset was generated as explained in appendix B, with 24 inputs (adding the next gate 3-dimensional position and yaw as well as a disturbance force on the z-axis) and 4 outputs to be encoded. The dataset is made of 2048 sequences of 2000 timesteps which resembles the flight of the quadrotor for 10 seconds along the circuit.

Using this dataset the autoencoder was trained and the results can be observed in Figure 11 where the 28 parameters (both inputs and outputs) were encoded with various numbers of parameters. As can be seen, the performance flattens at around 200 neurons after a huge drop in MSE loss when fewer neurons are being used. This leads to a ratio of 1 to 7-8 spiking neurons to achieve an accurate representation of the input space using spiking neurons. The research was also extended by analysing this ratio when encoding groups of parameters. The grouping was done such that parameters with common features and interdependencies were encoded together. For example, the 3-dimensional position and 3-dimensional velocity parameters were encoded as one group as well as the 3-dimensional attitude and the 3-dimensional angular velocity together. The analysis showed that the flattening of the curve for encoding 6 floating point values was happening at around 45 neurons where an accurate representation of the input space was achieved, confirming once again the previously mentioned ratio.

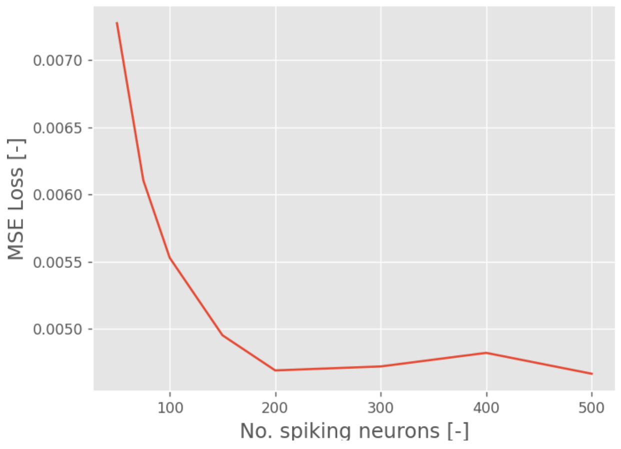


Fig. 11. The influence of the number of spiking neurons on MSE Loss used to encode 28 floating point inputs using a neuromorphic autoencoder

Appendix B
Dataset generated with reinforcement learning

An alternative approach used to generate the dataset was to use the work of Ferede et al. [25] that applied reinforcement learning for flying the racing track time optimally. For this, a neural network using the same structure as in the supervised learning approach [24], was trained with reinforcement learning to fly along the track as fast as possible. With the trained model, a quadrotor was flown in the racing track environment and thus a dataset was generated with a timestep of $0.005s$. In addition, the dataset would record more parameters with the next gate 3-dimensional position and yaw as well as the disturbance force on the z-axis being registered at every timestep. In the end, the simulations were done for 10 seconds (equivalent to 2000 timesteps) and the dataset was given 2048 of such simulations.

Using the generated dataset, an SNN controller was trained but its performance proved to be heavily sub-optimal. This weak performance can be attributed to the noisy signal that can be found in the dataset caused by the inherent aggressiveness of the neural network controller trained with the reinforcement learning approach. This leads to a controller that cannot fly in real life as it lacks adaptability to possible disturbances and deviations from learned optimal track. Consequently, training both the SNN controller and an MLP controller on this dataset proved challenging, resulting in poor performance in both models, as demonstrated by the results shown in Table VII. Analysing Figure 12, it can be revealed that both networks tend to converge towards the average signal value, neglecting the peaks, thereby indicating a failure to capture the noisy behaviour of signal dynamics which is crucial for flying the quadrotor in the end. An interesting outcome from the current analysis is that the SNN model performs slightly worse than the MLP model when the same network structure is passed. In any case, the performance drop is not drastic and can be attributed to the noisy behaviour inherent in SNN that was also observed with the energy optimal dataset in 6a. Moreover, the SNN output command occasionally overlaps the maximum possible value of 1 which requires the imposing of a limit for an accurate functioning of the quadrotor.

TABLE VII
MSE loss of difference models trained on the dataset generated with reinforcement learning approach for all 4 propellers output command

Model \ Propeller	u1	u2	u3	u4
SNN	2.354e-2	4.833e-3	8.648e-3	9.633e-3
MLP	2.254e-2	4.051e-3	8.116e-3	8.372e-3
SNN smooth	1.603e-3	1.602e-3	2.238e-3	1.995e-3

To address the issue of noisy behaviour found in the dataset, a penalisation strategy was implemented within the reward function to reduce the slope of the output command at every timestep, thereby diminishing the quadrotor's speed. This adjustment led to the drone command outputs converging closer towards a hover average of approximately 0.6 as can be visualised in Figure 13. As a result, the SNN was able to follow the output more closely, as evidenced by the results presented in Table VII with the MSE loss being reduced by more than 3 times. However, the signal remained relatively noisy and the training was highly specialised to fly the racing track, ultimately failing to facilitate a successful flight in the

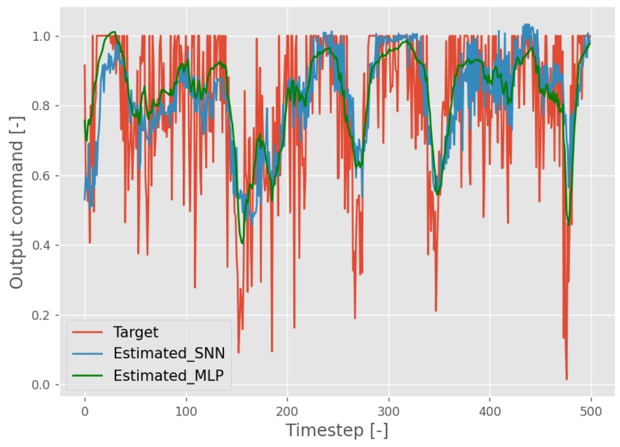


Fig. 12. The MLP and SNN controller estimations trained on the dataset generated with the reinforcement learning approach in comparison with target signal

simulator. Consequently, this approach for dataset generation was abandoned and the focus switched towards the dataset generated with the energy optimal problem. An alternative approach that could be promising involves training the SNN directly using the reinforcement learning approach, though this method presents significant challenges that could potentially yield beneficial outcomes to navigate the racing track with close to optimal speed.

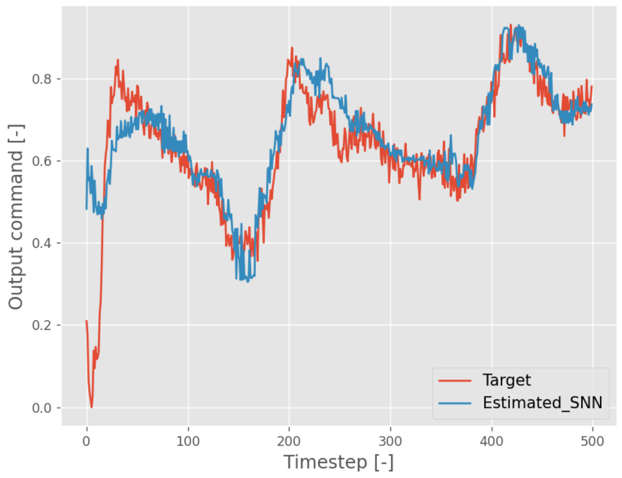


Fig. 13. The SNN controller estimations trained on a smoothed dataset generated with the reinforcement learning approach in comparison with target signal

Appendix C
Hover to Hover Dataset Simulation

The current section shows the simulation results of various models trained on the hover to hover (h2h) dataset. The simulations were done as described in section III and Figure 14 indicate that the dataset is not very effective in achieving the goal of rapid guidance and control of the quadrotor. The models are clearly less robust with numerous unpredictable flying trajectories being incapable of reaching a clear round pattern around the race track. This could be caused by the large number of possibilities offered in the dataset which does not allow the model to learn the characteristics of the racing track. Even though, an important outcome can be observed when comparing the models trained with different dataset sizes. Using a bigger dataset achieves overall higher robustness with the SNN model being able to fly the track in a more predictable manner but at much lower speeds. The lower speeds might be caused simply by the fact that the model does not require to fly faster as its goal is to reach the gate in its hovering state. Thus, the high speed shown for the models trained on less samples are simply an error and are either caused by overshooting or missing a gate.

References

[1] N.T. Chi and L.T. Phong and N.T. Hanh, ”The drone delivery services: An innovative application in an emerging economy”, *The Asian Journal of Shipping and Logistics*, vol. 39, no.2, pp.39-45, January 2023.

[2] X. Li and J. Tupayachi and A. Sharmin and M.M. Ferguson, ”Drone-Aided Delivery Methods, Challenge, and the Future: A Methodological Review”, *Drones*, vol. 7, no. 3, March 2023.

[3] S. Sanz-Martos and M.D. López-Franco and C. Álvarez-García and N. Granero-Moya and J.M. López-Hens and S. Cámara-Anguita and P.L. Pancorbo-Hidalgo and I.M. Comino-Sanz, ”Drone applications for emergency and urgent care: A systematic review”, *Prehospital and Disaster Medicine* , vol. 37, no. 4, pp.502-508 August 2022.

[4] K.Y. Ma and P. Chirarattananon and S.B. Fuller and R.J. Wood, ”Controlled Flight of a Biologically Inspired, Insect-Scale Robot”, *Science*, vol. 340, no. 6132, pp.603-607, May 2013.

[5] T.S. Clawson and S. Ferrari and S.B. Fuller and R.J. Wood, ”Spiking neural network (SNN) control of a flapping insect-scale robot”, *2016 IEEE 55th Conference on Decision and Control (CDC)*, December 2016, pp.3381-3388.

[6] M. Karasek and F.T. Muijres and C. de Wagter and B.D.Q. Remes and G.H.C.E. de Croon, ”A tailless aerial robotic flapper reveals that flies use torque coupling in rapid banked turns.”, *Science*, vol. 361, no. 6407, pp.1089-1094, May 2018.

[7] A. Pflaum and F. Eutermoser, ”Quadrocopter Stabilization using Neuromorphic Embedded Dynamic Vision Sensors (eDVS)”, *Research Practice*, Linprunstr. 06, 80335, München: Technische Universität, at München, November 2012.

[8] A. Mitrokhin and P. Sutor and C. Fermuller and Y. Aloimonos., ”Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception”, *Science Robotics*, vol. 4, no. 30, pp.1-19, May 2019.

[9] F. Paredes-Vallés and K.Y.W. Scheper and G.C.H.E. de Croon, ”Unsupervised Learning of a Hierarchical Spiking Neural Network for Optical Flow Estimation: From Events to Global Motion Perception ”*IEEE Transactions on Pattern Analysis and Machine Intelligence*, July 2018, pp. 2051–2064.

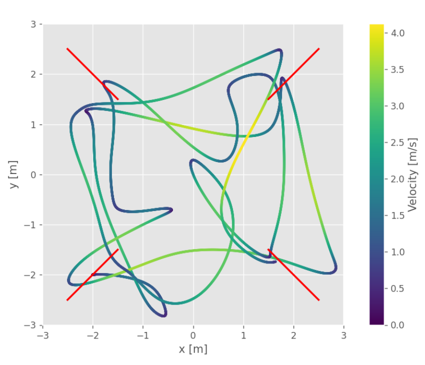
[10] T. Landgraf and B. Wild and T. Ludwig and P. Nowak and L. Helgadottir and B. Daumenlang and P. Breinlinger and M. Nawrot and R. Rojas, *NeuroCopter: Neuromorphic Computation of 6D Ego-Motion of a Quadcopter. Biomimetic and Biohybrid Systems*, Springer Berlin Heidelberg, November 2013, pp. 143–153.

[11] R.K. Stagsted and A. Vitale and J. Binz and A. Renner and L.B. Larsen and Y. Sandamirskaya, ”Towards neuromorphic control: A spiking neural network based PID controller for UAV”, *Robotics: Science and Systems*, July 2020.

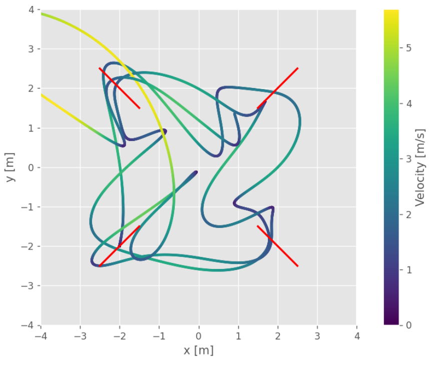
[12] S. Stroobants and J. Dupeyroux and G.C.H.E. de Croon, ”Design and implementation of a parsimonious neuromorphic PID for onboard altitude control for MAVs using neuromorphic processors”, September 2021.

[13] R.K. Stagsted and A. Vitale and A. Renner and L.B. Larsen and A.L. Christensen and Y. Sandamirskaya, ”Event-based PID controller fully realized in neuromorphic hardware: a one DoF study”*2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2020, pp. 10939–10944.

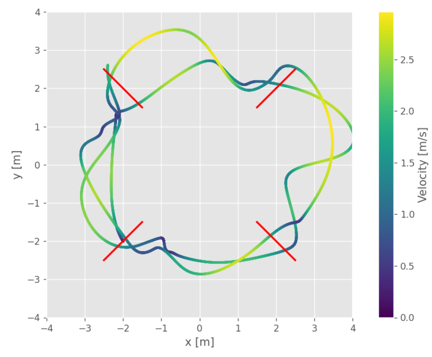
[14] S. Stroobants and C. de Wagter and G.C.H.E. de Croon, ”Neuromorphic Control using Input-Weighted Threshold Adaptation”, April 2023.



(a) The performance of the SNN model with 5000 samples of the whole dataset



(b) The performance of the SNN model with 10000 samples of the whole dataset



(c) The performance of the SNN model with 100000 samples of the whole dataset

Fig. 14. Racing track performance of the SNN models trained on the hover to hover dataset with 500 neurons per layer

[15] J. Jiang and D. Kong and K. Hou and X. Huang and H. Zhuang and Z. Fang, "Neuro-Planner: A 3D Visual Navigation Method for MAV With Depth Camera Based on Neuromorphic Reinforcement Learning"*IEEE Transactions on Vehicular Technology*, vol.72, no.10, September 2023, pp. 12697–12712.

[16] F. Paredes-Vallés and J. Hagenaars and J. Dupeyroux and S. Stroobants and Y. Xu and G.C.H.E. de Croon, "Fully neuromorphic vision and control for autonomous drone flight", March 2023.

[17] A. Vitale and A. Renner and C. Nauer and D. Scaramuzza and Y. Sandamirskaya, "Event-driven Vision and Control for UAVs on a Neuromorphic Chip", *2021 IEEE International Conference on Robotics and Automation (ICRA)*, May 2021, pp. 103-109.

[18] L. Zanatta and A. Di Mauro and F. Barchi and A. Bartolini and L. Benini and A. Acquaviva, "Directly-trained Spiking Neural Networks for Deep Reinforcement Learning: Energy efficient implementation of event-based obstacle avoidance on a neuromorphic accelerator", *Neurocomputing*, vol. 562, December 2023.

[19] L. Salt and G. Indiveri and Y. Sandamirskaya, "Obstacle avoidance with LGMD neuron: Towards a neuromorphic UAV implementation", *IEEE International Symposium on Circuits and System*, May 2017, pp. 1-6.

[20] E. Kaufmann and L. Bauersfeld and L. Loquerico and M. Müller and V. Koltun and D. Scaramuzza, "Champion-level drone racing using deep reinforcement learning", *Nature*, vol. 620, August 2023, pp. 982-987.

[21] M. Pfeiffer and T. Pfeil, "Deep Learning With Spiking Neurons: Opportunities and Challenges", *Frontiers in Neuroscience*, vol. 12, no. 774, October 2018.

[22] J. Dupeyroux and F. Paredes-Vallés and J. Hagenaars and G.C.H.E. de Croon, "Neuromorphic control for optic-flow-based landings of MAVs using the Loihi processor", November 2020.

[23] J. Hagenaars and F. Paredes-Vallés and S. Bohte and G.C.H.E. de Croon, "Evolved Neuromorphic Control for High Speed Divergence-Based Landings of MAVs.", *IEEE Robotics and Automation Letters PP*, July 2020, pp. 1-8.

[24] R. Ferede and G.H.C.E. de Croon and C. de Wagter and D. Izzo, "End-to-end neural network based optimal quadcopter control", *Robotics and Autonomous Systems*, vol. 172, February 2024.

[25] R. Ferede and C. de Wagter and D. Izzo and G.H.C.E. de Croon, "End-to-end Reinforcement Learning for Time-Optimal Quadcopter Flight", November 2023.

[26] J. Lu and X. Wu and S. Cao and X. Wang and H. Yu, "An Implementation of Actor-Critic Algorithm on Spiking Neural Network Using Temporal Coding Method", *Applied Sciences*, vol. 12, no. 20, pp. 10430, October 2022.

[27] S.F. Chevtchenko and Y. Bethi and T.B. Ludermir and S. Afshar, "A Neuromorphic Architecture for Reinforcement Learning from Real-Valued Observations", July 2023.

[28] J.H. Lee and T. Delbruck and M. Pfeiffer, "Training Deep Spiking Neural Networks Using Backpropagation", Frontiers in Neuroscience, vol. 10, pp. 508, September 2016.

[29] J. He and Y. Li and Y. Liu and J. Chen and C. Wang and R. Song and Y. Li, "The development of Spiking Neural Network: A Review", *2022 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 385-390, December 2022.

[30] E.O. Neftci and H. Mostafa and F. Zenke, "Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks," *IEEE Signal Processing Magazine*, vol. 36, no. 6, pp. 51-63, November 2019.

[31] W. Fang and Z. Yu and Y. Chen and T. Huang and T. Masquelier and Y. Tian, "Deep Residual Learning in Spiking Neural Networks", *Advances in Neural Information Processing Systems*, vol. 34, May 2021, pp. 21056-21069.

[32] K. Yamazaki and V. Vo-Ho and D. Bulsara and N. Le, "Spiking Neural Networks and Their Applications: A Review.", *Brain Sciences*, vol. 12, no. 7, June 2022, pp.716-746.

[33] Sanaullah and S. Koravuna and U. Ruckert and T. Jungeblu, "Exploring spiking neural networks: a comprehensive analysis of mathematical models and applications", *Frontiers in Computational Neuroscience*, vol. 17, August 2023.

[34] M.K. Bouanane and D. Cherifi and E. Chicca and L. Khacef, "Impact of spiking neurons leakages and network recurrences on event-based spatio-temporal pattern recognition", *Frontiers in Computational Neuroscience*, vol. 17, November 2023.

[35] J. Svacha and K. Mohta and V.R. Kumar, "Improving quadrotor trajectory tracking by compensating for aerodynamic effects", *2017 International Conference on Unmanned Aircraft Systems*, July 2017, pp. 860-866.

[36] R. Fourer and D.M. Gay and B.W. Kernighan, "A modeling language for mathematical programming", *Management Science*, vol. 36, no. 5, May 1990, pp. 519-554.

[37] P.E. Gill and W. Murray and M.A. Saunders, "SNOPT: An SQP algorithm for large-scale constrained optimization", *SIAM Review*, vol. 47, no. 1, April 2005, pp. 99-131.

[38] D.P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", December 2014.

[39] B. Gati, "Open source autopilot for academic research-the paparazzi system", *2013 American Control Conference*, IEEE, June 2013, 1478-1481.

[40] S. Zheng and L. Qian and P. Li and C. He and X. Qin and X. Li, "An Introductory Review of Spiking Neural Network and Artificial Neural Network: From Biological Intelligence to Artificial Intelligence", April 2022.

[41] H. Kamata and Y. Mukuta and T. Harada, "Fully Spiking Variational Autoencoder", December 2021.

Part II

Literature Study

Abstract

The present literature study aimed to identify a research gap by exploring the intersection of neuromorphic computing and racing drones as well as devising a strategy to address the issue. Both domains show promise for synergy: racing drones demand cutting-edge solutions, while neuromorphic computing offers instead high parallelisation and energy efficiency. Although neuromorphic computing is a novel field, some progress has been made, with applications ranging from specialised hardware to relatively simple problem-solving tasks, including drone control in conjunction with event-based cameras. While most applications demonstrate improved performance compared to artificial neural networks especially focusing on energy consumption, this typically requires specialised neuromorphic hardware. Despite these advancements, the current investigation revealed a significant research gap: the absence of studies on rapid guidance and control with neuromorphic computing, which is crucial for the flight of fully neuromorphic racing drones. Overall, this study seeks to bridge the gap between neuromorphic computing and racing drones, laying the groundwork for future advancements in both fields leading to the following research objective:

The research objective of the current paper is to assess whether a Spiking Neural Network architecture can preserve state-of-the-art performance for rapid quadrotor guidance and control.

Conducted as part of the master's program at Technological University of Delft, this study is limited time constraints and priorities simplicity in implementation due to its pioneering nature. Thus, the focus lies on assessing the feasibility of implementing a spiking neural network (SNN) for rapid drone control, with minimal optimisation and direct adaptation of existing implementations provided by Technological University of Delft's resources, including neuromorphic libraries and simulation software. The project is divided into three phases, each with distinct objectives. The first phase involves building an encoder to accurately represent input parameters with minimal spiking neurons, utilising an autoencoder structure. The second phase focuses on training with supervised learning a spiking neural network for drone control, using datasets created with 2 separate approaches: one through a neural network trained with reinforcement learning approach and the second one trained using energy optimal problem formulation. Finally, the third phase quantifies the performance of the SNN model compared to conventional artificial neural network models through simulation and real-flight data analysis.

In analysing how an SNN could learn the task, several considerations were made. Autoencoders were chosen for the first phase due to their efficient and recursive learning capabilities, being able to understand relations between various parameters effectively. The leaky integrate-and-fire model was selected for neurons due to its great computational efficiency and extensive usage and analysis in the literature while the connections between layers were assigned to simple linear synapses. Spiking neural network will impose the need for a combination between simple feed-forward fully connected neural network and recurrent neural network learning procedures for both the autoencoders as the structure is given by the initial procedure while the latter is given as information is encoded over time within the spiking neurons. Supervised learning was identified as suitable learning approaches due to its focus on optimality and adaptability for recurrency present in spiking neural networks, with surrogate gradient descent chosen for back-propagation due to its simplicity and good performance with non-continuous spiking neural networks.

1 Introduction

The relentless advancement of artificial intelligence (AI) technologies has brought humanity into an era where machines possess the power to process, analyse, and interpret vast amounts of data with unprecedented accuracy and speed. Some are even claiming that due to these rapid digital advancements, society has just entered a new technological revolution—specifically, an informational one[1]. A confluence of factors is considered to have contributed to this remarkable revolution, including the exponential increase in computing power, the accessibility of vast datasets, and groundbreaking advancements in machine learning algorithms.

A defining characteristic of AI's rise is its widespread influence across diverse industries, encompassing a broad spectrum of human activities. In the realm of healthcare, AI has revolutionised diagnostics through the analysis of medical images, outperforming human experts in detecting diseases such as cancer[2] and diabetic retinopathy[3]. Additionally, on a completely different topic, the advent of autonomous vehicles stands as a testament to AI's capacity to redefine transportation and safety standards. Starting gradually with technologies that assist the driver in keeping the lane and speed, car companies such as Tesla and Waymo have exploited AI's capabilities to develop complete self-driving cars, paving the way for a complete driverless transportation[4].

Moreover, the ascendance of artificial intelligence (AI) exhibits a sustained momentum, notably with the surge in popularity of transformers, culminating in substantial advancements across diverse domains, including natural language processing, object recognition, and segmentation[5]. Notably, ChatGPT, a recent prominent manifestation of this progress, exemplifies the evolving landscape of human-machine interaction, prompting a profound reassessment of domains such as education and academia[6]. Concurrently, heightened energy consumption in conventional AI approaches has prompted a gradual pivot towards neuromorphic computing, drawing inspiration from the human brain[7]. The appearance of neuromorphic hardware, epitomised by event-based cameras and specialised chips, has underscored its efficacy, leading to a surge in implementation, particularly in fields such as audio signal processing and robotics[8].

Similarly, drones, as manifestations of autonomous systems, embody the vanguard of automation implementations, consistently integrating cutting-edge technologies to enhance their operational capabilities. Automation in drones is fundamental to their functionality, enabling efficient and precise execution of tasks without direct human intervention. This pursuit of autonomy presents both challenges and opportunities. On one hand, it necessitates the development of advanced AI algorithms to ensure safe and reliable drone operations. Conversely, the absence of human presence within drones unlocks new possibilities, granting access to hazardous or inaccessible environments like radioactive or confined spaces but also in a fast manner. Notably, in the field of agriculture, drones equipped with automated spraying systems have revolutionised crop management, optimising resource utilisation and minimising environmental impact[9]. Furthermore, in the realm of logistics, companies are pioneering drone delivery systems, automating the last-mile delivery process[10].

High-speed drones represent a notable frontier in automation applications too, pushing the boundaries of technological innovation. In competitive racing scenarios, these drones exemplify automation's effectiveness by executing difficult manoeuvres at remarkable speeds, surpassing human capabilities[11]. The implementation of advanced algorithms allows these drones to navigate complex courses with precision, relying on real-time data processing to make split-second decisions. Beyond the realm of entertainment, the high-speed capabilities of drones offer potential applications in surveillance[12] and emergency response[13], where rapid, automated data collection and analysis can be critical. The combination of automation with high-speed drone technology underscores their versatility, emphasising their pioneering role in the continual evolution of automated systems.

In the pursuit of achieving greater autonomy, drones have conventionally leaned on energy-intensive artificial intelligence techniques. However, the incorporation of these, while instrumental in improving decision-making capabilities, imposes great challenges on the design and performance of smaller drones. The resultant need for larger batteries, essential for sustaining these types of computation, leads to increased weight, subsequently negatively affecting acceleration and speed metrics. Consequently, optimising energy efficiency has emerged as a pivotal concern for the fast-expanding high-speed drone industry[14]. Responding to this imperative, the industry is witnessing a notable paradigm shift towards the previously mentioned neuromorphic computing[15] — an approach that emulates the complex functionality of the human brain to facilitate energy-efficient computation. Moreover, this shift towards neuromorphic computing marks as well a transformative step in aligning drone technology with the imperatives of sustainability and enhanced operational performance.

In the realm of computer science, neuromorphic computing constitutes an interdisciplinary field wherein computation draws inspiration from human biological systems, notably the brain and nervous system. This convergence of disciplines, including computer science, biology, mathematics, electronic engineering, and physics, results in the development of bio-inspired computer systems and hardware[16]. The main advantages of neuromorphic computing manifest in a significant reduction in energy demands and computation time, encouraging dynamic behaviour and enhanced resilience to noise and sensory perturbations. However, realising these benefits necessitates the complex integration of a comprehensive neuromorphic architecture, comprising both hardware and software components, a non-trivial task, particularly in the context of high-speed drone applications[17].

Central to neuromorphic computing, spiking neural networks (SNNs) emerge as a compelling branch from traditional artificial neural networks (ANNs), showing intrinsic capability in handling binary information (spikes) and extracting meaningful knowledge from it[18]. Noteworthy advancements in neuromorphic hardware include bio-inspired solutions that markedly reduce energy consumption and accelerate computation. Specialised hardware, exemplified by IBM's TrueNorth chip[19] and Intel's Loihi chip[20], achieves up to 10,000 times greater energy efficiency than conventional microprocessors, utilising power only when essential[1]. Parallel research in neuromorphic vision is realised through event-based cameras, which signal changes in light intensity at a pixel level, achieving processing speeds of up to 1 million frames per second[21]. This exploration underscores the promising trajectory of neuromorphic computing in reshaping computational paradigms and advancing applications across diverse domains.

Evident from the foregoing discussion is a recurrent theme: the nascent stage of neuromorphic technology, coupled with the dynamic and expansive landscape of drone technology, creates a domain at their intersection with a rich research potential. Beyond its technical significance, this literature study forms an integral component of a master's thesis at Technological University of

[1]https://spectrum.ieee.org/how-ibm-got-brainlike-efficiency-from-the-truenorth-chip

Delft. Executed within the Micro Air Vehicle department, the report seeks to identify a well-defined research topic conducive to comprehensive analysis and conclusion within a six-month time frame. It is imperative to note that this study is inspired by a larger faculty project aimed at constructing a fully neuromorphic high-speed drone, with its main objective to compare its performance against a conventional ANN approach. Currently, a concept is characterised by the state-of-the-art achievements of Ferede et al.[22, 23] whose work employs an end-to-end reinforcement learning approach as well as an end-to-end supervised learning approach to adeptly navigate a predefined track.

In synthesis, the literature study attempts to **identify and investigate a research topic integrating neuromorphic technology for rapid guidance and control of a drone, with a focus on feasibility analysis**. The study unfolds in two segments. Firstly, as neuromorphic technologies are novel in this context, the initial phase searches for their exposition and implementation. Commencing with chapter 2, the study outlines existing applications of neuromorphic technologies including hardware that is required to make the whole process brain-like. Given the binary nature of these technologies, the encoding methodologies are expounded in chapter 3. Subsequently, chapter 4 inspects the foundational unit of neuromorphic computing—neurons and their interconnections. Culminating in chapter 5, the structural aspects of neuromorphic computing are delineated, along with potential challenges. The second phase of the study entails selecting and detailing the chosen research topic, articulated in chapter 6. Methodological implementation is explained in chapter 7, with time planning considerations in chapter 8. Finally, a comprehensive conclusion encapsulating the entirety of the report is presented in chapter 9.

2 Existing Neuromorphic Applications

Even though several neuromorphic technologies have been theorised gradually during the 20th century, it wasn't until the last decade that they found some useful applications. For this reason, although this domain is still in its infancy, various applications have emerged which will be analysed in the current chapter. Firstly, the initial try-outs and the inspiration for this novel domain will be detailed in section 2.1 followed up by some general applications which will be presented in section 2.2, mostly focused on the origin of neuromorphic science. Next, it will explain how the neuromorphic hardware revolution was created in section 2.3, including sensors inspired by this novel technology. All things considered, some drone applications of the neuromorphic technology will be analysed in section 2.4.

2.1 History and Inspiration

The idea of using neuromorphic inspiration to solve problems is not a recent solution. Similarly, computer hardware, even from the very beginning, was mostly influenced by the thinking process of humans. Based on this information, Alan Turing proposed in one of his papers[24] a simple device that could be capable of performing any conceivable mathematical computation if it were given as an algorithm. This later evolved into today's computing industry where computers are built on the base of von Neumann's architecture[25]. However, in addition to the work leading to the digital computer, Turing foresaw the evolution of neuron-like computing by describing a machine that consists of artificial neurons connected in any pattern. The architecture would include modifier devices which could be configured to pass or destroy a signal similar to how a spiking neural network works[26].

Following Turing's work, researchers began to show greater interest and commitment to the field of artificial learning. Thus, Hebb[27] proposed synaptic plasticity as a mechanism for learning while Rosenblatt[28] theorised the principles of connectionism and the perceptron. The perceptron invention led to enthusiasm in the field and, with the subsequent discovery of error back-propagation[29] and recurrent neural networks[30] in the 1980s, the computational learning revolution began.

With the increasing importance of traditional artificial networks, interest in neuromorphic technology started to fade. However, some important results were still developed. One such discovery was ADALINE, a physical device that uses electro-chemical plating of carbon rods to emulate the synaptic elements[31]. Thus, this work became the first integration of memristive-like elements to emulate a learning system in computers. Inspired by this work, Chua postulated a decade later that the missing link in realising a fully neuromorphic computer is a memristor, an element whose memory resistance depends on the integral of the input applied to the terminals[32].

However, these discoveries did not inspire the necessity of switching computers to a fully neuromorphic approach. Thus, for the following decades, the studies included theory focused on computers using the von Neumann architecture such as synaptic modification[33] or VLSI (Very large scale integration) systems[34]. Only recently the attention has switched again to neuromorphic architectures by starting to produce practically some of the ideas presented in the previous research. In 2006, researchers at Georgia Tech released a field programmable neural array, marking the initial development of arrays comprising floating gate transistors. These transistors enabled the manipulation of charge on the gates to emulate the channel-ion traits observed in neurons within the brain[35]. In 2008, HP Laboratories started the production of the memristor, theorised by Chua, and explored its use as synapses in neuromorphic circuits[36]. Further in time, in 2011, a team from Massachusetts Institute of Technology created a chip that simulates the analogue, ion-based communication between two neurons using 400 transistors[37].

With these studies, research in neuromorphic engineering started to grow exponentially. But the current research is not limited to reproducing perfectly the brain including all of its functions. Instead, it focuses on extracting the structure and operations of the brain to be used computationally both for a better understanding of biology as well as for generating more energy-efficient algorithms. Thus, research has been preponderantly focused on replicating biological computations in an analogue manner but also on the role of neurons in cognition[38].

The main source of inspiration for neuromorphic engineering remains the human brain and its intricacy. To understand where this inspiration comes from, it will be detailed how the brain functions with the help of Figure 2.1. Firstly, the neuron is the fundamental atomic unit of the brain and consists of 3 parts: dendrites, soma and axons. Dendrites are structured in a tree-like distribution and transmit the received input signals from pre-synaptic neurons to the soma. With this input information, the soma adapts its membrane potential as a response to all data from the dendrites. When the membrane potential reaches a certain threshold, an action potential is generated (a discrete spike) which is sent further to the axon. With the help of the axon, the information is passed to the further neurons. It is important to mention that this was a generalisation and not all the neurons are the same with most of them being polarised cells which help the brain perform specialised functions.

The connection between neurons is done with synapses which are the spaces between axon terminals and dendrites. These send a biological signal which typically consists of either chemical or electrical signals. In the chemical case, the signal releases chemical neurotransmitters that bind to receptors of the postsynaptic neuron, which then cause the propagation of an electrical signal. In the electrical case, the neurons are connected by gap junctions which pass the current having the great benefit of quicker transmission times. Synaptic transmission has two basic forms: excitation and inhibition which either prompts the downstream neuron to fire or to further its membrane potential from the threshold. The neuron continuously receives these types of transmission and, when this input sum reaches or exceeds the threshold, it excites an action potential, otherwise, it remains silent.

The membrane potential is the electrical potential of the neuron and the key to the spike generation that is regulated with the help of substances inside and outside the cell membrane such as charged ions and molecules. To understand this process, a brief example will show how a spike is generated. The membrane potential is initially resting at the minimum threshold of the neuron. When the neuron is stimulated, voltage-gated sodium channels will open, causing the membrane potential to rise rapidly. This happens until a threshold is reached. After this, if the membrane potential keeps rising, sodium channels deactivate and potassium channels open. Then, the membrane potential decreases rapidly because of the sodium channels' inactivation and the opening of potassium channels, which gradually return to the resting potential with the closing of potassium channels.[39, 40]

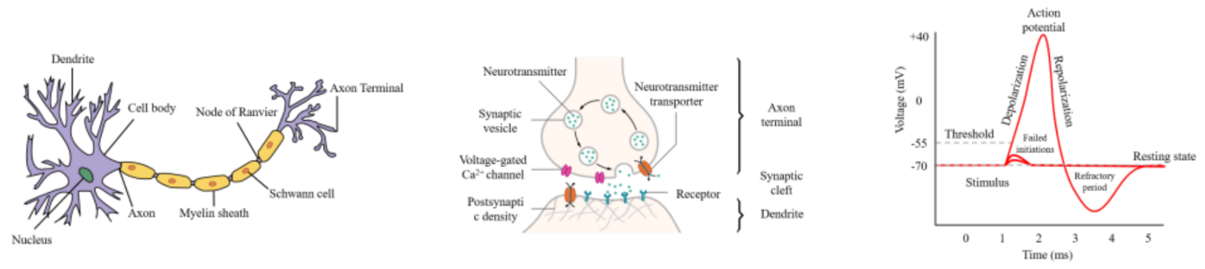


Figure 2.1: The graphical explanation of how the brain functions; on the left, a schematic presentation of the neuron; in the middle, the connection between 2 neurons, called synapse; on the right, the membrane potential phases along with the spike generation [40]

2.2 General Applications

The applications of neuromorphic engineering are highly diverse, with no domains yet discovered where this innovative type of computing and devices clearly excel. Initially, the focus of applications has primarily centred on replicating a wide range of small learning tasks already performed by traditional ANNs. However, only recently, with the rapid evolution of neuromorphic hardware, attention has shifted towards more challenging tasks by integrating various neuromorphic architectures and establishing more comprehensive networks.

But to begin with, a great survey of the first applications was performed by Schuman et al.[7]. As mentioned in the survey, implementing neuromorphic networks to learn these types of applications on neuromorphic hardware may lead to lower power consumption and faster computation than on a von Neumann architecture. However, as many of these applications do not necessarily require any of those characteristics due to their small size, researchers have not thoroughly tested the previous performance allegations but only focused on the feasibility of neuromorphic technologies.

With this in mind, Schuman et al.[7] summarised the existing applications of neuromorphic engineering in 2017 which can be visualised in Figure 2.2. Most of the applications were focused on reproducing the existing capabilities of traditional ANNs without explicitly trying to surpass their results. Thus most of the applications focused on image classification and processing performing relatively easy tasks such as edge detection, image filtering, image compression or feature extraction on common datasets such as MNIST or CIFAR-10. Other common applications include pattern recognition capabilities for tasks such as data classification and anomaly detection.

Similarly, researchers have looked for reproducing closely and biologically accurate sensory systems such as visual and sound capabilities. Due to the infancy of this realm, a big part of the applications focused on basic benchmark tests including common problems such as N-bit parity, two spirals or simple logic gates (AND, NOR, XOR). Last but not least, some applications focused on solving robotics and control problems as very small, power-efficient and real-time performance systems are often required in these domains. These include easy motor control tasks such as learning a particular behaviour, joint control, target following, autonomous navigation, cart-pole problem or inverted pendulum but also classical games such as PACMAN or Pong.

However, in recent years, the difficulty of these applications increased, including in some cases the implementation on specialised hardware. For example, Zheng and Mazumder created 2 hardware architectures that can implement supervised learning tasks. One is relatively close to the conventional von Neumann architecture while the other one focuses on memristors and phase-change memory[41]. Another focus of recent neuromorphic research was on mapping a pre-trained DNN to SNN which led to several applications. Tasks such as keyword spotting,

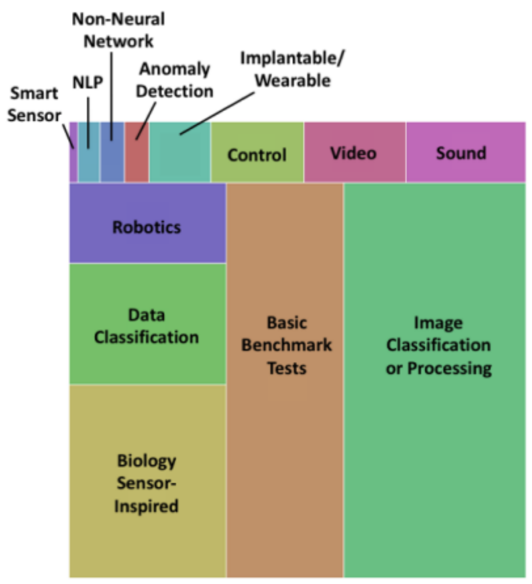


Figure 2.2: Visual breakdown of applications to which neuromorphic systems have been applied; the size of the boxes is directly proportional to the number of works in which a neuromorphic system was developed for that application[7]

medical image analysis and object detection have been demonstrated to run efficiently on existing platforms such as Intel's Loihi[20] and IBM's TrueNorth[19]. Inspired by the human brain, several neuromorphic applications were found in the realm of medicine. For example, clustering mechanisms have been used as a spike sorter in brain-machine interfaces and functional magnetic resonance imaging signals have been used in applications such as sleep state detection and prosthetic controllers[8].

Recently, a lot of attention has also been focused towards robotics and control applications. Evolutionary algorithms have been successfully applied to control applications such as autonomous robot navigation or for various games using the LIDAR techniques and SNNs[8]. These include encoding sensory information into distributed maps and generating motor commands, head direction, reference frame transformation, distance mapping, observation likelihood, Bayesian inference and other simultaneous localisation and mapping (SLAM) implementations. Other applications focus on a more biologically accurate approach and try to reproduce the central pattern generators (CPGs) for robot control. Using neuromorphic hardware they help a hexapod robot to learn to walk, but also to simulate repetitive motions inspired by a human's gait or the swimming of lampreys. Moreover, inspired by classic control, PID controllers have also been theorised and implemented in the neuromorphic domain[18].

2.3 Neuromorphic Hardware

Neuromorphic hardware refers to the specialised computing architectures that are completely based on the neural network structure. For this reason, the behaviour of neurons is emulated by dedicated processing units while the physical interconnections are done in a web-like manner and facilitate the rapid exchange of information. This concept is fully inspired by the human brain, where biological neurons and synapses work similarly to the hardware presented above.

Thus neuromorphic computers can be viewed currently as non-von Neumann computers whose structure and function are inspired by brains. In a neuromorphic computer, memory and processing are ruled by the neurons and the synapses while in a Von Neumann computer, these 2 are separate units. For this reason, programs in neuromorphic computers are defined using the structure of the neural network and its parameters, rather than by explicit instructions. In addition, von Neumann computers use numerical values (as binary values) to express information. In neuromorphic computers, information is received and output as spikes, encoded in the associated time at which they occur, their magnitude and their shape[8]. Their architectures and their comparison can be visualised in Figure 2.3 leading to some advantages and disadvantages for neuromorphic hardware:

- **Advantages:**
  - Highly parallel operation due to all of the neurons and synapses potentially working simultaneously;
  - Collocated processing and memory which helps mitigate von Neumann's bottleneck regarding the separation of processor and memory, which causes a slowdown in the maximum throughput that can be achieved; this collocation also helps avoid data accesses from main memory, which consumes a considerable amount of energy in conventional computing systems;
  - Event-driven computation which means that neurons and synapses only perform work when there are spikes to process; this allows for extremely efficient process and low power consumption;
  - Inherent scalability by combining additional neuromorphic chips which simply increases the number of neurons and synapses; in the end, the newly created chip can be treated as a single large neuromorphic implementation to run larger and larger networks;
- **Disadvantages:**
  - Difficult to program it due to its novelty in the market; Having a completely different architecture than von Neumann computers, neuromorphic hardware requires all the basic programming to be redone which makes complex solvers hardly implementable;
  - Hard to interpret and understand due to having everything encoded in spikes, neurons and synapses including the structure of the network, memory and processing logic;

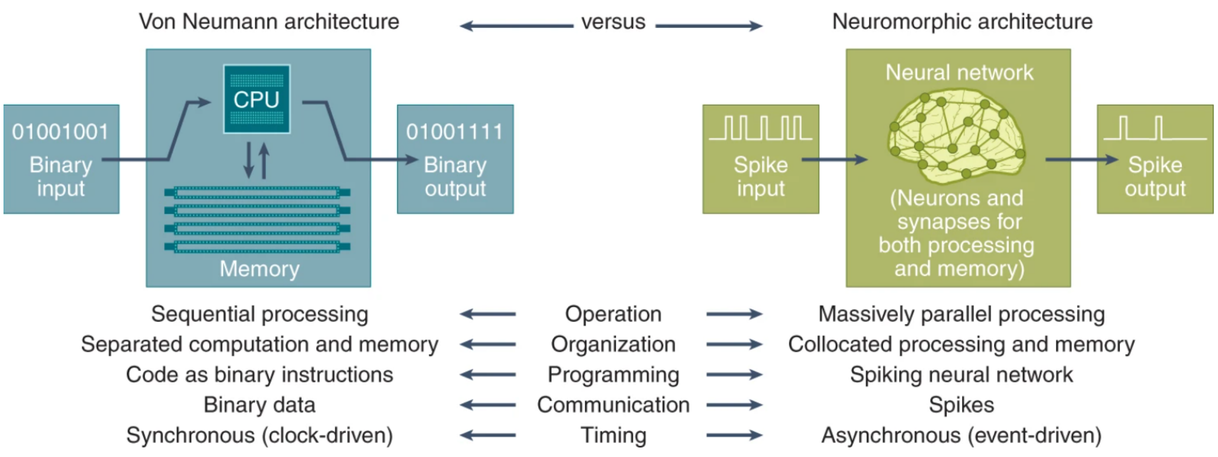

Von Neumann architecture
versus
Neuromorphic architecture
01001001
Binary input
CPU
Memory
01001111
Binary output
Spike input
Neural network
(Neurons and synapses for both processing and memory)
Spike output
Sequential processing
Separated computation and memory
Code as binary instructions
Binary data
Synchronous (clock-driven)
Operation
Organization
Programming
Communication
Timing
Massively parallel processing
Collocated processing and memory
Spiking neural network
Spikes
Asynchronous (event-driven)

Figure 2.3: The fundamental differences between von Neumann and neuromorphic architectures[8]

With these differences in mind, it is important to note that during the last decade, neuromorphic engineering-focused to a great extent on specialised hardware. Thus, several neuromorphic hardware became recently available for use with many more projects under development. A summary of the most important ones include:

- Loihi[20]: by far, the most common hardware for SNNs, it is a neuromorphic many-core processor that supports on-chip learning, making it a viable option for both inference and deployment; it also has 128 cores, simulates up to 131,072 neurons with 130,000,000 synapses possible[42];
- SpiNNaker[43]: it is an open-access cloud that contains 18 on-chip dynamic random access memory (DRAM) processor cores with approximately 1000 simulated neurons and 182 MB off-chip DRAM where synaptic parameters are stored; it can be used for simulation and testing of different applications that do not require on-site implementations[42];
- TrueNorth[19]: its computational core contains 256 neurons and a 256 $\times$ 256 size synaptic array; it uses both synchronous and asynchronous event-driven calculations, which helps the chip consume only 100 mW when it simulates an SNN with millions of neurons[44];
- BrainScaleS[45]: it is a hybrid analogue neuromorphic supercomputer which contains 352 chips with 512 spiking neurons for each chip; the system consumes approximately 1 kW[44];
- ODIN: Acronym for optimised digital neuromorphic processor, it allows the use of slightly more complex neuron models[8];
- Tianjic[46]: it is a platform that supports both neuromorphic SNNs and traditional ANNs for a wider class of applications[8];
- DYNAPS[47]: it is an offline learning mixed platform with each circuit board containing 9 chips each having 4 computing cores with 256 neurons per core[44];
- Neurogrid[48]: it contains 16 chips in the circuit board, each with 256 $\times$ 256 neurons, that are connected by a tree routing network, consuming in the end 3.1 W[44];

All the listed examples are silicon-based; however, the research trend focuses on developing new types of materials for neuromorphic implementations, such as phase-change, ferroelectric, non-filamentary, topological insulators or channel-doped bio-membranes. Similarly, research focuses their attention on memristors as the fundamental device to have resistive memory to collocate processing and memory, but other types of devices have also been used to implement neuromorphic computers, including optoelectronic devices[8].

So, neuromorphic computing hardware is a critical future technology becoming increasingly important and relevant in modern research. This is mostly caused by the von Neumann bottleneck which predicts that computer performance will soon reach its limits. However, a consistent problem for neuromorphic hardware remains in the non-cohesive nature of the research community. To achieve the actual results of neuromorphic hardware, all the architecture must become neuromorphic including computing and sensors such that steps are not lost for encoding and decoding of data[49].

Neuromorphic sensors refer mostly to the hardware which can be excited or inhibited by external factors and, transmit this information in the form of spikes. One domain where this was realised until now is haptic sensitivity. Thus haptic behaviour was summarised by Zeng et al.[50] which names various studies based on pressure sensory neurons or on monitoring toxic chemicals. Their paper focuses on the chemical part of sense, naming various devices that were implemented using this approach. One of them is Sarkar et al.[51] who created a neuron that mimics the biology of sensing by simulating accurately its chemical reactions.

Even though haptic domain research is mostly centred around chemistry and is still in its infancy, the same cannot be said about visual sensing. These have been facilitated by event-based cameras which, instead of capturing whole images at a fixed rate, measure the brightness changes for each pixel and output these as a stream of events that encode their time, location and sign[52, 53]. Its advantages vary from high temporal resolution (in the order of μs to very high dynamic range (140 dB versus 60 dB), low power consumption, and high pixel bandwidth (on the order of kHz) leading to a motion blur reduction. These advantages come with the drawback of novelty with new methods being required to process the unconventional output of these sensors[52]. Until today, event-based cameras were applied to various visual tasks including feature detection, optical flow, stereo vision, 3D monocular reconstruction, pose estimation, SLAM etc.[52]. With the help of neuromorphic computing, the applications extended to control tasks, creating drone applications that solve visual problems with the help of event-based cameras as will follow in the following section.

2.4 Drone Applications

After a general presentation of neuromorphic computing, the current section will centre on applications done in the drone realm. Drones were selected due to their more relaxed requirements for implementation but also due to their small size which requires low power consumption, neuromorphic computing promising gains in this field. For other aviation realms, neuromorphic did not get too prevalent mostly due to its infancy. This causes heavy safety concerns not alleviated by the possible size (and implicitly weight) gains. However, these could be implemented in the future according to Parlevliet et al.[54], first starting with inessential and easy tasks such as an altitude controller for an autonomous open-source blimp[55].

The very first drone applications were implemented on very small flyers to accurately simulate the flight of insects. Trying to create such a small machine capable of flying requires computational power but the traditional power-hungry ANNs along with von Neumann architecture require a big processor and memory space which cannot be physically implemented within the current size. Thus, for both computational power and energy reasons, the focus has been switched to neuromorphic computing. This led to the creation of a biologically inspired, insect-scale robot, weighing 80 milligrams and being able to hover and fly simple tracks[56]. Further, the research increased the difficulty of the performed tasks including the landing phase of the RoboBee[57] or mimicking the rapid escape manoeuvres of flies[58]. Lastly, McGuire et al.[59] designed the swarming flight of insect-sized MAVs.

Following this success with small flyers, drone researchers started to adopt neuromorphic computing for performing increasingly difficult tasks. Initially, the studies started with a relatively facile task, namely ego-motion which refers to estimating the camera's motion relative to a fixed background. For example, Pflaum et al.[60] implemented a drone stabilisation algorithm by processing the data from an embedded dynamic vision sensor (DVS), a sub-type of event-based camera. Later, researchers simulated flight control using either DVS[61] or using a process similar to a bat's echolocation[62]. Moreover, neuromorphic computing has been used for attitude estimation by processing data from traditional sensors[63].

In any case, most of the research performed at the intersection of neuromorphic computing and drone domains focused on the control tasks. In traditional control, a popular solution to fly the drone is the use of PID controllers. However, due to the integral and derivative parts of this technology, accurately implementing PID controllers on neuromorphic hardware is not trivial. Some studies focused on MAV applications managed to implement such a controller[64, 65]. Moreover, researchers have applied PID controllers for the control of one degree of freedom[66, 67]. Another prevalent task for drone flight is navigation. To solve this problem, Jiang et al.[68] used visual means and reinforcement learning algorithms to control the drone while Landgraf et al.[69] used optic flow to learn 6-dimensional ego-motion. As a novel domain still in great development, the algorithms of previous work have few functions in common, making them hard to be generalised to other applications. For this reason, researchers created some frameworks which were generalised for drone control[70, 71]. Last but not least, using inspiration from nature, researchers managed to solve various problems encountered by drones. For example, using an event-based camera, Paredes-Vallés et al.[72] could estimate the optic flow. The resulting algorithm could be used for landings[73], one approach even using a divergence-based algorithm for high-speed drones[74].

Research was also directed towards making a fully neuromorphic drone which flies using an event-based camera and whose information is processed and output by a neuromorphic chip. Computing ego-motion with an event-based camera, a few researches were centred on making a fully neuromorphic drone capable of hovering, landing or following a course of set points[75, 76, 77]. The results look very promising with low power consumption and latency while performing online learning and even surpassing traditional architecture performance in some cases. Another recurrent task for drones is obstacle avoidance and research also suggests that a fully neuromorphic drone achieves better energy consumption (up to 6 times less power) but also better performance compared to a conventional approach[78, 79, 80, 81]. To facilitate future implementations, Bian et al.[82] created a framework that can be used for fully neuromorphic drones to solve various tasks.

3 Encoding

Due to the inherent need for spikes to process data within neuromorphic computing, encoding is usually required. Thus, it is important to examine the methods by which numbers can be encoded (and later decoded if necessary) in spikes. It is not always the case that the SNN requires encoding as data expressed as spikes start to be more common with the introduction of neuromorphic sensors and event-based cameras. However, most neuromorphic drone applications still rely heavily on encoding as implementation on neuromorphic hardware represents a difficult procedure hardly considered during feasibility analysis. For this reason, common encoding methods will be studied in the current chapter. The analysis will start with clearly defined methods using the rate (section 3.1), time (section 3.2) and population (section 3.3) of spikes. Then, a non-deterministic method of learning a specific encoding technique using the principles of autoencoders will be described in section 3.4.

Before diving into the details of these conventional encoding methods, it is important to state that more encoding possibilities are available or can be created, but due to their scarce applicability, it was decided to discard them. To briefly mention a few, Schuman et al.[83] performed a thorough analysis of novel encoding methods. Similar to position coding summarised in section 3.3, inputs can be encoded by sending several spikes at a fixed rate. Thus the range of the input is again encoded in bins but now each bin is represented by how many spikes are generated at a fixed rate. Another method is more probabilistic and relies on the charge value that enters the neuron which may or may not make the neuron spike. Moreover, combining various encoding methods is also possible and has been largely done in literature. However, for the sake of simplicity, the resulting merged algorithms will not be detailed but only the traditional basic methods will be presented below.

3.1 Rate Coding

Rate coding is used to encode the information into the spiking rate. In short, the value of the stimulus entering the neuron is proportional (direct or inverse) to the firing rate. However, this encoding method ignores any information present in the exact timing of the spike as well as specific spike sequences and only works by recording the number of pulses in a time interval. As a result, it can be considered a quantitative measure of the neuron's output[84]. Moreover, because the sequence of spikes generated by a given stimulus varies from trial to trial even though the firing rate is the same, the output of a neuron is usually treated as probabilistic[85]. This coding scheme is inspired by biology and it was initially demonstrated by Adrian and Zotterman[86] which found these types of neurons in the muscle. They found that a high stimulus (e.g: the muscle has to carry a heavier weight) led to a higher frequency of spikes and this effect can be visualised in Figure 3.1[1]. This rate encoding research set the standard in describing the

[1]https://backyardbrains.com/experiments/ratecoding

properties of neurons by measuring the firing rates which inspired further the encoding methods in neuromorphic computing.

Being the first encoding model discovered in the brain, it was also the easiest to understand and apply. However, many drawbacks were discovered over time. For example, encoding data only using rate, has poor performance compared to temporal encoding as the information encompassed in the exact timing of the spikes or in the interval between spikes is lost[87]. This means further that stimuli that vary fast are not accurately represented by rate encoding[44]. Despite all these disadvantages, researchers found some strong points of this method including here the ease of implementation. Moreover, rate encoding is highly robust to Inter-Spike Interval Noise as compared to temporal encoding[85] and it can be applied easily to ANN-to-SNN conversions[84]. With these pluses and minuses in mind, researchers have theorised multiple encoding methods to be applied to neuromorphic computing. One such method takes the proportionality mentioned above and transforms the linear relation between the intensity of the stimulus and firing rate to other functions such as the Poisson or Bernoulli distributions[88]. More such methods are presented by Dupeyroux et al.[87]:

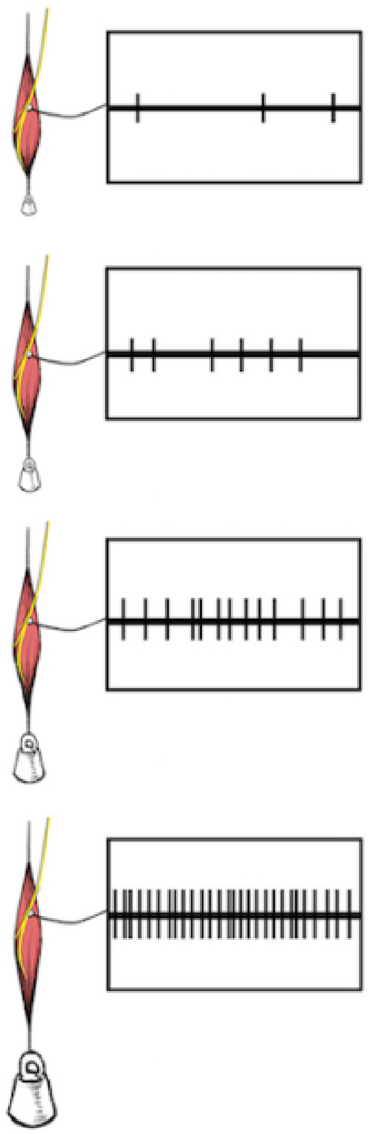


Figure 3.1: The rate encoding experiment

- **Hough Spike Algorithm**[89] starts from the premise that the output will be decoded with a convolution which means that the encoding replicates a ”reverse convolution” to invert the effect; the spike timings are determined with a finite impulse response method;
- **Threshold Hough Spike Algorithm**[90] is similar to the Hough Spike Algorithm but it introduces a threshold to be compared with the error between input and output; when the threshold is exceeded, a spike is emitted and the input signal is updated by subtracting the threshold;
- **Ben’s Spike Algorithm**[90] is similar to the previous methods but it uses 2 errors to generate a spike now: the sum of differences between the signal and the filter and the sum of the signal values; then a spike is generated by comparing the first error to a fraction of the accumulated signal, defined as the product between the second error and a predefined threshold

The drawbacks of rate encoding did not discourage the researchers from using this method and it found application easily, especially in innovative solutions to neuromorphic computing problems due to its simplicity. For example, Stroobants et al.[65] used this encoding approach to implement a better-performing PID controller that adapts thresholds based on the input weights. The method was implemented to control the rate of the very small flyer Crazyflie. Next, Zaidel et al.[91] implemented rate encoding for manipulating a 6 DOF robotic arm. To solve this task, a PID controller along with inverse kinematics to compute an approximation of the robot’s state was used. In these examples, only basic raw encoding was implemented. However, Clawson et al.[57] applied a more complex rate encoding for performing hover, trajectory-following, and perching on a very small flyer. To get faster information from the firing rate, multiple neurons encode the same input which allows the averaging window to be much smaller and the decoded output can therefore respond much more quickly to changes.

3.2 Temporal Coding

Compared to rate encoding, the temporal encoding scheme is based on the precise timing of spikes where the dynamics of the stimulus and the nature of the neural encoding process influence the temporal structure of a spike train[85]. For example, information that is more important can be encoded as earlier spike times[18]. The inspiration for this encoding approach stems from biology where it was found that neurons exhibit high-frequency fluctuations of firing rates. While rate coding implies that these irregularities are noise, temporal coding suggests that these encode information. Apart from the evolutionary disadvantages of the irregularities, responses between similar stimuli are different enough to suggest that the spike train contains a higher volume of information than can be encoded in rate. For example, information can be acquired from the time-to-first-spike, phase-of-firing, spike randomness, temporal spike patterns, etc.[92]. A brief visualisation of temporal encoding can be seen in Figure 3.2 where the time-to-first-spike is exemplified. These allegations were supported by neuro-physiological studies showing that, due to fine timing of the tasks, auditory and visual information is processed with high precision in the brain such as sound localisation or defined edges[87]. Moreover, this encoding works with no absolute time reference, which means that the information is carried either in terms of the relative timing of spikes or with respect to an ongoing oscillation.[93]

Analysing the theoretical background of temporal encoding, some clear advantages can be noticed with respect to rate encoding. For example, much sparser spikes are produced which allow for expressing the input with few neurons that reduces the energy consumption even more in the end[18]. But probably the most important advantage is that temporal codes employ many more features of the spiking activity that cannot be described by the firing rate which allows more information capacity to be carried in a shorter period and with fewer resources[87, 85]. However, there are some disadvantages too. Due to the quantity of information that has to be stored in time, several studies have found that the temporal resolution of the neural code is on a millisecond time scale, proving that precise spike timing is required. This makes temporal encoding vulnerable to input noise and temporal jitter[18]. Moreover, using this coding approach can lead to extracting a sheer volume of unimportant information as every timing feature now leads to data[44]. Last but not least, temporal encoding has not been applied often until now due to the difficulty of identifying a temporal code in the interplay between stimulus and encoding dynamics[85]. With the advantages and disadvantages in mind, some encoding approaches were still implemented for neuromorphic computing as summarised by Dupeyroux et al.[87]:

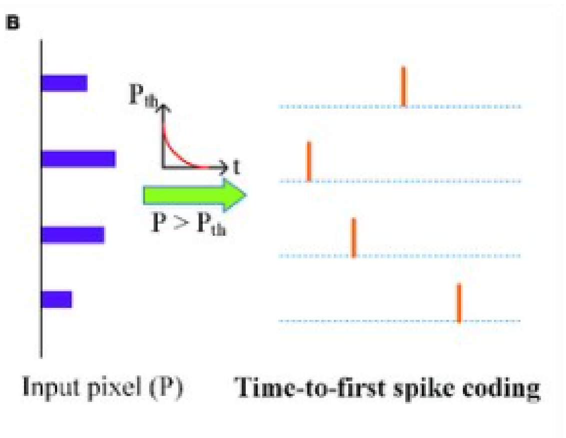


Figure 3.2: A schematic of how information can be encoded in time using time-to-first-spike[94]

- **Temporal Based Representation**[95] generates a spike whenever the signal difference between two consecutive timestamps, gets higher than a fixed threshold; this is the same principle used by event-based cameras which generate a spike when the pixel brightness changes;

- **Step Forward**[96] which is based on the previous algorithm but uses a baseline signal (usually the first input) to compute the signal differences; when the variation exceeds the threshold, the baseline gets updated, taking into account the threshold;
- **Moving Window**[96] which is based on the Step Forward algorithm but the baseline signal is calculated as the mean of the previous signal intensities over a time window;

The drawbacks of temporal encoding mentioned above made it difficult to be applied consistently. For this reason, this approach was not implemented in drone applications but some researchers still found utilisation in other realms. For example, as mentioned above, the Temporal Based Representation algorithm was specifically made for interpreting the event-based camera dataset. Other applications include learning and acquiring information from datasets such as image classification[97], pattern recognition and event prediction[96]. Temporal coding has also stayed at the basis of SpikeProp, a solution that promises to solve the back-propagation problems often recurrent in neuromorphic computing[98]. To conclude, temporal coding has been mostly applied to innovative and low-level solutions that require extremely fast encoding.

3.3 Population Coding

Different from the previous methods which encoded information within time, population coding is an approach that represents stimuli by using the joint activities of several neurons. In more detail, each neuron acquires a distribution of responses over the range of inputs. When a stimulus arrives, the activity of the neuron is greater if the stimulus value is close to the neuron's distribution and vice-versa. Then, for reconstruction, the spiking activity of the neurons can be combined which will result in the decoded response. The overall pattern of activity in the set of neurons is taken and then combined with various methods such as simple averaging or, for a more accurate response, maximum likelihood[99]. This approach was also theorised and it remains one of the few mathematically well-formulated problems in neuroscience. This makes this method simple enough for theoretical analysis but it also maintains the essential features of neural coding[85]. This method was found to be widely used in the brain for sensing and locomotion such as ganglion cells of the retina or the somatosensory cortex[87]. To exemplify the whole population coding process, an experiment was done that relates the process of visually following a source of light. Shortly, if each neuron represents movement in its preferred direction after the vector sum of all neurons is calculated, the sum will point in the direction of motion[100]. This is exemplified visually in Figure 3.3[2].

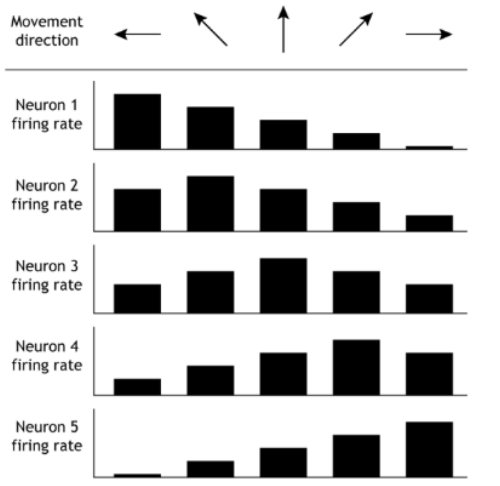


Figure 3.3: A schematic of how information can be encoded using a population of neurons

As a population of neurons is being used compared to a single one, this encoding method has several advantages. For example, in the neuron system, it was found that individual neurons are susceptible to interference, while population encoding helps to solve this problem and can increase accuracy[44]. Moreover, due to the higher number of parallel neurons, it can represent many different stimulus attributes simultaneously. However, it was found that individual neurons typically select to which stimuli to respond. This means that, based on the difficulty and accuracy of the task to be performed, not all the neurons in a population respond to the stimulus

[2]https://openbooks.lib.msu.edu/neuroscience/chapter/execution-of-movement/

to save more power. Compared to rate encoding, population encoding is also much faster and it can process stimuli variability nearly instantaneously[100]. Even though population encoding has several advantages, there is a drawback that seriously reduces its application. To have accurate and fast results, it is necessary to assign numerous neurons to one stimulus that usually penalises the size and power required. However, the advantages and the clear mathematical formulation led to various implementations, as theorised by Dupeyroux et al.[87]:

- **Position coding** is based on binning (dividing the range of possible input in multiple bins) assigning a spiking neuron for each bin which fires when the input is in the range of the respective bin; the division can be done linearly with all neurons representing a similar portion of the input space but also non-linearly allowing more precise representation at certain parts of the input space where accuracy is important;
- **Gaussian Receptive Fields** encodes the signal using a set of neurons whose activity distributions are defined as Gaussian waves determined by a centre(μ) and a variance(σ^2); all the centres of neurons are regularly spaced to cover the maximum amplitude of the input signal, while the variances are set equal; this method also looks at the timing of the spikes as the neurons with a high activation will fire at the beginning, and neurons with a lower activation later;

These 2 implementations have been applied to various tasks in numerous domains which makes population coding an easy approach to solve a wide variety of problems. One perfect application for binning remains discrete input as possible values can be accurately split for each neuron. This was implemented in the work of Patton et al.[101] for autonomous racing where the steering commands were discretised. Linear position encoding has been applied also in the realm of drones where neuromorphic PID controllers were theorised and implemented to navigate to various locations[64, 66]. The non-linear approach has been used for a drone control task as well. In their paper, Dupeyroux et al.[73] manage to control the thrust based on the divergence of the optic flow field for landing. However, for an accurate representation of the input space, a lot of neurons are required which sometimes is not possible. For this reason, Gaussian Receptive Fields have been used for image recognition tasks[85].

3.4 Autoencoder

Autoencoders use a different procedure to transform values to spikes compared to the previous encoding methods which have a biological inspiration. While the other approaches focus on creating clear rules for the transformation, autoencoders try to learn these rules with the help of neural networks. For this reason, they are usually coupled with other encoding methods but autoencoders formulate the rules. The inspiration for this encoding method came from the realm of traditional ANNs. Autoencoders can be considered as a type of representation learning that was first introduced with the scope of performing dimensionality reduction[102]. They aim to learn a data-driven representation of the input information in abstracted and compressed form without performing any additional "cognitive" task[103]. Moreover, its applications focused not only on the removal of noise from corrupted data but also on generating data with the help of variational autoencoders[102]. Recently, these applications have been largely applied to neuromorphic computing too for learning representations of values into spikes, as will follow from the following paragraph.

Most of the autoencoder applications used in neuromorphic computing were focused on reproducing the capabilities of traditional ANNs for image recognition. However, using this application, some researchers also looked at improving existing SNN algorithms. For example, Fang et al.[97] achieved better results when the time constant was also introduced in the learning process. Other papers focused on reducing the dimensionality of images with spiking autoencoder[104, 102, 103]. Using only one hidden layer between the input and output layers, Comsa et al.[102] concluded that keeping accuracy is still possible while having only 10% of the initial input neurons in the hidden layer. Autoencoders have been used for generating new datasets too, with the work of Kamata et al.[105] implementing a variational autoencoder with SNN that managed to exceed the performance of traditional ANNs. The visualisation of autoencoders can be seen at Figure 3.4. The applications of autoencoders do not resume only to image recognition, but also to encoding data for drones. Within a fully neuromorphic drone that uses event-based cameras as input, Paredes-Vallés et al.[75] have trained an autoencoder for decoding the spikes to motor commands. Similarly, Stroobants et al.[63] estimate the drone's attitude neuromorphically by encoding the input formed of gyro and acceleration values and then decoding the received roll and pitch values from spikes.

As can be seen, autoencoders are an increasingly popular method for encoding data into spikes, especially for image recognition but for drone applications too. One of the main advantages is their efficient approach. Managing to save only the most important features of the whole input space, autoencoders reduce noise and, implicitly, energy consumption. Moreover, the inspiration for this approach lies in the traditional ANNs which were analysed in detail making the spiking autoencoders become just a simple translation which eases the understanding and application of this approach. On the negative side, this method has some drawbacks too. Firstly, the translation from ANN to SNN is not as straightforward. Making an autoencoder with SNN requires the creation of a latent space which is often represented as a normal distribution in ANNs. However, sampling from a normal distribution is not possible within SNNs because all features must be binary time series data. This problem was solved with an autoregressive SNN model which randomly selected samples from its output to sample the latent space variables[105]. Another problem is that, as opposed to the previous methods based on clear rules, the neural network needs training. Thus, the creation of an unbiased dataset is required. Along with the training process, temporal investment is needed to achieve good performance.

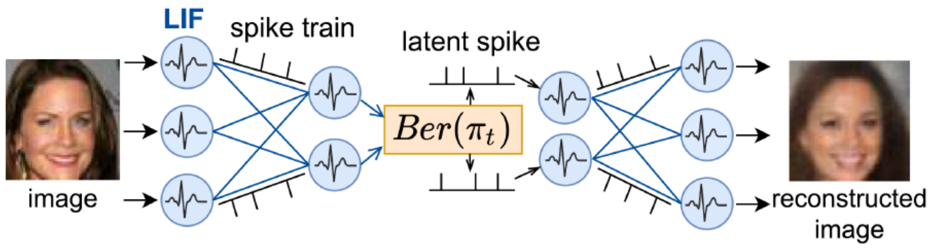


Figure 3.4: A visual explanation of how autoencoders work. For encoding data into spikes, only the first half is required while the second half is required for decoding[105]

4 Neuron Models

With the data encoded, it is important to start analysing the neuromorphic computing architectures. Similar to the brain, but also to the traditional ANNs, the powerhouse of the SNNs is represented by the neuron. Starting from this requirement, literature has proposed models which vary from mimicking the brain as closely as possible to focusing on being computationally efficient. In SNN literature, the neuron models are composed of 2 parts. Firstly, the actual processing dynamics are calculated inside the neuron module which will be presented in section 4.1 along with its different models. Secondly, the links between the neuron modules, namely the synapses, also require special dynamics which will be detailed in section 4.2.

4.1 Neuron Module

Within the field of neuromorphic computing, the analysis of neuron modules has been extensively documented in the literature leading to various possibilities depending on what the researcher seeks. For example, if accurate biological representation is required, several models were summarised in subsection 4.1.1. On the other hand, if the researcher looks for better computing performance, the Leaky Integrate-and-Fire (LIF) model is a better compromise, presented in subsection 4.1.2. Due to its simplicity but still being able to achieve accurate results, this neuron model was applied the most in the literature. However, different variations still arose which will be better detailed in subsection 4.1.3. A graphical comparison of these models focusing on biological plausibility and implementation cost is presented in Figure 4.1.

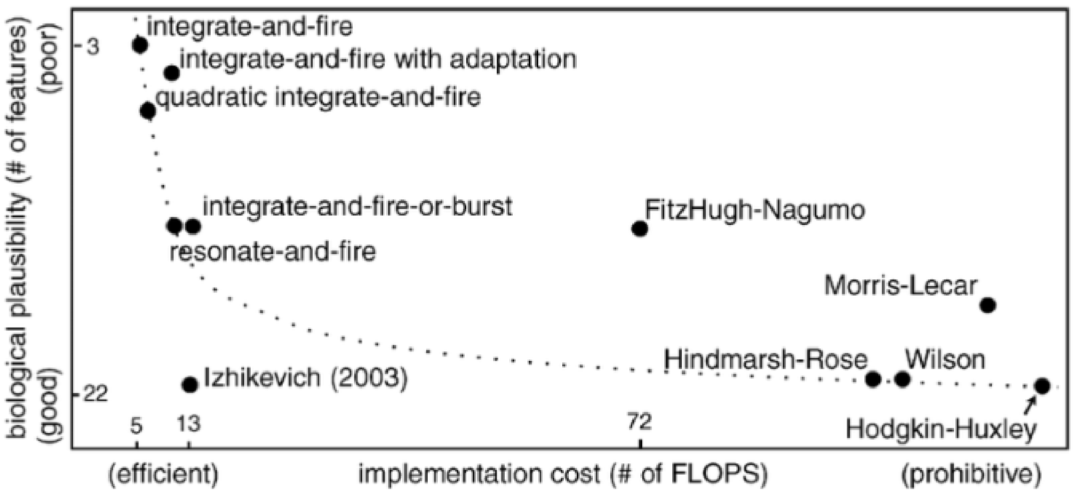


Figure 4.1: A comparison of spiking neuron models analysed in future sections by looking at both implementation cost and biological plausibility[18]

Before entering the analysis, it is important to mention some factors that are common to all models irrespective of biological accuracy or computational efficiency. Firstly, the neuron models are expressed using mathematical equations that describe the behaviour of spiking neurons. These equations consider various factors such as the input current, membrane potential, and membrane time constant, to simulate the behaviour of biological neurons. Each of the models mentioned below is implemented using the update method, which takes as inputs a current and a time step and returns whether a spike has occurred or not. With the help of the equations above, the update method calculates each neuron's membrane potential at each time step. If the membrane potential exceeds a certain threshold, a spike is generated and propagated to the next neurons through the synapse[106].

4.1.1 Biological Inspired Models

In this subsection, the neuron dynamics models, centred around biological accuracy will be detailed. To understand the process these models try to reproduce, it is important first to understand the biology behind the neuron, presented in section 2.1. With this in mind, the most accurate biological neuron model is the one created by Hodgkin and Huxley[107](for a summary of the equations check the work of Yamazaki et al.[18]). This neuron model was first introduced in 1952 and it is relatively complex to be implemented and computed. It is formed of four-dimensional nonlinear differential equations which describe the neuron's behaviour using the chemistry behind the transfer of ions into and out of the neuron. Because of their biological plausibility, Hodgkin-Huxley models have been heavily implemented in neuromorphic applications trying to accurately model biological neural systems. Another plausible biological model is the Morris-Lecar model[108] which focuses on slightly reducing the complexity due to its two-dimensional nonlinear equation. For this reason, it is also commonly implemented in neuroscience and neuromorphic systems[7].

However, for the computational side of neuromorphic computing, the complexity of the presented models is not favourable. Thus, some researchers created simplified models requiring fewer parameters that can be implemented on neuromorphic hardware. Trying to replicate the behaviour of the Hodgkin-Huxley neuron, the Fitzhugh-Nagumo[109] and Hindmarsh-Rose[110] models were created. Similarly, the Izhikevich spiking neuron model[111] was developed to produce similar behaviour as the Hodgkin-Huxley model when it comes to the most important neuron attributes (bursting and spiking behaviour) but using much simpler computations. Achieving great biological plausibility with a low computational cost, the Izhikevich model became increasingly popular in the neuromorphic literature. Following this success, other models arose which imitated the approach of Izhikevich such as the Mihalas-Niebur neuron model[112]. It is also important to note that other researchers looked at replicating the biological behaviour of neurons with lower implementation costs by focusing on accurately implementing parts of the neuron. Thus they contain a much higher level of biological detail than the models mentioned above but focus on hardware implementation of smaller components such as membrane dynamics, ion-channel dynamics or axons and dendrites[7].

4.1.2 Leaky Integrate-and-Fire (LIF)

Focusing especially on computational efficiency, researchers created a much simpler set of spiking neuron models, namely the integrate-and-fire family. These models are less biologically realistic but still vary in complexity from the basic integrate-and-fire model to those approaching complexity levels near that of the Izhikevich model. To start with, the basic idea used by the neurons in this family is, as the name suggests, to integrate the spikes received in the neuron until a certain threshold is achieved which makes the neuron fire a spike itself. This principle is summarised by the integrate-and-fire neuron model which maintains the current charge level of the neuron after a spike is received[7].

To achieve better biological accuracy by reflecting the diffusion of ions that occurs through the membrane when some equilibrium is not reached in the neuron, the leaky integrate-and-fire (LIF) model was created which includes a leak term to the model causing the neuron's potential to decay over time[18]. Moreover, a refractory time is often included after each spike during which the neuron will not integrate any presynaptic spikes[57]. This increase in accuracy makes it one of the most popular models used in neuromorphic systems but also one of the most researched models with many variations being brought in numerous papers. The dynamics are summarised in Equation 4.1[97]:

$$\tau \tfrac{dv_i(t)}{dt} = -(v_i(t) - v_{rest}) + X_i(t) \tag{4.1}$$

In Equation 4.1, τ is the time constant of the membrane potential which controls how fast the neuron potential leaks, $v_i(t)$ represents the membrane potential of the neuron i at time t, v_{rest} is the resting potential and $X_i(t)$ represents the input function to the neuron i at time t which can add various properties to the LIF neuron. The presented equation describes only the exponential leak equation which functions between spikes. Otherwise, the membrane potential $v_i(t)$ integrates the spikes received from the presynaptic spike train $s_j(t)$. When the membrane potential $v_i(t)$ exceeds a certain threshold v_{th} at time t_f, the neuron will cause a spike and then the membrane potential $v_i(t)$ goes back to a reset value v_{reset} which should be strictly smaller than v_{th}. Usually, the reset value v_{reset} and resting potential v_{rest} are equal in the neuron dynamics but they can differ too. After the spike is generated and the membrane potential is reset, the neuron enters a refractory period Δt_{refr} during which new incoming spikes do not affect the membrane potential $v_i(t)$. The whole dynamics of the neuron leads into a postsynaptic spike train $s_i(t)$. To conclude, the whole process can be visualised in Figure 4.2[72, 97].

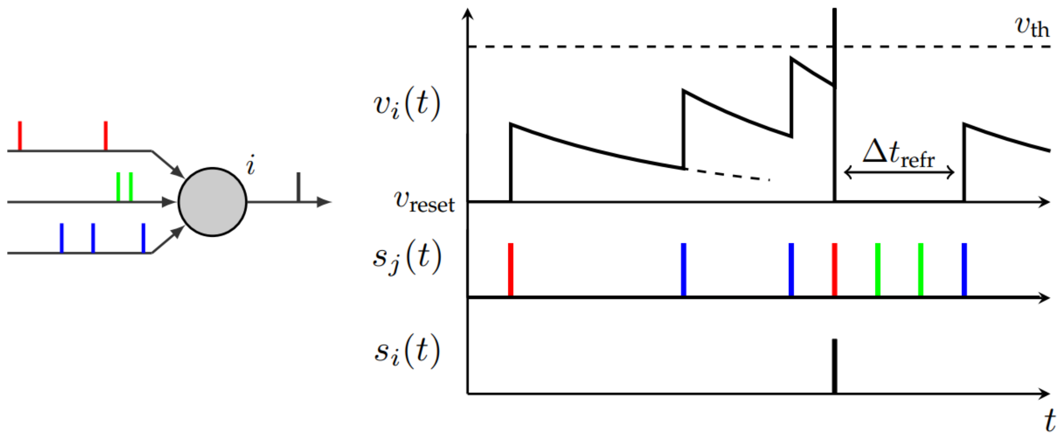


Figure 4.2: A generic model of a LIF neuron; the graph on the right shows the temporal course of the membrane potential $v_i(t)$ driven by a sample presynaptic spike train $s_j(t)$ coming from 3 input synapses which leads to the postsynaptic spike train $s_i(t)$; also, in this graph, the reset value v_{reset} and resting potential v_{rest} are equal in magnitude [72]

The LIF neuron description given above is mostly focused on the physical side. However, to implement it on neuromorphic hardware some changes are required. A trivial implementation using the dynamics of a simple RC circuit was done by Clawson et al.[57]. However, for a more complicated implementation taking into account more parameters and factors in the hardware, the method presented by Burkitt[113] could be used. All this analysis was done if specialised neuromorphic hardware is available but applications can still be implemented on a von Neumann architecture. The problem translates then to the necessity of derivative discretization which requires a discrete-time system of equations as suggested by Stroobants et al.[63]. Also, it is important to state that different variations of the LIF neuron model can be implemented using the input function $X_i(t)$ from Equation 4.1. For example, Clawson et al.[57] use this function to add a disturbance scalar meant to represent noise in the neural circuit. Another example was given by Paredes-Vallés et al.[72] who created a LIF neuron model that adapts neural response to the varying input statistics using the presynaptic trace as an excitability indicator. Using this method, the LIF neuron model can be used also for rapidly varying input statistics such as event-based cameras.

To conclude the analysis of the LIF neuron model, the advantages and disadvantages of using such an approach will be presented. First of all, it seems the main advantage of LIF neuron models is their simplicity allowing for a low computational cost while keeping the biological plausibility of the spiking behaviour. In this way, the method provides both analytical methods of solution and intuitive insights into many important questions regarding neuromorphic computing[113]. Moreover, its simplicity also allows for computational traceability but also for the rapid conversion of an ANN to an SNN. This can be done by transforming the firing rate of the neuron in SNNs in such a manner that it behaves similarly to the ReLU activation function in ANNs[18]. However, some drawbacks exist too and mostly concern the biological plausibility of the model. In other words, the LIF neuron model cannot stimulate accurate neuronal behaviours except for leakage, accumulation, and threshold excitation[44]. Moreover, the model neglects the spiking mechanisms and the lack of spatial structure which leads to issues with the irregular nature of interspike interval but also of neural gain modulation[113].

4.1.3 Other variations of Integrate-and-Fire

The integrate-and-fire models became popular in neuromorphic research due to their low computational complexity, allowing for numerous papers on this method. Some of these papers also focused on fixing some drawbacks by slightly altering the model as it was mentioned in subsection 4.1.2. However, its main drawback remained the low biological plausibility compared to other neuron models as can be seen in Figure 4.1. To fix this issue, researchers created more complex variations of the integrate-and-fire models with a higher biological plausibility while keeping the computational cost at an acceptable level. This subsection will analyse some of them while trying to present their assets and drawbacks. Before jumping into the analysis, it is important to state these methods follow the basic principles of the neuron model presented in the previous subsection if not indicated otherwise. There exist many variations of the basic integrate-and-fire models but the most important ones that were used frequently in literature can be found in the list below.

- **Non-Linear Integrate-and-Fire (NLIF)**[114]: compared to the LIF neuron model, NLIF is made more complex by adding two additional parameters for the refractory period; thus, the non-linear relationship between the membrane potential and the input current in the integration process is included;
- **Adaptive Exponential (AdEx)**[115]: this neuron model includes an exponential term to account for the adaptation of neuron firing rates over time; this model has parameters for the membrane time constant, rheobase voltage, spike voltage, adaptation parameter, reset voltage, initial voltage, and the number of neurons which makes it more biologically accurate;
- **Quadratic Integrate-and-Fire (QIF)**[116]: this model is an application of the NLIF model as it incorporates a quadratic term to capture the non-linearity of the neuron's behaviour near the spike threshold; it is thus a simple yet effective way to introduce non-linearity, which allows it to capture certain behaviours of spiking neurons more accurately;
- **Integrate-and-Fire with Spiking Frequency Adaptation (IF-SFA)**[117]: IF-SFA is based on the LIF model with an additional adaptation current, which modifies the mem-

brane potential and firing rate of the neuron over time in response to input stimuli; the adaptation current is computed based on the difference between the membrane potential and the resting potential and is added to the input current; thus an increase in the adaptation current leads to a decrease in the firing rate of the neuron over time, allowing the neuron to adapt to the input stimulus;

- **Spike Response Model (SRM)**[118]: SRM is less inspired by the basic integrate-and-fire model as it tries to provide a more detailed description of neuron behaviour; it does this by capturing the post-spike response characteristics of neurons, which include the refractory period and the shape of the post-spike potential; this model manages to accurately model the temporal effects in neural computations;
- **ThetaNeuron**[119]: this neuron model is even closer to biological plausibility as it focuses on capturing the theta rhythm observed in the brain; it incorporates the influence of a sinusoidal waveform, in addition to the input current and membrane potential to update the membrane potential;

Sanaullah et al.[106] described the mentioned models while also performing a comparison analysis (including also the basic LIF neuron model). To compare the performance of these models as accurately as possible, Sanaullah et al.[106] used various methods. The first one calculates classification performance per neuron model providing insights into the diverse spiking behaviours and relative strengths of the different SNN models. The metrics used for comparison are: accuracy which is calculated as the mean of the element-wise equality comparison shown as a percentage and error rate which is simply the percentage of misclassified samples. The results are summarised in Table 4.1 and show some interesting conclusions. For example, the neuron models that are less similar to integrate-and-fire models and try to simulate closer the biological neuron (SRM and ThetaNeuron) show the worst learning capabilities for classification tasks. On the contrary, the integrate-and-fire models that include some adaptation components (IF-SFA and AdEx) for learning perform the best. The models closest to the basic integrate-and-fire model (LIF, NLIF and QIF) perform similarly. An interesting outcome is that making the integrate-and-fire models not depend on time (QIF and NLIF) decreases the performance which could be explained by the missing temporal component of the learning process.

Table 4.1: Comparison of various integrate-and-fire models considering the accuracy and the error rate as calculated by Sanaullah et al.[106]

Parameter	**LIF**	**NLIF**	**AdEx**	**QIF**	**IF-SFA**	**SRM**	**ThetaNeuron**
Accuracy [%]	71.20	66.55	90.05	70.70	84.30	49.95	58.55
Error rate [%]	0.32	0.35	0.10	0.27	0.14	0.50	0.42

Next, Sanaullah et al.[106] looked at comparing the network integration of the aforementioned models. Thus a consistent network with a similar topology was created. The network included one layer of 1000 neurons using random initial weights. Even though this network structure might favour some neuron models, it was decided to still perform the comparison as the results will evaluate the inherent characteristics of each model in a controlled setting. Using this approach, a deeper understanding of the fundamental properties of each model, such as their spike response dynamics and computational capabilities can be gained.

One such property is the spiking activity of the neuron models which provides information on the mechanisms underlying neural computation and communication. For instance, it can be investigated how different input patterns affect the firing rate and temporal precision of spike trains which can further lead to better simulations of biological neurons behaviour. Additionally, spiking activity can provide insights into the computational efficiency and resource requirements of different models. With these in mind, the results of Sanaullah et al.[106] lead to some interesting conclusions. Firstly, the neuron models which are more similar to the integrate-and-fire model, show similar behaviour, with spikes being generated when the threshold is reached. The main difference between them consists in the exact timing of the spike. For example, the LIF neuron spikes exactly when the threshold is reached. On the other hand, AdEx, NLIF, QIF and IF-SFA introduce non-linearity near the spike threshold which allows it to capture certain behaviours of spiking neurons more accurately. On the other hand, the other models (ThetaNeuron and SRM) spike continuously. Using this feature, the ThetaNeuron model can be adapted to capture various patterns by adjusting the model's parameters. Moreover, the leak conductance of SRM makes it a valuable model for computational neuroscience and neural network simulations that involve synaptic interactions.

Last but not least, computational complexity (which can be argued that is directly proportional to energy consumption) was analysed by Sanaullah et al.[106] by looking at the mathematical operations required by these models. These are summarised in Table 4.2 by showing the required operations for updating the neuron state in one time step. Before starting the analysis, it is important to state that addition/subtraction requires lower complexity than multiplication/division and the comparison operation requires the least complexity. Based on the results, it can be observed that the LIF neuron model is the least complex by a great margin. With similar complexity, the adaptive neuron models (AdEx and IF-SFA) follow the ranking along with SRM. The non-linear neuron models (QIF and NLIF) show a dramatic increase in complexity compared to LIF. ThetaNeuron is the most complex which is explained by the focus of the model on biological plausibility.

Table 4.2: Comparison of various integrate-and-fire models considering the computational complexity[106]

Operation	**LIF**	**NLIF**	**AdEx**	**QIF**	**IF-SFA**	**SRM**	**ThetaNeuron**
Addition/Subtraction	1	3	4	3	3	3	4
Multiplication/Division	1	3	1	3	2	2	3
Comparison	1	1	1	1	1	0	0

4.2 Synapse Module

The synapse is the linking part between neurons, being also the most numerous component in spiking neural networks. For this reason, nowadays, many hardware implementations and novel materials for the neuromorphic industry focus on optimising the synapse implementation. Regarding computation, unless they are attempting to explicitly model biological behaviour, synapse models tend to be relatively simple without meeting as much variation as the neuron module[7]. However, it is still important to present the existing models and how they differ. Again, while some models focused on the accurate biological representation, others concentrated on making the computation more efficient as will result from the following paragraphs.

For more biologically accurate neuromorphic networks, synapse implementations that model explicitly the chemical interactions of synapses, such as the ion pumps, ion channels or neurotransmitter interactions, have been utilised in some neuromorphic systems such as the Hodgkin-Huxley model. Another popular inclusion for more complex synapse models is a plasticity mechanism, which causes the neuron's strength or weight value to change over time as was observed in biological brains. Neuromorphic synapses that exhibit plasticity are common for novel biologically inspired learning mechanisms that use excitation and inhibition in synapses as will be detailed in section 5.2. Moreover, synapses have also been used as homeostasis mechanisms to stabilise the network's activity, a common issue in SNN systems.[7]

The methods presented above are mostly based on conductance which is easier to implement on neuromorphic hardware and more biologically accurate. However, this is more difficult to implement on traditional hardware which led to the need for a current-based (CuBa) synapse. The methodology behind it was inspired by the LIF model which uses a decaying exponential model for the post-synaptic potential (PSP). The novelty of the CuBa synapse includes the modelling of the post-synaptic current (PSC) using a similar decaying exponential model[18, 120]. Another method suggested by Zheng et al.[40] is called synapse integration which simulates plasticity using 2 neurons. One neuron spikes when the synapse should inhibit the weight while the other one is used similarly but for excitation. Last but not least, synapses could also be used as a linear connection between neurons and all the learning parameters depend on the neuron itself. In this case, the synapses will only contain the weight values which will be learned as it will follow from the following chapter.

5 Implementation

With neuromorphic technology and its possible applications inspected, it is crucial to understand how learning can be achieved with this novel technology. Until now, most of the information focused on the basics of neuromorphic computing where the neuron and synapse models were examined followed by possible encoding techniques to adapt this novel technology to existent hardware. This chapter will analyse how all this information can be linked and thus an SNN learn a model. It is important to note that most of the information presented below is heavily inspired by ANN applications but the focus of the sections will be on the SNNs adaptations. Firstly, possible architectures of neurons and synapses to generate better-performing networks will be analysed in section 5.1. With a network generated, learning can be performed and various general algorithms for this will be shown in section 5.2. Moreover, as the information is not continuous anymore but is discretised due to spikes, the traditional back-propagation method of learning cannot be used and the new neuromorphic methods of learning will be included in the same section. Lastly, after learning a model, quantifying the performance is crucial and possible metrics will be presented in section 5.3.

5.1 Network Architectures

The focus of the literature study in the previous sections was on analysing the network components such as the neuron or synapse while their possible combinations have not been analysed. The numerous possibilities of connecting the neurons and synapses lead to various capabilities of the resulting network. The current section will highlight how a network architecture may prove advantageous for one application while not necessarily suitable for another. The analysis will begin with the classical feed-forward neural networks as well as its multitude of variations presented in subsection 5.1.1. This will be followed by recurrent neural networks, a structure that allows also for passing the information in cycles in subsection 5.1.2. Lastly, a sub-layer of feed-forward, namely convolutional neural networks, which has been heavily used for image processing will be detailed in subsection 5.1.3. Most of the network architectures mentioned in the current section can be visualised in Figure 5.1[1].

5.1.1 Feed-Forward Neural Networks

The standard method of creating an ANN and by far the most prevalent neural network used to learn a model is the feed-forward structure. Within this architecture, neurons are organised into layers and they can communicate between them only sequentially from layer to layer, using synapses. Thus, neurons receive the input and pass the information further only to the next layer's neurons. The process is repeated and finishes when the output layer is reached. In this structure for artificial neural networks, the basic element is called the perceptron. This works by

[1]https://www.asimovinstitute.org/neural-network-zoo/

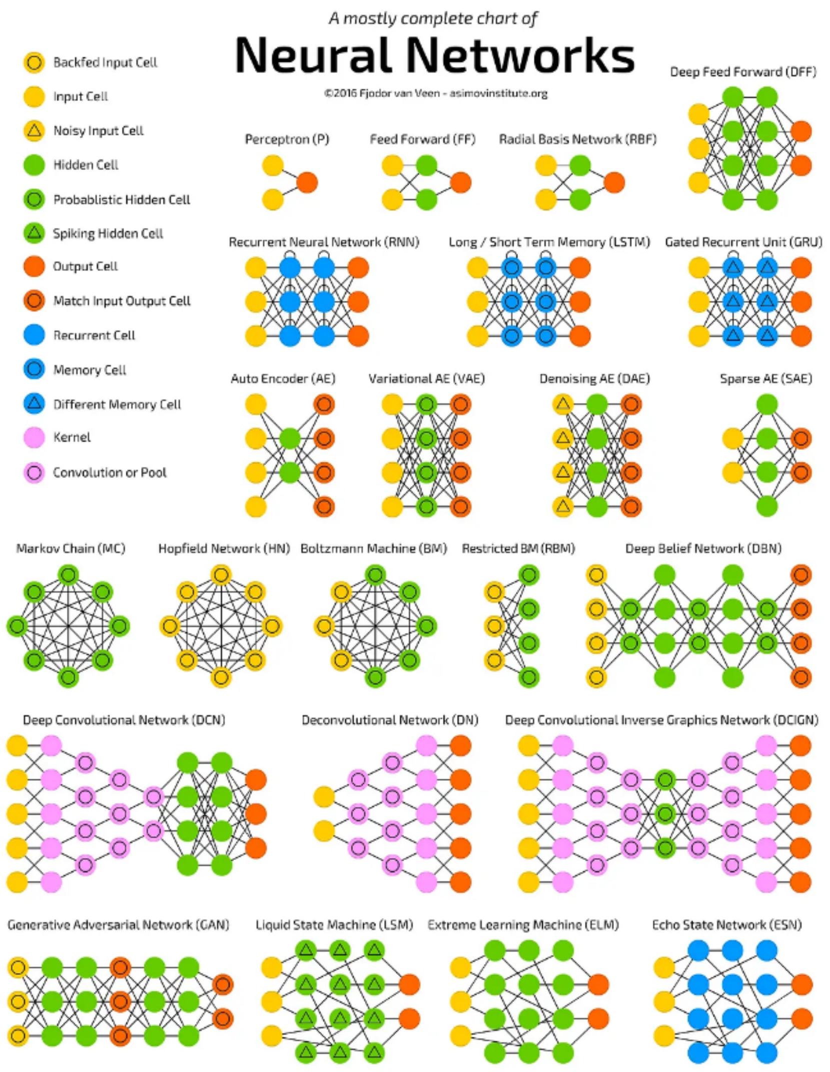


Figure 5.1: The schematic representation of various neural network architectures (both neuromorphic and non-neuromorphic)

first adding up the values received from the previous layer's neurons and then, after passing the result through an activation function, the perceptron sends its information to the neurons of the next layer. In this structure, learning is being done through back-propagation, a technique in which the weight of every synapse is adjusted to improve the performance of the whole network. The performance is usually quantified by comparing the output of the network model to the expected output. Due to its simplicity, this network architecture remains the standard of the feed-forward structure, being used for general learning problems but also for understanding the principles of ANNs.

To broaden the applications of feed-forward neural networks, several variations have been brought regarding its components. One of them analyses the number of synapses between layers with the standard model having fully connected layers (every neuron in a layer is directly connected through synapses to every neuron in the preceding and succeeding layers). However, researchers found that this usually leads to increased computational consumption especially during training as more weights have to be optimised. Thus they proposed more sparsely connected layers or fully connected only locally. Similarly, literature studied the possibility of saving the stochastic behaviour of neurons by introducing a new layer to record the probabilistic features of the network. This led to the creation of the probabilistic neural network which expresses weights and biases using a distribution function instead of a fixed value (Bayesian

Neural Network). Next, researchers examined single-layer feed-forward networks that are capable of learning complex, non-linear information using radial basis function instead of the simple neuron's summation. Last but not least, augmenting the complexity of the network by increasing the number of hidden layers, led to the notion of deep neural networks thereby enhancing performance, efficiency and accuracy.

The information presented in the paragraphs above applies to traditional ANNs but neuromorphic computing works with slightly different principles. A spiking neuron learns by correctly distinguishing the times of desired output spikes. Even though the information is processed differently, the network structure remains generally similar with neurons being split into layers connected by synapses and information being passed successively between layers. The difference consists mainly in the neuron and synapse models as presented in chapter 4 and in the back-propagation techniques detailed later in section 5.2. The network structures mentioned above were adapted to neuromorphic computing principles and technology. For example, the perceptron became the tempotron[121] in which the spiking neurons optimise the synaptic weights by minimising the error between the actual and desired output potential[41].

Based on the elementary feed-forward component of tempotron, more complex networks could be generated that focus on imitating the learning capabilities of traditional ANNs. Thus, single-layer feed-forward networks with spiking RBF were created[122] as well as spiking probabilistic neural networks[123] and their application was mostly focused on learning general problems such as image recognition with the MNIST or CIFAR-10 datasets or basic benchmark tests. However, the performance of these networks is generally worse than their traditional counterparts but these papers proved a breakthrough in neuromorphic research. To reduce the overhead, some papers looked to increase the complexity of the SNNs by adding more layers to achieve better performance, efficiency and accuracy leading to deep SNNs[124]. Neuromorphic research oriented its study also towards different approaches to learning information. This includes the work of Diehl et al.[125] who used 2 neurons (excitatory and inhibitory) for controlling the synapse weight applying this technique to extracting features from images[40].

Using the feed-forward principle but in a completely different manner, 2 more types of networks can be distinguished. The first one requires training more models to improve the accuracy of the output. It is called a generative adversarial network (GAN) which creates a new framework for estimating generative models using an adversarial process. Thus 2 models are simultaneously trained: firstly, a generative model (G) that captures the data distribution and generates data such that the other model is forced to make a mistake and secondly, a discriminative model (D) that estimates the probability that a sample came from the training data rather than from G. As long as G and D are built of multi-layer perceptrons, the entire system can be trained using the same feed-forward principles while achieving excellent results for both generative and recognition problems on MNIST dataset[126]. Using neuromorphic computing, this adversarial approach to learning was applied to the MNIST dataset but worse results were achieved compared to its conventional counterpart. For example, Rosenfeld et al.[127] created a GAN composed of a probabilistic SNN generator and an ANN discriminator while Kotariya and Ganguly[128] created a full neuromorphic GAN structure.

Another principle implemented using the feed-forward structure is the attention mechanism. This mimics cognitive attention by using 2 types of weights: soft which can be changed during each runtime and hard that are pre-trained and fine-tuned remaining frozen afterwards. This principle applied on a feed-forward neural network with fully connected layers for both the encoder and decoder led to transformers. These proved great capabilities for sequence modelling and transduction in various tasks. This allowed transformers to model the dependencies between bits of information without regard to their distance in the sequences leading to global dependencies between input and output. Moreover, at each step, the model is auto-regressive, consuming the previously generated symbols as additional input while also generating the next ones[129]. This network structure has also been successfully applied to neuromorphic computing for image classification[130], audio signal processing[131] and automatic speech recognition[132] with models outperforming standard SNNs but not the traditional ANNs. One possible solution to improve results might be to create special hardware such as a spike-driven transformer[133].

5.1.2 Recurrent Neural Networks

A second method to combine the neurons and synapses in a network does not require such an organised structure with information being passed successively from layer to layer. Instead recurrent neural networks (RNNs) allow for cycles that translate to information being passed backwards jumping to previous layers. There are still similarities to the feed-forward neural networks such as the usage of the perceptron neuron. But as opposed to feed-forward structures which memorise the overall sequence information, RNNs focus on memorising the relation between the information in the sequence. Thus they can be used for time series by combining the representational information of the previous time step in the hidden layer with the input of the current step to infer the output of the current time step. This makes RNNs a great application for processing sequential data or time-series data or for solving ordinal or temporal problems, such as language translation, and speech recognition[40]. To learn the correct synapse weights the same principles as feed-forward networks are used but slightly altered to adapt to the change in time leading to the back-propagation through time approach.

The distinctive feature of RNNs lies in their flexibility regarding structure, resulting in varying degrees of connectivity. For example, the feed-forward structure can still be preserved but recurrence between layers is also allowed now. An application of this principle is the one-layer recurrent neural network that proved itself efficient for sequence processing such as time series analysis. Extending this principle to feedback connections between each neuron leads to the long short-term memory (LSTM) network that improves memory capabilities. Another level of recurrence scraps the principle of layers and allows neurons to communicate between themselves directly and at any time. This leads to fully-connected recurrent networks where each neuron is connected to every other neuron. Although this architecture allows for very dynamic nonlinear behaviour, it is nevertheless complex and thus difficult to analyse and train. Therefore, simplification is required such as fewer connections between neurons or more structured relations. With this in mind, reservoir computing networks were created which allow for sparsely connected recurrent structures in the middle of the network. Their key advantage is that it does not require specific training rules as they use the sparse and recurrent connections with synaptic delays to cast the input to a spatially and temporally higher dimensional space. However, this requires a readout mechanism, such as a linear regression, that is trained to recognise the output of the reservoir. When the reservoir receives the network input and passes further the network output, the network becomes an echo state network (ESN)[7, 8].

RNNs have also been prevalent in the neuromorphic computing field as SNNs mostly process sequences of information due to the recurrent nature of the neuromorphic neurons. Moreover, the brain also uses a chaotic structure with neurons being connected randomly, allowing for complex nonlinear dynamics. Following the same principles as traditional RNNs, feedback connections to multi-layer feed-forward SNNs were implemented too. This led to slight variations as the feedback could be either external (passing the output back to hidden layers) or just internal (passing the information generated by hidden layers back into hidden layers themselves)[41]. Reservoir computing was also adapted to neuromorphic technology leading to liquid-state machines (LSMs). These work by transforming the time-varying input information into a higher dimensional space that can exhibit rich temporal and spatial properties as well as memorising past input information[40]. Several demonstrations of LSMs have shown their effectiveness at processing temporally varying signals for bio-signal processing and prosthetic control applications to video and audio signal processing applications[8].

A slightly distinct network that does not fit clearly into the field of RNNs even though it uses recursion is the central pattern generator (CPG) model. This model was inspired by nature and was found to underlie the rhythmic motor patterns in virtually every system of the human body. What makes these models special is that they create the rhythmic motor patterns locally with minimal sensory feedback leading to extremely fast reaction times without the need to pass the information to the brain and back. Based on this information, researchers have created CPG mathematical models, also implemented in neuromorphic computing. One such implementation was researched by Angelidis et al.[134] who designed a modular SNN architecture in which the oscillatory centre is represented by a population of spiking neurons as can be visualised in Figure 5.2. To solve the coupling between the neural oscillators, they introduce an intermediate population of neurons that receive the x and y values from neighbour oscillators and compute the coupling term. The final application of the paper was to control a simulated lamprey robot.

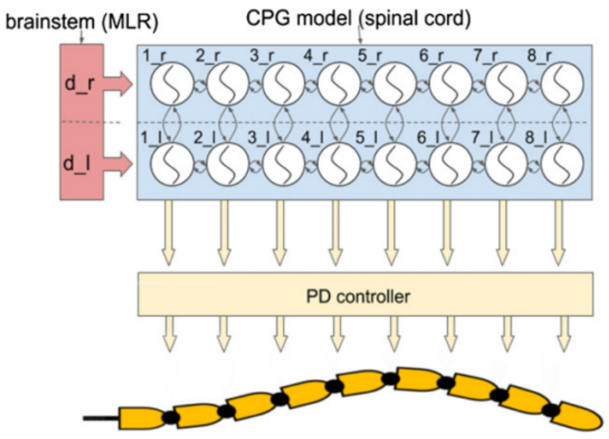


Figure 5.2: A schematic of how a central pattern generator works especially for rhythmic motor patterns[134]

5.1.3 Convolutional Neural Networks

Even though convolutional neural networks (CNNs) follow generally a feed-forward network structure, they deserve a separate section for the outstanding performance and usage they achieved in image analysis and language processing. To achieve this performance, they use layers that are specially created to detect patterns (edges, corners or textures) in the gridded data by grouping it in smaller batches. CNNs add to the feed-forward network structure convolutional and pooling layers and have been intensively used for image processing. The convolutional layer allows the CNNs to perform feature-oriented, two-dimensional processing. Each neuron in the convolutional layer receives input only from the local receptive field of the previous feature map layer and reuses the convolutional kernel weights to perform local two-dimensional convolution. In other words, this layer is essentially a cross-correlation operation, followed by a nonlinear activation function, and multiple filters to obtain multiple corresponding outputs equivalent to the features. In addition, the pooling layers resize the feature maps extracting only relevant features. Lastly, a fully connected layer is used to build the final classifier, a highly abstract and low-dimensional information representation[41, 40].

As mentioned above, a common application of neuromorphic computing is image recognition with several papers analysing the performance of created SNNs on common datasets such as MNIST or CIFAR-10. Some of these applications such as texture, edge or corner identification in images implemented also a spiking CNN that uses spiking convolutional kernels. In some cases, the spiking CNNs require a novel error function made of spike trains, based on spiking kernels. Then the synaptic weight can be derived using a learning rule that uses directly the mechanism of error back-propagation. However, Zheng et al.[40] found that the performance of directly trained spiking CNNs is often inferior to that of traditional CNNs but superior when referring to training time. One possible solution is CNN conversion to spiking CNNs as many studies have shown that converted networks work well and perform close to the traditional approach while also consuming less energy[40, 41].

5.2 Learning Algorithms

With a clear description of the spiking neural network structure as well as its neuromorphic components, the focus can be switched to the learning algorithms. These were initially developed and applied to traditional ANNs. Even though their common principles are not altered by neuromorphic computing, implementing them in a network will require special operations. Due to the necessity of differentiability and as spikes introduce a temporal dimension to the error assignment problem, traditional ANN learning algorithms cannot be directly adapted to SNNs. To solve this problem, every learning algorithm created its solutions that will be presented below in their respective section. The first algorithm and the most used in literature and industry is supervised learning (SL), analysed in subsection 5.2.1, which requires mapping the input-output relation for acquiring information. Next, information can also be obtained without the need for input-output mapping and this can be done through unsupervised learning (UL), detailed in subsection 5.2.2. Lastly, learning can be done solely through repetition and reinforcement learning (RL) algorithm exploit that in subsection 5.2.3.

Before presenting the mechanisms of learning algorithms, several general comments must be made. Even though the previous paragraph mentions that "every algorithm created its methods", it is important to state that these methods are not clearly divisible between learning algorithms. The best example is reinforcement learning, which, in some cases, can be classified as an application of supervised learning and thus most of the learning methods presented in subsection 5.2.1 can also be applied to subsection 5.2.3. With this versatility in mind, it is important to mention the categorisation made by Schuman et al.[7] which splits the neuromorphic learning methods into 4 clear categories that can be visualised in Figure 5.3 as follows:

- **Back-propagation**: inspired by the traditional ANN learning method, it is mostly used for supervised learning algorithms, showing broad applicability but being relatively complex to be implemented in neuromorphic architectures due to the differentiability requirement;
- **Evolutionary**: also used preponderantly for supervised learning, this learning method is easier to implement but, compared to back-propagation, it requires more time for convergence;
- **Hebbian**: even though this method was derived for unsupervised learning, several applications used it for supervised learning too; it is more biologically plausible than previous methods but it has not yet been applied largely to industry;
- **Spike-timing-dependent plasticity (STDP)**: it is very similar to Hebbian methods and it is the most commonly used algorithm proposed for training spiking systems; however, it has a more complex implementation than Hebbian algorithms with the literature not specifying a clear learning or training rule;

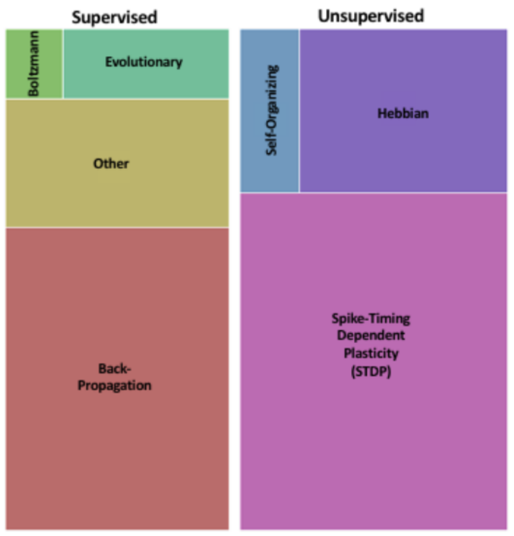


Figure 5.3: A visual summary of training/learning algorithms. The size of the box corresponds to the number of papers in that category[7]

While analysing the learning methods, some general principles and terms were observed to be recurrent throughout every neuromorphic learning algorithm. The first of them is synaptic plasticity, heavily inspired by the biological process, where specific synaptic activity patterns result in changes in synaptic strength. One repeated problem encountered in neuromorphic computing is that neuronal activity can fall silent or saturate when the average synaptic input falls extremely low or rises significantly high. One possible solution involves intrinsic plasticity, found in the biological brain, which regulates intrinsic excitability to promote stable firing within an appropriate range. By changing the neuron's intrinsic properties learned during training, it can achieve the optimal firing rate distribution. Another possible solution involves the normalisation of synapse weights. Thus, when a synapse weight overpasses the maximum threshold, all other synapses are required to decrease their weights. An alternative approach to learning a model involves the ANNs to SNNs conversion which is based on the idea of importing pre-trained parameters such as weights and biases. This way, the heavy computational cost of running ANN models is mitigated by neuromorphic computing which reduces the number of bits per transmission and makes signals sparse in time. However, handling data with spatio-temporal properties cannot be exploited. Converting ANNs to deep SNNs has achieved comparable results to those of original ANNs while being more energy efficient[18].

5.2.1 Supervised Learning

Supervised learning (SL) is a type of learning that uses labelled datasets to train algorithms capable of predicting outcomes and recognising patterns by creating a mapping from the input space to the output space. With high-quality input-output relations, SL can achieve very efficient learning. Unfortunately, this requires the annotation of data, a process that requires abundant human resources. In ANNs, the mapping can be learned using a supervised method through a loss function between the current output of the network and the expected output. The loss function should be minimised and this can be done through back-propagation which updates the weights after every forward pass. To find the minimum of the loss function, various methods exist including stochastic gradient descent or RMSProp but also evolutionary algorithms. The supervised learning algorithm described in the lines above is based solely on offline learning but several papers have shown that online learning is possible too. Moreover, supervised learning has been successfully applied to feed-forward neural networks, recurrent neural networks (but using back-propagation through time) and convolutional neural networks. The successful application of back-propagation has caused a renaissance in the field of research on ANNs, becoming the standard in machine learning.[7, 40]

However, the back-propagation method presented above does not work in neuromorphic computing for several reasons. One of them is rooted in biology and it is called the weight transport problem, which argues that forward and feedback neurons cannot share the same synaptic weights during learning in biological neural networks. Another issue is that spike signals are not continuous which makes the SNNs not differentiable and thus the back-propagation through stochastic gradient descent cannot be simply done. To solve this issue, researchers have created continuous functions that approximate the spiking behaviour[40, 84]. Moreover, the traditional back-propagation method ignores the temporal effects of spikes[135]. To address these issues, researchers created novel approaches to learning information in a supervised manner within SNNs including adaptations or simplifications of the traditional method. Nevertheless, additional challenges emerged over time, such as the restriction of back-propagation to specific types of neuron models, network architectures, and topologies for efficient utilisation, as well as the difficulty of implementation in hardware. Despite all these problems, supervised learning managed to achieve some success in neuromorphic computing being applied for feed-forward networks, RNNs and CNNs[7]. Further, general learning methods applied successfully for supervised learning in SNNs will be listed:

- **SpikeProp**[98]: heavily inspired by back-propagation, SpikeProp was one of the first supervised learning methods for SNNs; it uses the SRM neuron model to avoid the discontinuity problem at the threshold by linearly processing the relationship between the pulse firing time and the membrane potential; SpikeProp takes the time difference between the target pulse and the actual pulse as the objective function and the loss function is calculated as the sum squared error. It has low learning efficiency, mostly because it can only optimise a single neuron connection; to solve this issue, Multi-SpikeProp was created which adjusts the connection between neurons for the multi-connection case; similarly, a new supervised learning rule named MultilayerSpiker was made capable of training multi-layer feed-forward SNNs using stochasticity; moreover, due to its popularity, several variations were brought to this method; for example, in one case, SpikeProp specialised in optimising event-driven data that led to the event-driven random BP (eRBP) rule that uses error-modulated synaptic plasticity for learning deep representations; in other cases, SpikeProp performance was enhanced by adding learning rules based on gradient descent for parameters such as synaptic delays, the threshold of spike firing, and the time constant; inspired by classical back-propagation, SpikeProp was further improved and sped up either by adding a class of nonlinear neuron models or by adding new terms such as momentum, adaptive learning rate, adaptive delays or regularisation; furthermore, SpikeProp was also adapted to RNNs creating thus SpikeProp through time (SPTT) method[84, 44, 18, 41];

- **SuperSpike**[136]: it is an improved method of SpikeProp that utilises the derivative of the membrane potential for optimisation instead of the spike, which allows training a model with an absence of spike occurrence; SuperSpike uses the van Rossum distance between the output and desired spike trains as the loss function[18];

- **Spatio-Temporal Back-Propagation(STBP)**[137]: this algorithm combines both the spatial domain and the temporal domain in the training phase based on approximating the derivatives of impulse functions; STBP achieved high accuracy and could be used on both static and dynamic datasets[84, 41];

- **Tempotron**[121]: inspired by the ANN's perceptron, Tempotron is a single-connection optimisation method that uses the difference between the target output membrane potential and the actual output membrane potential as the objective function; it adjusts the synaptic weights by minimising the sum squared error; an improved method is the Chrontron, which uses the Victor-Purpura distance as the construction basis of the loss function. Further, Chrontron proposed two learning rules: E-learning (based on gradient descent) and I-learning (more biologically explanatory). Experimental results showed that Chrontron converges faster and more reliably than Tempotron and does not rely on a specific reset mechanism[84, 41];

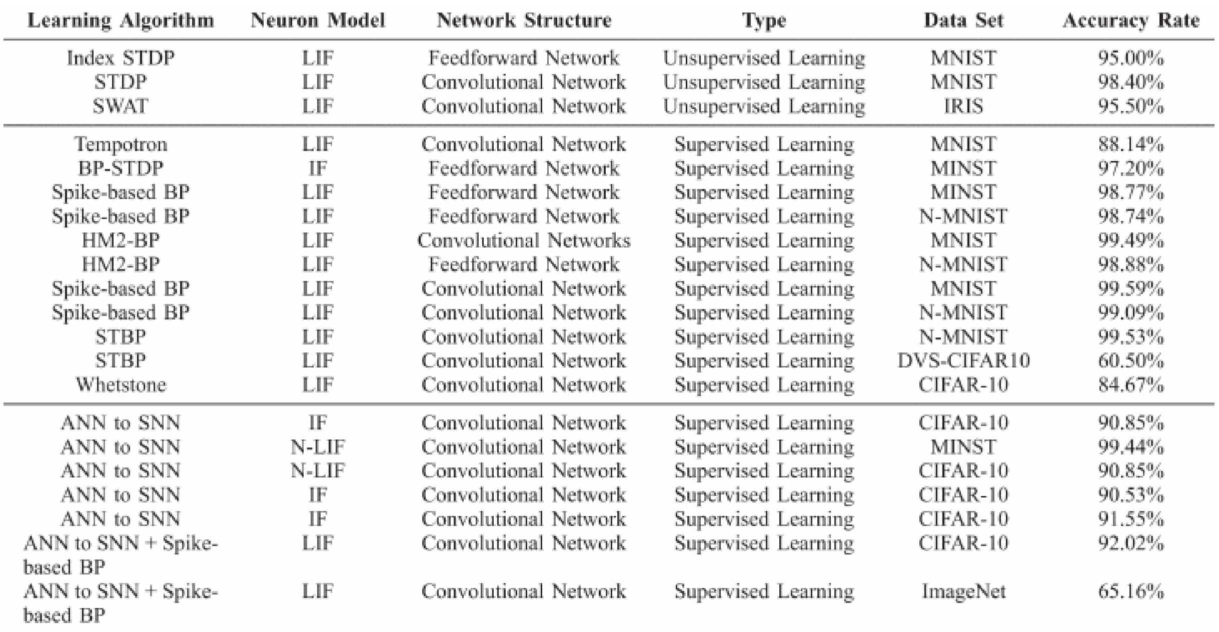
- **Perceptron-Based Spiking Neuron Learning Rule (PBSNLR)**[138]: it is an SNN-to-ANN conversion method that transforms SL into a classification problem and then solves the problem using the perceptron learning rule; some improvements were proposed including dynamic learning parameters (e.g.: threshold) with distance items and a better method for choosing negative samples[41];

- **ReSuMe**: this method combines the unsupervised learning methods of STDP and anti-STDP with remote supervision to minimise the gap between the output and the target without the need for gradient descent; the STDP and anti-STDP methods work after the Widrow-Hoff rule which continuously adjusts the weights and thresholds of the network in the direction of the fastest reduction of the sum of squared errors; the ReSuMe algorithm can also learn the mapping of RNNs, being capable of altering the synaptic weights of the liquid state machine; a variation of this method includes the delay-learning remote supervised method (DL-ReSuMe) in which synaptic weights are updated by the delayed version of the ReSuMe rule; as ReSuMe can only be applied to single connections, Multi-DL-ReSuMe extended the applications to more neurons by proposing a cross-correlated delay shift (CCDS) method, in which synapse and axonal delays are modulated together with weights during learning; in addition, T-ReSuMe is an improved learning algorithm that combines the ReSuMe algorithm with triplet-based STDP too[44, 84, 41];

- **Spike Pattern Association Neuron (SPAN)**[139]: it is another single-connection optimisation method created for learning in spiking CNNs; the idea of the SPAN algorithm is to transform the spike trains during the learning phase into analogue signals using an α-kernel function such that the Widrow-Hoff rule can be applied directly to the transformed signal to adjust synaptic weights; inspired by the SPAN algorithm, a precise-spike-driven (PSD) supervised learning rule can be used to associate a spatio-temporal spike pattern input with a desired spike train by the double-exponential kernel function[84, 41];
- **Spike-based Back-Propagation**[140, 141]: this algorithm treats the membrane potential of spiking neurons as a continuous differentiable signal, where the discontinuity in the spiking time is regarded as noise; spike-based BP follows the error back-propagation mechanism of traditional ANNs but acts directly on spikes and membrane potentials; one special case is the surrogate gradient descent that uses surrogate derivatives to define the derivative of the threshold-triggered firing mechanism; this approach achieved remarkable robustness while having no restrictions on simulating time steps compared to ANN-to-SNN conversion as it is not based on rate coding[84, 97];
- **SLAYER**[142]: it assumes a stochastic spiking neuron approximation for the IF model with a refractory response and can simultaneously learn both synaptic weights and axonal delays; to solve the drawback of event-based methods, SLAYER distributes the credit of error back in time[18];
- **Prescribed Error Sensitivity (PES)**[143]: suited for online learning, this algorithm learns a function by minimising an external error signal frequently used with the neural engineering framework[18];
- **Hybrid Macro-Micro Level Back-Propagation (HM2-BP)**[144]: it performs error back-propagation on both macro-level trigger frequencies as well as micro-level pulse sequences; then HM2-BP directly computes the gradient of the loss function for frequency encoding with adjustable parameters spatio-temporal back-propagation (STBP) algorithm in SNN based on approximating the derivatives of impulse functions[84];
- **Spike Train Kernel Learning Rule (STKLR)**[145]: it was created for spiking CNNs to translate kernel functions to be adapted into neuromorphic computing; various kernel functions were transformed using this algorithm and it was even adapted to RNNs with the help of the R-STKLR algorithm; moreover, STKLR was extended to more neurons leading to the multi-STIP algorithm[41];
- **Feedback-based Online Local Learning Of Weights (FOLLOW)**[146]: created for both feed-forward and recurrent SNNs, the error of the network is fed back through fixed random connections with a negative gain, which causes the network to learn the desired dynamics[41];
- **Whetstone algorithm**[147]: this algorithm implements an ANN-to-SNN conversion, in which the activation function of each layer is gradually approached towards the threshold activation during training[84];

As can be seen, a multitude of algorithms for supervised learning were developed and choosing only one for training an SNN depends on several factors. For example, some of them were tailored for specific applications or specific network algorithms such as CNNs or RNNs. Despite these factors, a comparison can still be made by looking at accuracies achieved by these methods. Wang et al.[41] found that, for single connection optimisation, the spike train learning performance of the SPAN algorithm is the best, while that of the SpikeProp algorithm is the worst. Likewise, for learning a multi-layer SNN, the Multi-STIP algorithm, inspired by STKLR, achieves the highest learning accuracy for the shortest training time. Moreover, this algorithm along with Multi-ReSuMe has a relatively short running time making the Multi-STIP algorithm suitable for training large-scale feed-forward SNNs. For RNNs, it was found that the R-STKLR algorithm achieved a higher accuracy while the FOLLOW algorithm required less training as well as a slightly shorter running time.

Furthermore, He et al.[84] analysed the accuracy of various learning algorithms on classical image classification datasets and can be seen in Figure 5.4. For the MNIST dataset, the best performances were achieved by the Spike-Based Back-Propagation algorithm followed closely by HM2-BP applied both on convolutional networks. For the N-MNIST dataset, it was found that Spatio-Temporal Back-Propagation achieves the best accuracy followed closely by Spike-Based Back-Propagation. Otherwise, single-connection learning algorithms such as Tempotron achieved the worst performance while, on more complex datasets such as CIFAR-10 or ImageNet, ANN-to-SNN conversion achieved inferior results.



Learning Algorithm	Neuron Model	Network Structure	Type	Data Set	Accuracy Rate
Index STDP	LIF	Feedforward Network	Unsupervised Learning	MNIST	95.00%
STDP	LIF	Convolutional Network	Unsupervised Learning	MNIST	98.40%
SWAT	LIF	Convolutional Network	Unsupervised Learning	IRIS	95.50%
Tempotron	LIF	Convolutional Network	Supervised Learning	MNIST	88.14%
BP-STDP	IF	Feedforward Network	Supervised Learning	MINST	97.20%
Spike-based BP	LIF	Feedforward Network	Supervised Learning	MINST	98.77%
Spike-based BP	LIF	Feedforward Network	Supervised Learning	N-MNIST	98.74%
HM2-BP	LIF	Convolutional Networks	Supervised Learning	MNIST	99.49%
HM2-BP	LIF	Feedforward Network	Supervised Learning	N-MNIST	98.88%
Spike-based BP	LIF	Convolutional Network	Supervised Learning	MNIST	99.59%
Spike-based BP	LIF	Convolutional Network	Supervised Learning	N-MNIST	99.09%
STBP	LIF	Convolutional Network	Supervised Learning	N-MNIST	99.53%
STBP	LIF	Convolutional Network	Supervised Learning	DVS-CIFAR10	60.50%
Whetstone	LIF	Convolutional Network	Supervised Learning	CIFAR-10	84.67%
ANN to SNN	IF	Convolutional Network	Supervised Learning	CIFAR-10	90.85%
ANN to SNN	N-LIF	Convolutional Network	Supervised Learning	MINST	99.44%
ANN to SNN	N-LIF	Convolutional Network	Supervised Learning	CIFAR-10	90.85%
ANN to SNN	IF	Convolutional Network	Supervised Learning	CIFAR-10	90.53%
ANN to SNN	IF	Convolutional Network	Supervised Learning	CIFAR-10	91.55%
ANN to SNN + Spike-based BP	LIF	Convolutional Network	Supervised Learning	CIFAR-10	92.02%
ANN to SNN + Spike-based BP	LIF	Convolutional Network	Supervised Learning	ImageNet	65.16%

Figure 5.4: Accuracy of various neuromorphic learning algorithms for image recognition on different datasets from the paper of He et al.[84]

In summary, current research on SNNs has focused on adapting the core optimisation algorithms from ANNs. The results show that the current SNN algorithms perform well on relatively small datasets but poorly on large datasets such as ImageNet and are still far from the mainstream ANN algorithms. However, finding an SNN optimisation algorithm that is both bio-interpretative and efficient is necessary to exploit the properties of SNNs fully[84]. Nevertheless, there seems to be a great variety of applications using supervised learning. On-chip supervised weight training has been used for least-mean-squares algorithm, weight perturbation and CNN training but also hardware implementation such as Boltzmann machines or deep belief networks and hierarchical temporal memory. Even evolutionary learning such as genetic algorithms or particle swarm optimisation has been trained both online and offline[7]. All these applications focused on general problems but supervised learning for SNNs was also applied for a great variety of problems. Out of these, a few will be named such as image classification with spike-based back-propagation[97], angular velocity regression with SLAYER[135], adaptive control of quadrotor flight with PES[18], classification with ReSuMe[84], real-time user authentication with PBSNLR, image recognition and classification with STKLR and standard XOR problem with Multi-SpikeProp[41].

5.2.2 Unsupervised Learning

Unsupervised learning (UL) is an algorithm for learning patterns from unlabelled data, which is prevalent in biological systems. At the same time, this learning approach hopes to solve the resource problem of SL which requires intensive data labelling in great quantities and of good quality. Moreover, it was found that UL manages to discover hidden structures in the data. Due to these 2 features, it has become a research hotspot in recent years even in the traditional ANN realm. Thus numerous unsupervised learning algorithms were created to obtain patterns from unlabelled data. For example, autoencoders were created to compress the input information or reduce the dimensionality of the input space. These were also applied to neuromorphic computing for encoding as detailed in section 3.4. Similarly, UL also helps to create self-organising maps, a method capable of producing a low-dimensional representation of the input space. Another application of unsupervised learning includes the generation of data such as image creation or writing creatively. To do this, Generative Adversarial Networks (GANs) were created which were detailed in subsection 5.1.1[40].

In the domain of neuromorphic computing using unsupervised learning, it is important to mention that this approach finds its closest analogue in the learning processes observed in biological systems. In the brain, it is used preponderantly for information encoding in biological visual systems such as recognition and comparison of the outside world without the need for labelling[72]. However, looking at neuromorphic computing, unsupervised learning rules are still in their infancy when compared to SL[40]. However, as is the case with biological systems, these self-learning training algorithms will almost certainly be necessary to realise the full potential of neuromorphic implementations and thus fully reach its advantages. With this in mind, several implementations of on-chip unsupervised training mechanisms in neuromorphic systems have been done, most of them inspired by the spike timing-dependant plasticity (STDP), as it will follow from the list below:

- **Hebbian STDP**[148]: it is an unsupervised learning mechanism, that adjusts synaptic weight based on the temporal order of the pre-and post-synaptic spikes; when the pre-synaptic spike arrives before a post-synaptic spike, the synaptic weight is increased, which is known as long-term potential (LTP); otherwise, the synaptic weight is reduced, which is known as long-term depression (LTD); the current rule seems to have no boundary on the synaptic strength which is capable of being increased or decreased infinitely that makes it biologically unrealistic; this is solved with a Winner-Take-All (WTA) mechanism; this form of competition implies that when a neuron fires a spike and the presynaptic weights are updated, the rest of the postsynaptic cells (from the same layer) locally connected to the same input neurons get inhibited; as a result, these cells are prevented from triggering STDP while the neuron that fired first, remains in the refractory period; furthermore, STDP rules do not include attenuation mechanisms or enhancement thresholds for synaptic connections, so the model is susceptible to noise and not stable[18, 72, 84];
- **Anti-Hebbian STDP (aSTDP)**: aSTDP shows the opposite dependence on the relative timing of pre-synaptic input and the post-synaptic spike compared to Hebbian STDP; with aSTDP, pre-synaptic activity occurring before post-synaptic activity leads to depression, and vice versa; this algorithm was inspired from biology where not all systems follow the Hebbian STDP synaptic weight modification rules but some follow a different order such as synapses between parallel fibres and Purkinje-cells in the cerebellum-like structure[18];
- **Mirrored STDP (mSTDP)**[149]: introduced as an effort to implement autoencoders in a biologically realistic fashion, mSTDP combines STDP and aSTDP for feed-forward and

feedback connections of a two-layer autoencoder such that the layers are symmetric; this learning rule accounts for a high LTP correlation with no causality which leads to a low biological plausibility[18];

- **Triplets STDP (tSTDP)**[150]: in this algorithm, LTP is constructed as a combination of one presynaptic and two postsynaptic spikes, while LTD is based on the combination of one presynaptic and one postsynaptic spikes; one advantage of tSTDP is that it is not affected by the spike-timing interactions as the relative timing is more accurate by adding the third "observer" spike[40];
- **Probabilistic STDP (pSTDP)**: [151]: as mentioned above, STDP formulations change the weights based on the relative timing between spikes; these algorithms are referred to as additive rules and are inherently unstable, requiring the use of constraints for the weights; to solve this issue, multiplicative STDP rules were created which incorporate the current weight values in changing the weights themselves; moreover, by incorporating the weight dependency in an inversely proportional manner, stable, robust distributions are obtained irrespective of the complexity of the network; this method represents the current state-of-the-art in pattern recognition with SNNs[72, 18].
- **Bienenstock-Cooper-Munro (BCM)**[152]: this method assumes that the neuron determines the threshold at which the synaptic weight changes direction; moreover, BCM allows the threshold to dynamically adapt to the neuron's historical activity so that the connection weight eventually reaches a steady state; to measure neuron activity, the firing rate is required which makes this method compatible with rate encoding[84];
- **Synaptic Weight Association Training (SWAT)**[153]: it combines the variable threshold feature of the BCM rule with the feedback given by the STDP plasticity window; this method enhances the stability of SNNs during training offering SWAT good generalisation ability[84];

With many models presented, a comparison between them based on their performance should be executed. Unfortunately, the literature showed very little applicability, with He et al.[84] comparing some applications of unsupervised learning for image recognition, summarised in Figure 5.4. They found that Hebbian STDP as well as BCM can achieve good results on small datasets (MNIST, IRIS) but the performance is still lower than supervised learning algorithms. Despite this lack of success, researchers applied unsupervised learning for tasks such as self-organising maps, self-organising rules or expectation-maximisation algorithms but also for accurately reproducing biological systems[7, 97]. Looking at more practical applications, unsupervised learning was used for estimating optical flow with the help of probabilistic STDP[72], for clustering and segmenting images and for detecting edges with Hebbian STDP[85] and for an asynchronous feed-forward spiking neural network that mimicked the ventral visual pathway using Hebbian STDP rules[84].

5.2.3 Reinforcement Learning

Reinforcement learning (RL) is the last presented machine learning method that functions by teaching an agent how to take action in a dynamic environment to maximise the final reward. It differs from SL as it does not need labelled data and from UL as it should achieve a predefined goal. The RL focuses on finding a balance between exploration (what SL lacks) and exploitation (what UL lacks) to maximise the long-term reward. To achieve this performance, RL learns by receiving feedback over iterative trials that are sequential and evaluated through the use of powerful nonlinear function approximations. Based on these rules, a great variety of algorithms were created that can be applied to different tasks based on a series of factors: continuous vs discrete domain, off-policy (follow the most advantageous action irrespective of the policy) vs on-policy (follow the policy all the time), model-based (already have some rules in place) vs model-free (figure out from scratch how it works) or policy-based (policy is learned as a mapping from the state space to the action space) vs value-based (choose an action based on the value function). Based on the problem that has to be solved, the characteristics can be chosen and thus the best RL algorithm too[40].

The inspiration for reinforcement learning roots deeply in biology where it is heavily used for learning motor control tasks[72]. Moreover, RL was found to act on multiple regions of the brain, resulting in changes in the connectivity of the neural networks. This works by creating reward or punishment signals induced by dopaminergic, serotonergic, cholinergic, or adrenergic neurons. Based on this, agents gradually develop expectations of stimuli in response to rewarding or punishing stimuli given by the environment, producing habitual behaviours that yield the greatest benefit. For example, dopamine neurons enable this function, comparing future expectations with previous mental benchmarks and thus releasing neurotransmitters depending on the result. This makes the creature happy or frustrated, using a reward mechanism as the basis for learning. Switching to neuromorphic computing SNNs have great potential in reproducing reinforcement learning rules based on the performance found in the human brain[97, 40]. But until now, few RL algorithms have been implemented:

- **Reward Modulated STDP (rSTDP)**[154]: while STDP operates based upon the correlation between the spike timings of the pre-and post-synaptic neurons, a reward signal is introduced to modulate STDP to implement a reinforcement learning mechanism; if the reward is positive, the corresponding synapse is reinforced; otherwise, the corresponding synapse is weakened[18, 155];
- **Three-factor Learning Rules**[156]: this approach works by setting a flag, called an eligibility trace, on the synapse upon co-activation of presynaptic and postsynaptic neurons; if a third factor (indicating reward) is present when the flag is set, synaptic weights change[40];
- **Spiking Actor-Critic**[157]: the Actor-Critic algorithm is inspired by the Generative Adversarial Network as it contains 2 networks: the actor and the critic; the actor decides which action should be taken and the critic informs the actor how good was the action and how it should be adjusted; the learning of the actor uses a policy gradient approach while critics evaluate the action by calculating the value function; for neuromorphic computing adaptation, the same network structure is used but LIF neurons with temporal coding are incorporated;
- **Spiking Q-learning**[158]: Q-learning is a model-free reinforcement learning algorithm that finds an optimal policy by maximising the expected value of the total reward over all successive steps, starting from the current state; given infinite exploration time and a partly random policy, it can find the absolute optimal action selection; the neuromorphic adaptation uses the membrane voltage of non-spiking neurons as the representation of the Q-value.

As can be observed few learning methods were created in the realm of neuromorphic reinforcement learning and the existent ones are relatively novel. Moreover, as is the case for traditional RL, each method is tailored for specific applications making the performance comparison even more irrelevant. And this is not surprising as the neuromorphic neurons are inherent time dependent and performing reinforcement learning on sequences of information was found to be problematic in the research field. Nevertheless, RL applications were mainly focused on neuroscience research, with only several goal-directed navigation problems and digit recognition applications[72]. For example, the reward-modulated STDP method was used to solve a temporarily coded XOR problem with a delayed reward as well as a Morris water maze puzzle. Next using the spiking Q-learning algorithm, Chen et al.[158] learned how to play the Atari game. Similarly, the spiking Actor-Critic algorithm was used for classical RL environments such as mountain car, cart-pole, and acrobot problems as well as for learning a UAV to fly through a window and avoid a flying basketball task[157, 155]. A more popular approach to neuromorphic reinforcement learning involves simply learning the logic with an SNN while keeping the traditional RL structure. This approach could achieve good performance on much more complex tasks such as 3D Visual Navigation for MAV With Depth Camera[68].

5.3 Performance Metrics

After building a spiking neural network, its performance should be assessed. This process might involve comparison with existing models or simply self-analysis to understand what has been learned in the model and how it can be improved. The comparison can be performed both to other spiking neural networks (mostly to check if other formulations are performing better) or to other artificial neural networks (to assess the advantages and disadvantages of implementing neuromorphic computing). In any case, some clear metrics to have an objective comparison are needed that are listed below:

- **Accuracy**: it is probably the most important metric that is capable of describing in general terms how good the network performance is; it could be used for both SNN-to-SNN and SNN-to-ANN comparison as well as for training and quantifying the performance of the SNN itself; for example, it was used by Sanaullah et al.[106] to quantify the classification performance of SNN and compare various neuron models at the same time; in this case, the accuracy was calculated as the mean of the correctly classified samples or data points by a model; using the same testing method, Wang et al.[41] applied the accuracy metric to compare various learning methods for supervised learning; the accuracy was also implemented by Meftah et al.[85] for evaluating the performance of image segmentation; looking at the accuracy of pixel detection, several metrics were proposed such as precision (used if false positives are a concern), recall (used if false positives are a concern) or the F1 score (if a balance between precision and recall is required); Deng et al.[159] used top-1 accuracy to compare ANN to SNN for the visual recognition task which is measured as the percentage of the correctly recognised samples;

- **Performance loss**: complementing the accuracy, this metric quantifies the error of the model's predictions from the ground truth or desired output; it provides essential information for model selection, optimisation, and understanding of the behaviour of SNNs in practical applications; for classification problems, this can be calculated as simply the error rate which is the percentage of misclassified samples[106]; it can be better generalised for more variate data with the help of Mean Square Error (MSE) and Mean Absolute Error (MAE); for image clustering and segmentation, more metrics were proposed by Meftah et al.[85] such as Peak Signal Noise Ratio (PSNR) or Normalised Colour Difference (NCD);

- **Spiking activity**: this metric serves for analysing and understanding spiking neural networks as it describes the dynamic behaviour of neurons; it provides insights into the network's temporal characteristics, spike patterns, firing rates and information processing capabilities[106]; the spiking activity can indicate neurons that fail to generate spikes, allowing for their identification and removal from the network; to quantify this parameter, Dupeyroux et al.[73] proposed the infill percentage of a spiking sequence as the ratio of the number of spikes within a layer to the maximum of spikes in that layer; they used this metric to compare the behaviour of 2 SNNs: one simulated and one implemented on neuromorphic hardware;

- **Training time**: it can be used both to compare 2 SNN models and for getting insights into the network's training performance; it can be measured simply as the total time required to train a model;

- **Running time**: it describes how fast the network can perform a task and can be used for any network comparison; this parameter can reveal information about the network's capable frequency of generating an output; to calculate it accurately, it is important to verify the total number of operations, the network has to perform; in ANNs, the operational cost is mainly determined by the MAC operations, which are widely used in ANN accelerators; in SNNs, the major operational overhead is the spike-driven input integration but the costly multiplication can be removed due to the binary spike inputs; moreover, the integration is event-driven, implying no computation occurs if no spike is received[159]; Lemaire et al.[160] developed a system of equations to calculate the operational cost of both ANNs and SNNs for both feed-forward architectures and convolutional architectures focusing on addition and multiplication operations;

- **Memory cost**: This metric is critical for embedded devices as the hardware might prohibit the network implementation; it can be used to compare both ANN-SNN and SNN-SNN model performances; in ANNs, the memory cost includes the weight memory and activation memory; in SNNs, the memory cost includes weight memory, membrane potential memory, and spike memory; other parameters such as the firing threshold and time constant are negligible since they are usually shared in the network[159]; Lemaire et al.[160] have developed a system of equations to calculate the memory cost that involves the read operations to inputs, parameters and potentials as well as write operations to potentials and outputs;

- **Energy efficiency**: technically, this metric adds the results of the operational cost metric and the memory cost metric and it can calculate energy consumption; it is mostly used for ANN-to-SNN comparison as it shows the clear advantages of SNN due to its low energy consumption; Lemaire et al.[160] formulated a system of equations to calculate this metric including the operational and memory parameters and assigning a standard energy consumption for each of them;

These parameters can serve as a good start in describing some neuromorphic capabilities. As mentioned by Deng et al.[159], more evaluation metrics might be needed to assess the capability of temporal association, memorisation capacity, fault tolerance, and practical running efficiency on devices. Moreover, a great number of parameters were found in papers that quantify general aspects of neuromorphic computing but they will not be considered further as they find little applicability to the current study. For example, a prevalent metric mentioned in the section above is biological plausibility which describes how closely a neuron model resembles a biological neuron. Some hardware metrics were also found such as the maximal output and input frequency of neurons or the power required by one neuron.[41].

6 Research Proposal

Having examined the technical background, it is crucial to shift the focus towards the practical aspects of the current study. Until now, the paper analysed how a neuromorphic computing architecture works, how it can be built and what are its advantages and disadvantages. Several drone applications where this technology was used have also been explained and will serve as inspiration for the current work. But before making a decision, it is important to analyse the background of the paper such that the current high-speed drone realm is understood and practical limits are imposed in section 6.1. With these in mind, the objective of the research will be detailed in section 6.2 followed by some questions that will have to be answered at the end of the research period presented in section 6.3.

6.1 Current Development

This section aims to narrow down the selection procedure of the research proposal such that the following sections will focus on its details. The procedure will start with the background information of the research and the requirements imposed such that a feasible research topic will be selected. Thus the research is part of the Control & Simulation Master's programme of the Aerospace Engineering faculty of the Delft University of Technology. This leads to a strict rule that the research has to be conducted for approximately 9 months (excluding holidays). Moreover, due to their focus on artificial intelligence and neuromorphic computing applied to aerospace engineering, it was decided to continue the research with the MAV Lab[1]. This department is mostly focused on operating Micro Aerial Vehicles (MAVs) including the flight of small aircraft ranging from DelFly, a flapping wing aircraft weighing around 16 grams to much larger drones. As mentioned above, the research of MAV Lab is mostly focused on artificial intelligence which includes numerous research topics such as autonomous flight, collision avoidance and swarming.

One research topic very active in the MAV Lab department that stands out with vibrant activity is racing drones. As they require cutting-edge technology and relentless optimisation to push the speed boundaries, racing drones have also a remarkable degree of developmental freedom. As mentioned, optimisation is at the forefront of improving the performance of racing drones which is often facilitated by neural network control techniques. Through these methods, a precise control law is created for the dynamical system, that is built upon the optimisation of an objective function (in the case of racing drones, the pursuit of reducing lap times). This approach was heavily studied in MAV Lab starting with the creation of the world's smallest autonomous racing drone[161]. Over time, multiple optimisation research ideas were introduced. Ferede et al. found that training an adaptive neural network for the end-to-end guidance and control of a racing drone achieved better results than state-of-the-art methods such as Differential-flatness-

[1]https://mavlab.tudelft.nl/

based controller (DFBC) or Incremental Non-linear Dynamic Inversion (INDI) using both supervised learning[22] and reinforcement learning[23]. Another approach studied by Westenberger et al.[162] suggested doing time optimal control with a bang-bang model predictive method in which the helices are either set to full power or turned off and thus only the switching time needs to be found.

A novel domain that could bring many advancements to the racing drone field is the neuromorphic computing realm itself. The promises this technology keeps as mentioned previously regarding concerning energy consumption and processing power makes it a research hotspot. For this reason, numerous research papers that study neuromorphic computing were created inside the MAV Lab starting from a general toolbox developed for encoding and decoding input signals[87]. Similarly, many researchers focused on drone control with neuromorphic computing reproducing the technology of classical controllers such as PI[75], PID[66] or using input-weighted threshold adaptation[65]. Further, neuromorphic vision information was introduced in the control process of the drone. Thus optic-flow-based[73] and high-speed divergence-based[74] landings of MAVs, were implemented as well as more complex tasks such as autonomous flying using both simple control laws[75] or self-supervised learning[163]. Additionally, neuromorphic estimation of the attitude using simply the IMU measurements onboard quadrotors was conducted[63] and some of these researches were also implemented on neuromorphic hardware (namely Loihi)[73, 163, 66]. Even though numerous studies were created for MAV control, there still seems to be a missing link when it comes to performing guidance and control for high-speed drones using neuromorphic computing.

Before continuing the analysis, it is important to highlight the resources offered by the MAV Lab that could prove useful during the study period. Looking at the software realm, it is worth mentioning the Technological University of Delft GitHub platform which contains a neuromorphic toolbox repository that could be used for developing spiking neural networks (SNNs)[2]. Similarly, a clear framework was created for testing drone simulations using the Paparazzi software project[3]. Regarding hardware, if real testing is desired, MAV Lab offers the Bebop drone also used in the works of Ferede et al. where a dynamical model can also be found[22, 23]. Even though this drone was not specifically designed for autonomous drone racing, it still manages to reach saturation of its controls in a relatively small flight space while also being safer to test with. The Bebop drone is also equipped with an MPU6050 IMU and a Parrot P7 dual-core Cortex A9 CPU which is used to run the code in real-time onboard the drone. Even though this is not neuromorphic hardware, this drone can still simulate neuromorphic behaviour. The schematic presented throughout the current paragraph required for testing the drone performance can be visualised in Figure 6.1.

6.2 Research Objective

As mentioned above, even though neuromorphic computing seems to be a clear solution for increasing the efficiency and performance of racing drones, at the intersection of these 2 realms still lies a critical missing link: the guidance and control implementation with spiking neural networks. In other words, the operation of racing drones typically necessitates two core modules: vision/sensors for the perception of the environment and a control law for fast guidance and manoeuvring of the drone. While significant attention has been devoted to refining vision capabilities, including the integration of event-based cameras and sensor fusion techniques in drones, the realm of neuromorphic rapid guidance and control remained absent from the literature. This absence presents a great obstacle to the realisation of neuromorphic high-speed drones, but by successfully addressing this integration challenge, new opportunities may emerge,

[2]https://github.com/tudelft/spiking
[3]https://wiki.paparazziuav.org/wiki/Main_Page

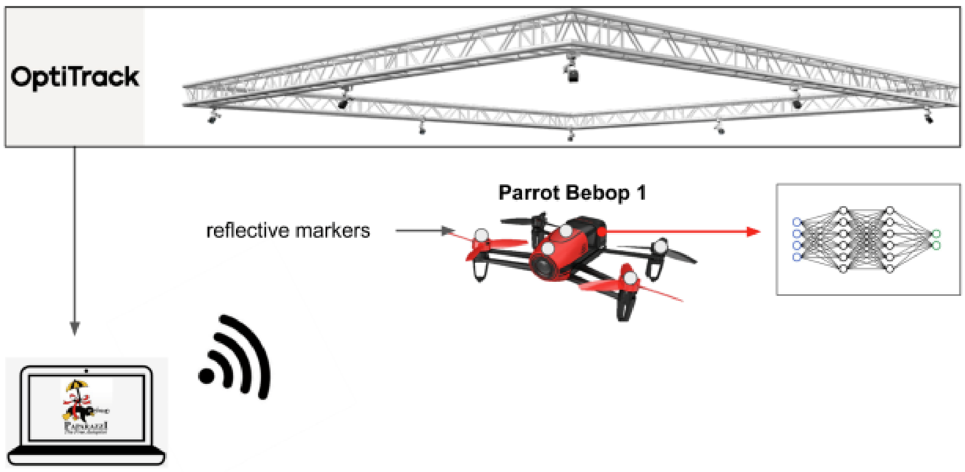


Figure 6.1: The schematic provided by Technological University of Delft for testing the performance of the drone in real flight

enabling further expansion of MAV boundaries.

However, the convergence of these two domains gives rise to several challenges, which explains the lack of research aimed at replicating it. On one hand, rapid control demands cutting-edge technology but with robust and clear laws for pushing guidance and navigation solutions to the boundaries of feasibility. On the other hand, neuromorphic computing is marked by its inherent instability and there is a brief understanding of how it functions. Moreover, despite the promises of neuromorphic computing regarding high energy efficiency and powerful computational power due to its parallelism, its implementation on racing drones might hinder performance as great quantities of data will have to be processed which is not favourable for this novel technology. Given these complexities, the research approach must proceed accordingly, prioritising simplicity and feasibility within the constraints of available resources and time. Therefore, the available resources of the MAV Lab will be used and the work of Ferede et al. for end-to-end neural network-based quadrotor rapid guidance and control[22] will serve as inspiration for the current study.

Throughout the research, the most complex and accurate model learned by Ferede et al. will be addressed based on the end-to-end reinforcement learning algorithm for generating rapid quadrotor flight[23]. The paper contains both a physical model of the drone and its environment as well as a complex model to learn the drone's behaviour. Thus, the created algorithm focuses on learning how to fly optimally through a rectangular circuit with the following corners in (x, y) format position on the map as can be visualised in Figure 6.2: (-2, 2), (-2, 2), (2, 2), (2, -2). To learn such a model, the algorithm receives several inputs about the drone's state and environment as follows: the 3-dimensional position (x, y and z), velocity (v_x, v_y, v_z), angular displacement (ϕ, θ, ψ) and angular velocity (p, q, r) of the drone as well as the actual angular velocity of the propellers of the drone (ω_1, ω_2, ω_3, ω_4). Regarding the environment, Ferede et al. found that to improve performance, it is necessary to add random disturbance in the learning process simulated as a 3-dimensional momentum ($M_{ext,x}$, $M_{ext,y}$, $M_{ext,z}$) and as a force in the vertical z-direction ($F_{ext,z}$) as well as sending the next gate 3-dimensional position ($gate_x$, $gate_y$, $gate_z$) and yaw ($gate_{yaw}$). Lastly, the output is expressed as motor commands the drone needs to generate for the 4 propellers registered as u_1, u_2, u_3, u_4. For a better understanding of the reference system used for the drone as well as the numbering of the propellers, Figure 6.3 can be checked. All these 28 values are used as floating points (32 bits) in the code and will serve as a starting point for the current research.

The future research work will be influenced by pragmatic considerations and the necessity to

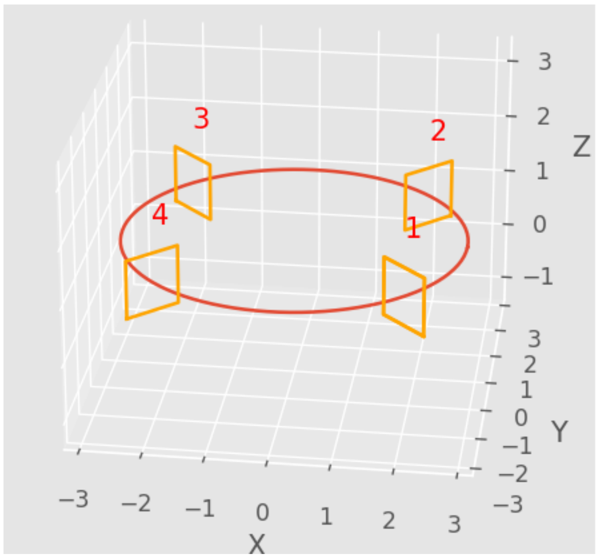


Figure 6.2: The track used by Ferede et al.[22, 23] for testing

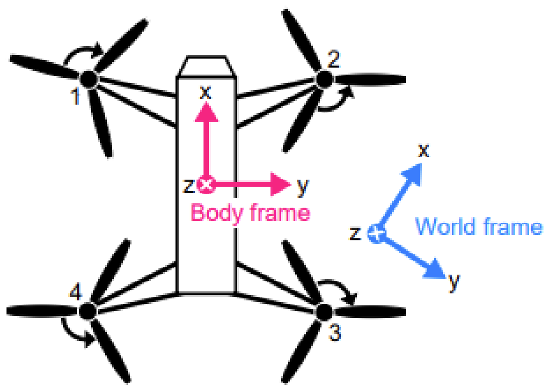


Figure 6.3: The reference system of the drone as well as the numbering order of the propellers[22]

reach achievable progress against heavy time constraints. Thus the broader goal of the research will be to establish a robust foundation for future advancements in neuromorphic high-speed drone technology. The first decision was to leverage readily accessible simulation tools and hardware platforms such as Paparazzi software and the spiking repository of Technological University of Delft mentioned above, alongside possible testing implementation on the Bebop drone.

Consequently, for the training procedure, a few decisions could be taken to speed up the research without degrading the conclusions leading to a neural network structure that is easy to implement and achieves good performance. Even though they have low biological plausibility (which is of little importance for the current research), it was decided to continue with LIF neurons connected by simple synapses. This comes as these 2 components were found to achieve good learning accuracy with low computational resources. Mostly due to time constraints and ease of understanding, the network will use a fully connected feed-forward structure, studied intensively in the literature. This network will use a supervised learning approach while the back-propagation will be solved using an approximation function (spike-based back-propagation with surrogate gradient descent) due to its good performance and ease of implementation. With all these in mind, the research objective could be summarised as:

The research objective of the current paper is to assess whether a Spiking Neural Network architecture can preserve state-of-the-art performance for rapid quadrotor guidance and control.

6.3 Research Questions

To best tackle the research objective, several key stages were established, each aimed at addressing a specific major component of the research. They were inspired by the learning process of a spiking neural network which led to the following 3 research questions, each addressing a particular hypothesis:

- **RQ.1: How can the input/output values be transformed into spikes such that the lowest resources are used?**

This question addresses the first stage of learning with a spiking neural network that requires transforming values into spikes or, in other words, the encoding procedure addressed in chapter 3.

As the research tries to achieve a higher performance for racing drones, it is crucial to make the encoding efficient. Thus, the research purpose of the current stage will be to express the input and output values in as few spiking neurons as possible while preserving good accuracy. Rate coding cannot be a solution for the current research as it usually requires longer times for processing (more spikes have to be sent to express a value) which is not desired for high-speed drones that require fast decisions. Similarly, temporal coding was not considered due to its infancy in neuromorphic computing which might lead to a multitude of problems for the short time available to the research. Lastly, population coding and autoencoders remain viable options. In the end, due to time constraints, it was decided to continue with the autoencoder approach as it was easier to implement for the multitude of variables involved. Moreover, the autoencoder is expected to find better ways to compress the information as population encoding is better concerned with accuracy per value.

To learn the encoding with the help of autoencoders, it is necessary to create a representative dataset following the model created by Ferede et al.[23]. As mentioned above there are 28 floating point values that have to be transformed which, for perfect accuracy will require 896 spiking neurons by assuming that one spiking neuron is responsible for 1 bit. However, accurate reproduction is not necessarily desired but the large number of neurons is a great problem for efficiency which should be solved by the autoencoder. Another reason for selecting autoencoders is that they are expected to save only the important information from all the values. For example, the gate position and yaw can take only a discrete number of values and thus a few neurons are required while other more complex parameters will require more neurons. Additionally, autoencoders will look at the relations between variables in time. For example, it is expected that the velocity will depend mostly on the position variance. These actions cannot be done by population encoding that encodes unimportant information at every time step. With these in mind, the following hypothesis could be drawn:

H.1: *With a variate and detailed dataset, autoencoders can transform values to spikes while compressing the information and thus using the least number of spiking neurons.*

- **RQ.2: How should a spiking neural network be implemented for rapid quadrotor guidance and control?**

The second stage of the research will involve the training procedure of the spiking neural network itself which will address the selection of the learning algorithm as detailed in section 5.2. This part of the project will imply that an analysis will be done that will consider the feasibility and accuracy of the algorithms as well as the implementation time. It was decided to continue with the same approach as Ferede et al. and implement a supervised learning algorithm. Due to its unpredictable nature and difficult implementation, unsupervised learning was discarded. Similarly, reinforcement learning was not considered due to its infancy in the neuromorphic domain and the great time and resources required to implement such an approach. Moreover, implementing reinforcement learning on a neural network that is recurrent has proved to be a challenging task. The current research question will also consider if it is feasible to do a direct ANN-to-SNN conversion or, in other words, if a direct adaptation of the code of Ferede et al.[22] to neuromorphic computing is possible. As supervised learning will be used, a thorough input-output dataset is required and this will require further analysis in chapter 7. In the end, the best dataset will be implemented for learning the guidance and control law of high-speed drones. This will lead to the following hypothesis:

H.2: *A spiking neural network can be implemented and can learn how to guide and control a rapid quadrotor.*

- **RQ.3: Compared to state-of-the-art models, how is introducing neuromorphic computing affecting performance?**

The last stage of the research will involve quantifying the performance of the created spiking neural network model and the comparison with existing non-neuromorphic solutions. The performance metrics will be selected from section 5.3 and will mostly involve the accuracy of the model as well as the computational efficiency. To obtain these results, the learned model will be implemented in both a simulation environment (the Paparazzi framework) as well as on a real drone (the Bebop drone). In any case, due to time constraints, it was decided that the learned model would be implemented in a von Neumann architecture and not on neuromorphic hardware. This factor will not favour the spiking neural network model which is expected to reach its full potential only on neuromorphic hardware. However, it is important to restate that the current research will focus on the implementation feasibility of a spiking neural network for guidance and control of a rapid drone. With this in mind, the following hypothesis can be formulated:

H.3: *Replacing a spiking neural network for guidance and control can achieve good learning accuracy but compared to state-of-the-art non-neuromorphic models, the accuracy and computational efficiency are worse when implemented on non-neuromophic hardware.*

7 Research Methodology

This chapter will detail the methodology used throughout the research that will lead to the research objective achievement. However, not every detail of the process will be presented as some design freedom will be left for adaptation during the research. Moreover, it is expected that the process might deviate a bit from the procedure if new results are found on the way. In any case, the research will follow the 3 stages mentioned in the previous chapter that led to the 3 main research questions. Thus, this chapter will stick to the same structure and will be divided into sections corresponding to the work required for each research phase. Firstly, the encoding procedure to transform the dataset input values to spikes will be detailed in section 7.1. Secondly, the phase of implementing and training a spiking neural network for rapid guidance and control will be analysed in section 7.2. Lastly, after a model was created capable of controlling and guiding a quadrotor, analysis and performance comparison will be required and the procedure will be detailed in section 7.3.

7.1 Encoding

As mentioned above, the practical implementation will debut with the encoding step required to transform the received input by the drone into spikes. This section will present the required methodology during the research to answer **RQ.1**. Moreover, looking at hypothesis **H.1**, 2 research points can be deducted. Firstly, a large and representative dataset has to be created to understand how efficient autoencoders can become for the transformation task. Secondly, using the created dataset, the autoencoder can be built and tweaked such that the required number of spiking neurons is heavily reduced. In the end, the quality of the encoding will be analysed with the sole goal of reducing the total number of spiking neurons.

There exists a multitude of neuromorphic datasets on the market ranging from drone flight in urban aerial for detection and localisation[164] to the flight of blimps with a radar-based altitude controller[55]. However, a critical aspect of dataset creation is ensuring completeness by incorporating all relevant input parameters essential for model development. While existing datasets may offer valuable insights into the research process, they often lack certain inputs necessary for capturing the intricacies of the system dynamics comprehensively. To solve this issue, the existing setup created by Ferede et al. can be used to generate a rich dataset with the required array of input parameters. This can be done with the reinforcement learning model[23] that can be trained to navigate complex circuits in a fast manner. Further, by simulating diverse flight scenarios and capturing the system states, a clear dataset is generated that encompasses a broad spectrum of operational conditions. Thus, it is expected that creating a tailored and comprehensive dataset for the current research problem, not only will enhance the fidelity of the guidance and control network models but will also facilitate robust training and evaluation processes.

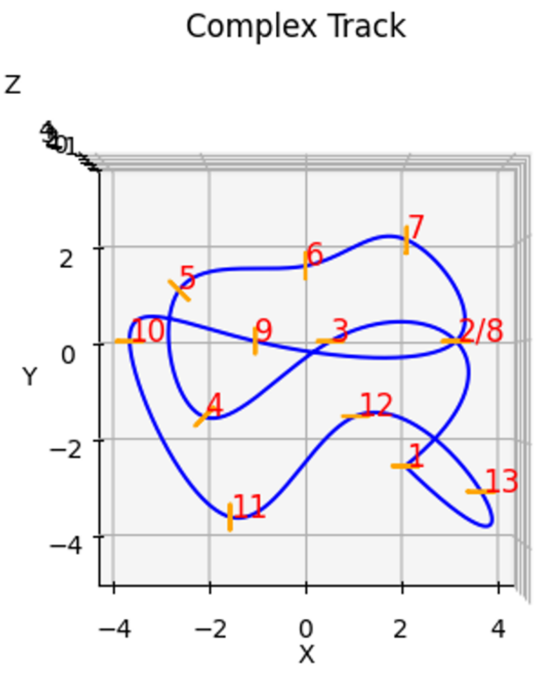


Figure 7.1: The track used to create the encoder dataset, seen from above

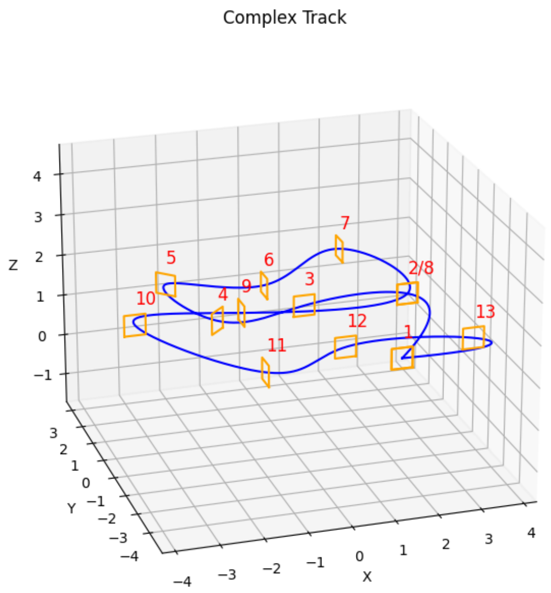


Figure 7.2: The track used to create the encoder dataset, seen from perspective

As mentioned above, to simulate datasets helpful for a better understanding on how to generate guidance and control with neuromorphic computing, the reinforcement learning approach created by Ferede et al.[23] will be used. A robust dataset is created by learning how to navigate complex circuits while time around the circuit is minimised. This ensures a faithful representation of real-world complexities and can be done by learning a great variation of manoeuvres. Fortunately, the reinforcement learning environment of Ferede et al.[23] can learn directly to mimic the target system dynamics even if the track is more complex. Thus the required work for this step will involve creating a complex map as seen in Figure 7.1-7.2 that the reinforcement learning model will learn to navigate it over iterative training cycles. The model will focus on improving the drone's control such that the time required for flying the created map is reduced. With the learned model, a complex dataset comprised of all the input parameters mentioned in section 6.2, can be generated by simply simulating new flights. To do this, the drone is initialised at various gates along the complex map and required to continue the flight from there. Even though generating data using the same map as the one used during training might lead to a slight bias, for the task of encoding, this was deemed acceptable as considerable time was saved.

In the end, the encoding dataset was created starting from various gates along the map and simulating the flight for 10 seconds. With a frequency rate of 0.05 seconds, this led to a batch size of 2000 timesteps, with one timestep having all the 28 required inputs as floating point values. Further, for a comprehensive and robust dataset, 2048 batches were created to train the autoencoder. To ensure the reliability and effectiveness of the trained models, a dataset-splitting strategy encompassing training, validation, and testing phases was adopted. During the training phase, the dataset is partitioned into a training subset (90% of the whole dataset) and a validation subset (10% of the whole dataset), enabling the model to learn from a diverse range of data while validating its performance on unseen samples. Additionally, a separate dataset is reserved for testing purposes, allowing for an unbiased assessment of the model's generalisation capabilities. Through this multi-stage dataset-splitting approach, the risk of overfitting is mitigated and the trained models are ensured to exhibit robust performance across a wide range of scenarios.

With the dataset acquired, the code to transform the 28 floating point inputs into spikes could be created. The structure designed for this task leverages the spiking library developed by Technological University of Delft within the PyTorch framework[1]. The architecture used is straightforward, consisting of an input layer with 28 floating point neurons fully connected to a layer of spiking neurons. The spiking layer is fully connected to an output layer aimed at reproducing the 28 initial floating point variables. The sole design philosophy centres on identifying the minimum number of spiking neurons in the middle layer necessary to maintain acceptable accuracy, facilitating further model refinement during the network training. For an accurate evaluation of the performance, the dataset was normalised between 0 and 1. With this in mind, the accuracy will be calculated as the mean squared error (MSE) between the actual dataset and the predicted values. An acceptable value was deemed to be around 0.05 for every input variable.

Regarding the structure of the network, the model adheres closely to the base model suggested in the repository as it is tailored to spiking specific requirements. Thus the autoencoder is created as a feed forward neural network that receives every time a batch of 2000 timesteps and learns how to encode the variable evolution in time. Moreover, the current structure employs the Arctan surrogate gradient descent function for optimisation and the Adam optimiser to find the global minimum. Additionally, it utilises the CuBa LIF neuron model and linear connections between layers which introduces recurrency in the network. As CuBa LIF neurons and a feed forward neural network are used, leaking of the voltage and current potentials is required which are initialised with sigmoid functions, constraining values between 0 and 1. Moreover, also the thresholds within the network are initialised but with ReLu functions that do not allow for negative values. Further, no biases are used inside the network as the spiking neural networks have a tendency to tweak especially those for encoding leading only to a local minimum convergence.

With an initial base model generated, the process passes further to the next phase where the focus lies on optimising the remaining parameters. On one hand, some parameters such as the potential values, thresholds and synapse weights remain adaptable throughout the learning process, allowing the model to dynamically adjust and optimise its performance. On the other hand, it is required to also fine-tune hyperparameters such as the number of epochs, number of spiking neurons, voltage and current potential leaks and thresholds initial values as well as the size of the dataset. The second step requires a methodical exploration of parameter space, aiming to strike a balance between model complexity and performance efficacy. By iteratively adjusting hyperparameters and fine-tuning the network architecture, the autoencoder seeks to improve its performance while enhancing the model's ability to generalise across diverse datasets. If this procedure does not succeed, encoding every parameter or batch of parameters with separate autoencoders can also be considered as a solution.

7.2 Learning Control Algorithm

The next step in the research procedure requires the practical implementation of the spiking neural network capable of learning the rapid guidance and control mapping for quadrotor flight. Thus this section will focus on answering to **RQ.2** as well as certifying the hypothesis **H.2**. Similar to the previous section, 2 research points can be deducted, one focused on the network itself and the other one on the dataset. The envisioned network entails a simple Spiking Neural Network (SNN) designed to learn through supervised learning, leveraging input-output relationships derived from a specialised dataset. The fundamental objective is for the network to assimilate input data of the drone's state and subsequently deduce through mapping, motor commands for each propeller (4 outputs). Secondly, this section also requires the creation of

[1]https://github.com/tudelft/spiking

a novel dataset which has to be carefully crafted to increase the performance of the network. For the dataset creation, 2 approaches were chosen that leverage the previous work of Ferede et al.[22, 23] as it will be presented below.

Firstly, the dataset creation through the supervised learning method will be detailed. This was inspired by the work of Ferede et al.[22] and it is created by generating optimal trajectories for a range of initial conditions. Thus a dataset of state-action pairs can be created in the form (x_i^*, u_i^*) based on an energy optimal control problem as follows: "Given a state space X and set of admissible controls U, the goal is to find a control trajectory $u : [0, T] \rightarrow U$ that steers the system from an initial state x_0 to some target state $S \subset X$ in time T while minimising some cost function"[22]. In mathematical formulation, the following cost function has to be minimised:

$$E(u,T) = \int_0^T \|u(t)\|^2 dt \tag{7.1}$$

subject to:

$$\dot{x} = f(x,u) \tag{7.2}$$
$$x(0) = x_0 \tag{7.3}$$
$$x(T) \in S \tag{7.4}$$

Further, according to Ferede et al.[22], this control problem can be transformed into a nonlinear programming (NLP) problem using Hermite Simpson transcription. The trajectories $x(t)$, $u(t)$ can be discretised into $N + 1$ points and thus the problem can be solved with a specialised NLP solver. In the end, optimal discretised trajectory $x_0^* \dots x_N^*$ and $u_0^* \dots u_N^*$ can be computed. The advantages of such a method include a better generalised dataset that is solely focused on achieving the optimal control rule without being constrained by a clear map that has to be followed. Moreover, this method does not need a model to be fully trained before creating a dataset which might reduce the computation time drastically.

By setting the values for x_0 and x_T, datasets can be generated with various constraints. Inspired by the work of Ferede et al.[22], one approach would be to learn to fly to a preset standing point from a widespread range of initial conditions as expressed by the following starting point intervals:

$$\begin{cases} x \in [-5,5] \;\; y \in [-5,5] \;\; z \in [-1,1] \\ v_x \in [-0.5,0.5] \;\; v_y \in [-0.5,0.5] \;\; v_z \in [-0.5,0.5] \\ \phi \in [-2\pi/9, 2\pi/9] \;\; \theta \in [-2\pi/9, 2\pi/9] \;\; \psi \in [-\pi,\pi] \\ p \in [-1,1] \;\; q \in [-1,1] \;\; r \in [-1,1] \\ \omega \in [\omega_{min}, \omega_{max}]^4 \end{cases} \tag{7.5}$$

With these as initial conditions, the final point was imposed to be set fully to 0. Or, in other words, this was defined by $\mathbf{x}(T)$, $\mathbf{v}(T)$, $\lambda(T)$, $\mathbf{\Omega}(T)$, $\dot{\mathbf{v}}(T)$, $\dot{\mathbf{\Omega}}(T)$, $\dot{\omega}(T) = 0$. With the initial and goal conditions established, a dataset can be created that will learn how to fly to a hover state. In order to fly the controller trained on the created dataset on a racing circuit, the simulator will have to update the goal conditions every time a gate is passed. It is expected that the current dataset, even though very generalisable and capable of performing even navigation, to fly slow as it is trained to converge towards a standing still ending point. In order to combat this problem and as the racing track is known, a new dataset will be generated with different initial and final conditions. The initial conditions are given by the following system of intervals:

$$\begin{cases} x \in [-5,-2] \;\; y \in [-1,1] \;\; z \in [-0.5,0.5] \\ v_x \in [-0.5,5] \;\; v_y \in [-3,3] \;\; v_z \in [-1,1] \\ \phi \in [-2\pi/9, 2\pi/9] \;\; \theta \in [-2\pi/9, 2\pi/9] \;\; \psi \in [-\pi/3, \pi/3] \\ p \in [-1,1] \;\; q \in [-1,1] \;\; r \in [-1,1] \\ \omega \in [\omega_{min}, \omega_{max}]^4 \end{cases} \tag{7.6}$$

The ending point is assumed to be a gate and keeping in mind that the flight has to be performed in a circular clockwise direction new constraints for $x(T)$ are set. Thus the target conditions are imposed to be 0 with the exception of the roll angle ψ which is set to $\pi/4$ and the y and x velocity ratio v_y/v_x which is set to $tan(\pi/4)$. A more specialised dataset can be generated in this manner and the corresponding trained controller can be simulated in a similar fashion as explained in the paragraph above.

The second dataset will be generated through a reinforcement learning approach as was done in the previous section. In a few words, using the approach of Ferede et al.[23], a network model can be trained through reinforcement learning to fly through a circuit. Using the model, a discretised dataset with the drone's states and motor commands can be created. However, as opposed to the previous section, the map has to be better analysed as learning to navigate a random complex map will not provide value for rapid guidance and control. To have a fair comparison with the models trained by Ferede et al.[23], a similar map will be utilised that can be seen in Figure 6.2.

The current reinforcement learning approach to generate a dataset was still chosen due to several factors. One of the most important ones refers to the fact that a dataset was already created and reproducing the method will save considerable time. Furthermore, the encoder is trained on datasets generated specifically through this approach, facilitating the integration of the autoencoder and spiking neural network in future networks. Secondly, while slightly improved performance may be anticipated when the model is tested on the same map it was trained on, generalising this approach to more complex maps poses a greater challenge. As mentioned previously, the focus of the current research is just to prove the feasibility of rapid guidance and control with neuromorphic computing. Thus the drawback mentioned previously is not crucial for the current research.

With a carefully crafted dataset established, a spiking neural network (SNN) model for rapid control and guidance of the drone can be built. While further adjustments and refinements will be implemented afterwards, an effort will be made to adhere closely to the network architecture established by Ferede et al.[22] which can be seen in Figure 7.3, primarily to facilitate a more accurate and insightful comparison. This architecture contains a three-layered network housing 120 neurons per layer. This structure is deemed crucial to respect due to the harsh hardware limitations caused by the drone's operational environment. Moreover, it is important to mention that the same network structure, as the autoencoder structure mentioned previously, will be used for creating the current spiking neural network. Using this approach, a versatile framework for data processing and feature extraction within the SNN model development process can be created.

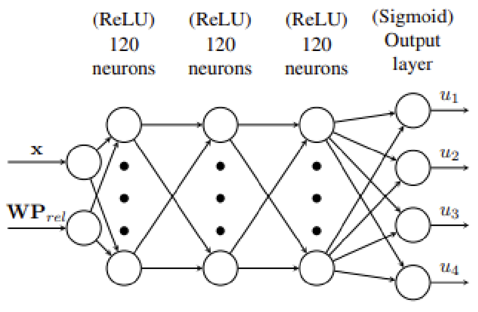


Figure 7.3: The neural network used for optimal control by Ferede et al.[22, 23]

However, the current approach detailed in this section summarises in a big picture what is expected to happen and how theoretically guidance and control can be done with neuromorphic computing. As this approach depends severely on the previous encoding phase results some mitigation solutions are going to be suggested in the current paragraph. Firstly, one possible problem might be that the dataset is too spread and contains too much detail to be encoded accurately. One possible solution would require the creation of a reduced dataset with less problematic parameters. For example, if disturbances mentioned in section 6.2 might be difficult to encode, discarding them will not affect the performance greatly.

7.3 Testing

The last step that has to be taken involves choosing the best-created model, testing it and comparing it to existing state-of-the-art models. With this in mind, the current section will focus on presenting the methodology to answer **RQ.3** as well as certifying the hypothesis **H.3**. Thus this segment can also be split into 2 subparts: one focused only on testing the created spiking neural network performance and another one focused on comparing the performance to the state-of-the-art artificial neural network approach. Moreover, it is important to remember that the hypothesis does not expect outstanding performance of the neuromorphic model. In the end, just flying the circuit can be considered satisfactory, with the performance being expected to be largely below the artificial neural network's.

To test the model thoroughly and compare it, a definitive course must be charted for the drone to be trained on and fly during testing. In the end, a strong consideration is given to using a rectangular trajectory similar to that employed in the previous work of Ferede et al.[22] that can be observed in Figure 6.2. This choice will facilitate direct performance comparison with non-neuromorphic models and will offer practical advantages, given the simplicity of mapping and the existing implementation in both the Paparazzi simulator and real-world drone setups. Moreover, simulation exercises will be conducted using Paparazzi, while real-world testing will be conducted at the Cyberzoo facility using the Bebop drone described in section 6.2 and the environment visualised in Figure 6.1. Key parameters that will be calculated for estimating the performance of the models were inspired by section 5.3 and include accuracy, performance loss and simulation duration, alongside more practical metrics such as training time and memory utilisation, coupled with energy efficiency assessments.

8 Research Planning

This chapter will offer an overview of the logistics required for the research project. Thus, a clear plan that should be followed for the remainder of the project will be presented. It is important to state that this planning is being done halfway through the research but the past tasks will still be considered for a better understanding and approximation of the timeline. The chapter will begin with a presentation of the main stages of the planning along with some milestones set during the research project. This will be followed by a Gantt chart to show the clear timeline of the project along with several practicalities during the development of the planning. For managing the work undertaken for this research, five major project phases have been established:

1. **Literature Study:** The first phase of the entire research project is the literature study which involves acquiring information such that the research objective can be generated and answered. The work usually includes an overview of the current state of the art, identifying a research gap and formulating clear research questions and a corresponding research methodology. This will result in a literature review paper which is the current document. Moreover, this phase will include a project plan report for a better organisation of the research project and will conclude with a mid-term review presentation to assess the feasibility of the project and propose a research alternative.

2. **Encoding:** This will be the second phase of the research project and will focus on the work required to answer the first research question. As encoding is considered to be the main problem of the research project that will dictate the continuation of the work expected during the mid-term review, it will be done in parallel with the literature study phase. This step will include the analysis and creation of the dataset and the implementation and tweaking of the autoencoder.

3. **Guidance and Control Network:** The first step after the mid-term review will focus on answering the second research question and thus creating a spiking neural network for the guidance and control of a quadrotor. This step is dependent on the encoding phase and thus can be analysed after a performing and efficient encoder is achieved. It will be focused on programming and including dataset generation and network optimisation.

4. **Testing:** Following the creation of a guidance and control network, its performance has to be quantified. Thus this phase will answer the third research question and will be concerned with the implementation of drone simulations and real flights. Moreover, the results obtained will be compared and analysed.

5. **Reporting:** The final phase will be to report all results and conclusions of the current research project through a final paper and a final presentation. It can be done in parallel to the previous 2 phases and it will contain numerous milestones which imposes several limitations to the timeline of this phase.

Throughout the previous enumeration, several milestones have been mentioned. For the current project, they are of utmost importance as they serve as clear time limits, aimed at evaluating the progress of the project as the research is conducted:

1. **Kick-off Meeting:** The kick-off meeting marks the initial significant milestone of this project and also represents the formal beginning. It aims to assess the research proposal after the research domain has been thoroughly studied. During the meeting, attention will be directed towards the methodology crafted for the research, as well as the research questions and objectives. Moreover, it will set some clear guidelines that have to be followed for the mid-term review.

2. **Literature Study Submission:** The subsequent significant milestone during the research is the literature study paper submission. This is scheduled close to the mid-term review as a thorough analysis of the current literature is required. Moreover, due to the infancy of the neuromorphic domain, it is necessary to start the encoding implementation for a more accurate research objective and questions for the remainder of the project.

3. **Mid-term Review Presentation:** The mid-term review is scheduled approximately three months before the green light review. During this session, a presentation will be delivered, offering an overview of the approach taken and presenting details on the methodological steps and current findings. Furthermore, detailed plans for the subsequent stages of the project will be outlined to ensure its successful progression. This milestone will serve as an intermediary GO/NO-GO meeting to clarify the status of the research project.

4. **Green Light Review:** The green light review aims to assess the current state of the research and determine if the progress made thus far is adequate to initiate the submission and review process. This milestone is the final GO/NO-GO meeting in which the assessment of the research work thus far will be analysed.

5. **Thesis Hand-in:** The ultimate version of the thesis, incorporating all received feedback throughout the project, will subsequently be submitted for review and defence and has to be at least 2 weeks before the final defence presentation.

6. **Defence:** The thesis defence represents the concluding phase of the process. During this stage, the research will be presented, and a group of examiners will have the opportunity to pose questions or concerns before conducting a final assessment of the entire project. This milestone has to be set at least 4 weeks after the green light review.

The summary of the phases and milestones along with their time prediction can be seen in Figure 8.1-8.4 where the Gantt chart presents the plan of the master thesis. This will follow a 5-day work week including the national Dutch holidays. Supplementary, 6 weeks of the holiday will be included (one during October, 3 during the Christmas period, another one during February and another week split into 2 periods in May-June). Additionally, as the strict ending deadline of the project was established to be around the middle of August 2024, a 4-week buffer time is introduced when possible delays or wrong estimations can be mitigated. Moreover, the time required for the tasks is also heavily overestimated to mitigate the problems that might arise during research. This overestimation and risk mitigation is required by the novelty of the neuromorphic domain which might lead to many problems during development.

A lot of freedom was introduced in the task succession due to risk mitigation issues. Thus it is important to see that some tasks that depend on one another, were made partially in parallel to include some buffer time. Similarly, almost every task has another different task in parallel which also allows for backup work in case the research gets stuck in a practical problem. Moreover, the learning curve was a considered factor when determining the work effort during the project.

For example, during the first half of the research, the first tweaking of hyperparameters was assigned 9 weeks while, later, performing the same task was only given 3 weeks. This is because the first time doing the block requires also understanding the task and creating the code which is not necessary when reiterated.

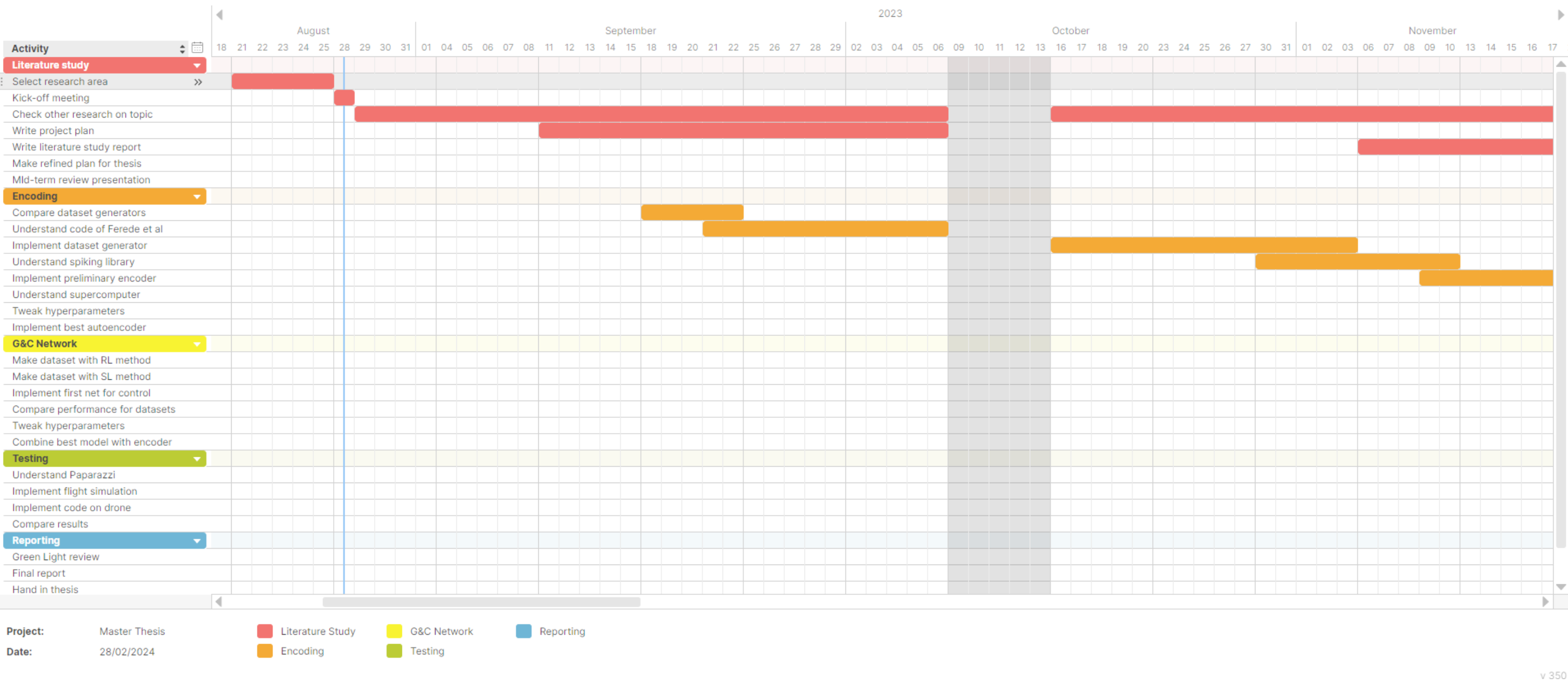


Figure 8.1: Gantt chart I

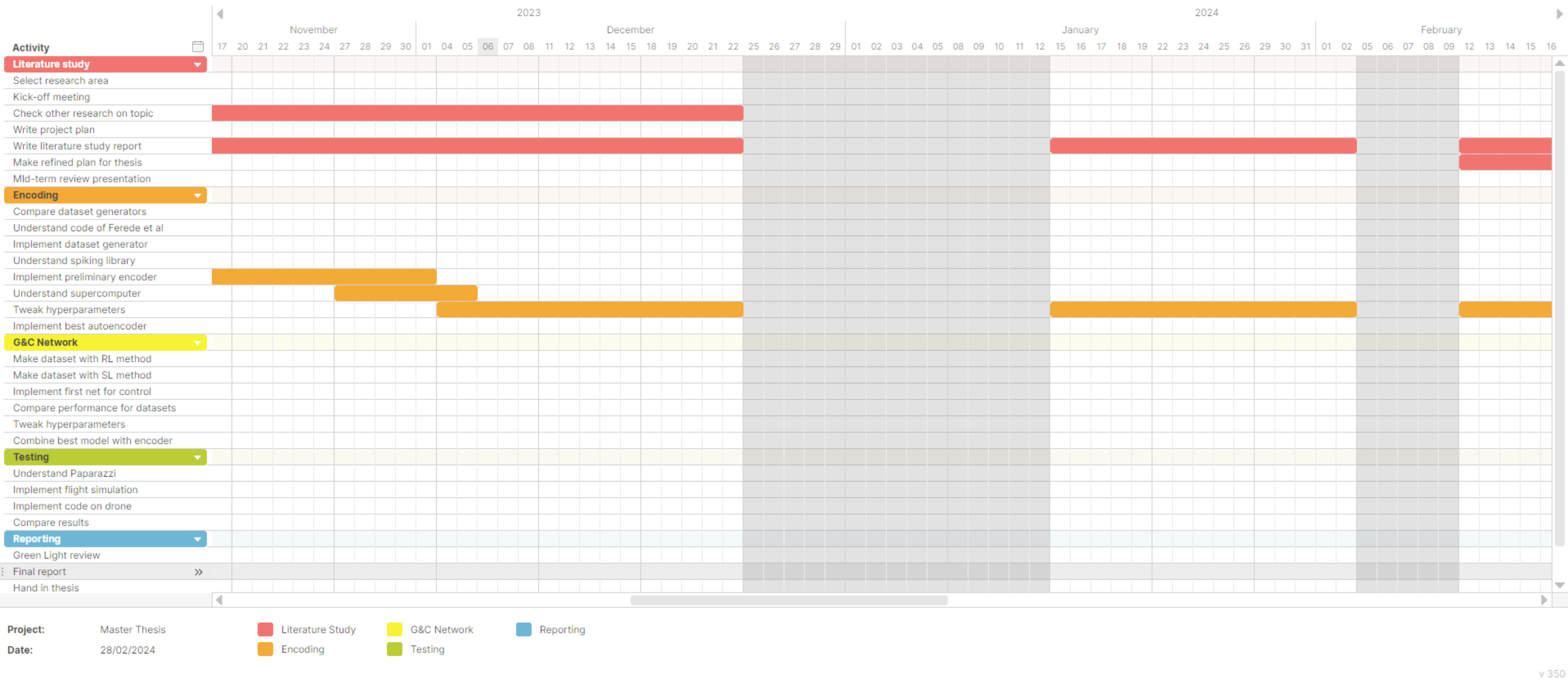


Figure 8.2: Gantt chart II

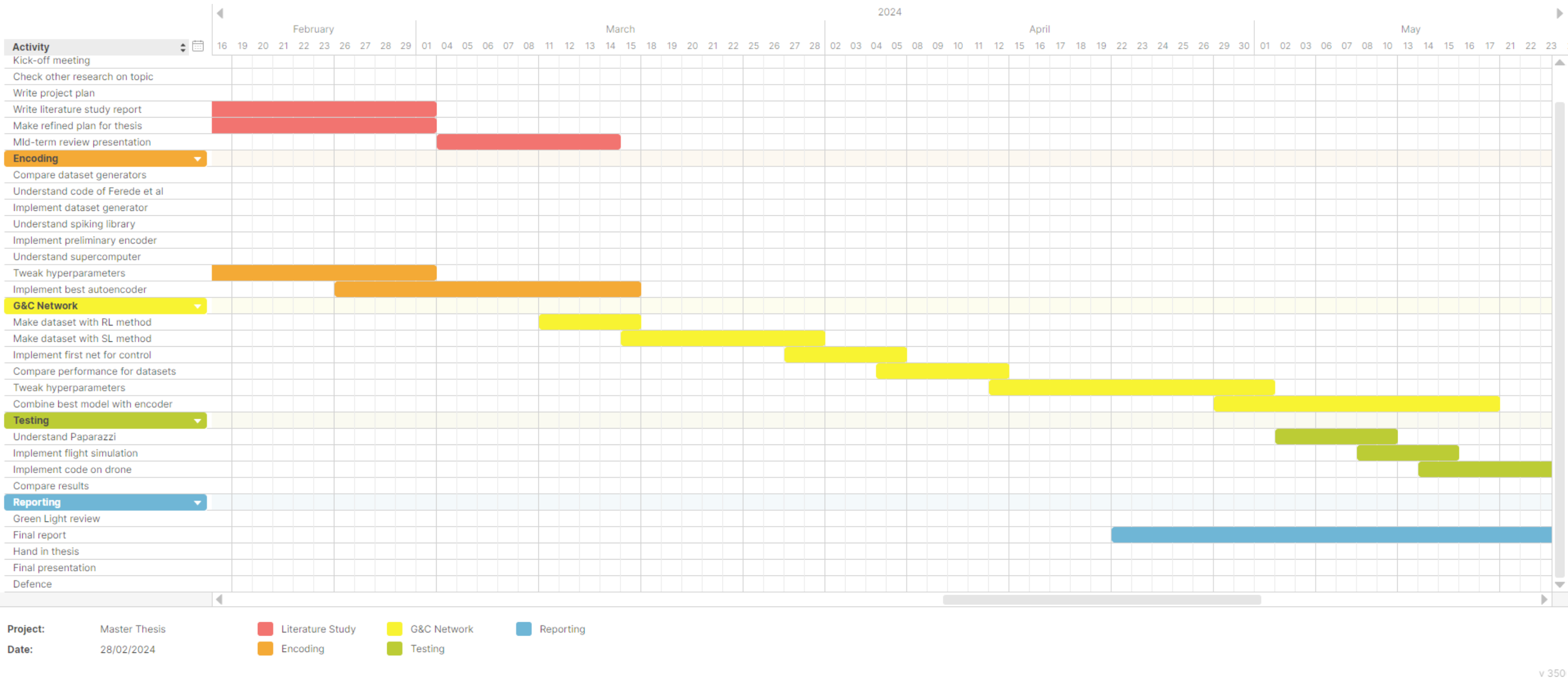


Figure 8.3: Gantt chart III

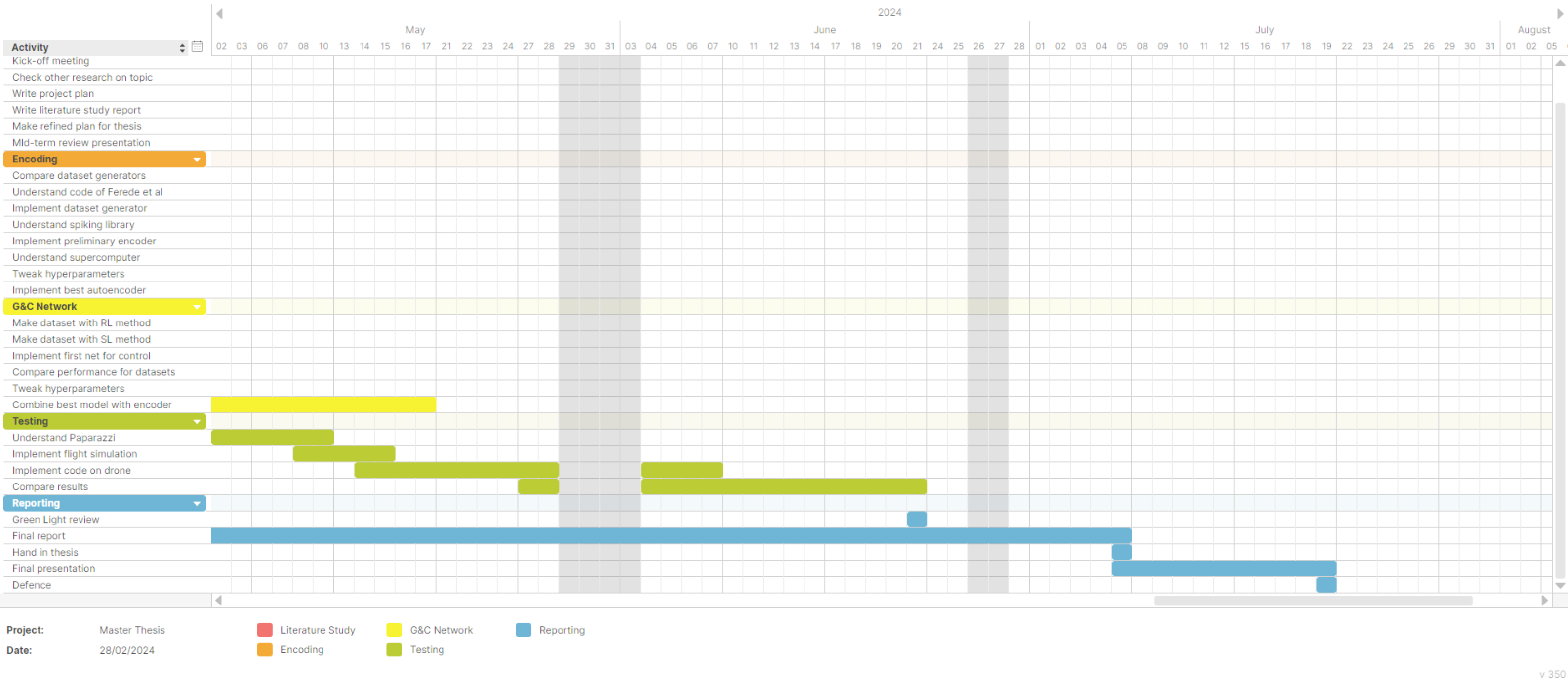


Figure 8.4: Gantt chart IV

9 Conclusion

The current literature study aimed to analyse the implementation of guidance and control for a racing drone using the novel domain of neuromorphic computing. After a comprehensive exploration of applications leveraging neuromorphic computing, a research gap was identified in the realisation of a fully neuromorphic racing drone, particularly in terms of rapid guidance and control methodologies. Despite an extensive review of the literature, no existing studies were found that addressed the neuromorphic implementation of rapid guidance and control which offers the current research a pioneering role at the confluence of these two domains. However, given the novelty, prudent risk mitigation strategies are imperative. Constrained by a relatively short timeline, the research project will lay the groundwork for future endeavours, focusing primarily on verifying the feasibility of such an implementation. Accordingly, the current research will focus on elucidating how a spiking neural network can be trained to generate propeller motor commands based on the drone's state to fly rapidly a racing circuit. To address this research problem, the project was split into three distinct phases.

The initial phase entails the encoding of floating-point inputs into spikes. Out of different approaches, this task was entrusted to autoencoders due to their capacity for information compression and temporal correlation identification. The spiking neural network structure comprises input and output layers for floating-point variables, interconnected by a layer of spiking neurons. This layer employs leaky integrate-and-fire dynamics, connected by simple, linear synaptic connections. The time dependency specific of spiking neurons will impose reccurency to the feed-forward neural network architecture. Supervised learning, facilitated by surrogate gradient descent methods for back-propagation, is adopted, necessitating a meticulously generated dataset derived from 2 frameworks proposed by Ferede et al. One will focus on generating the dataset with a neural network model trained using a reinforcement learning approach[23] while the second will focus on solving an energy optimal problem and thus generate multiple flying trajectories. The anticipated outcome involves a significant reduction in spiking neuron count, leading to a model capable of deployment on diminutive drones.

Subsequently, using insights drawn from the autoencoder framework, a spiking neural network for learning fast drone manoeuvring can be constructed. However, this phase will differ by the number of neural network layers and by the respective number of spiking neurons. Moreover, the dataset will be regenerated to accommodate a facile integration of the neural network. Based on the performance achieved with the current structure, several mitigation techniques have been proposed. Lastly, the developed neuromorphic guidance and control undergoes evaluation using state-of-the-art artificial neural network models of Ferede et al.[22, 23], encompassing both simulation and real-world drone flights. While achieving similar performance to conventional artificial neural network models is not anticipated, this stage is crucial for determining the efficacy of the proposed approach.

Despite the steps made in developing the groundwork for a fully neuromorphic racing drone, several research gaps persist, leading to some recommendations for future investigations. Among these, there is the imperative need to explore alternative approaches, such as a fully neuromorphic reinforcement learning algorithm, for enhanced performance of a racing drone. Additionally, a correct comparison to conventional artificial neural networks necessitates the integration of event-based cameras and specialised neuromorphic hardware to facilitate a comprehensive evaluation. Furthermore, a deeper understanding of unsupervised learning and temporal encoding mechanisms prevalent in biological systems requires the need for continued exploration. Likewise, advancements in neuromorphic hardware remain pivotal for maximising the potential of spiking neural networks, that inquire sustained effort into streamlined implementation methodologies.

References

[1] Daniel Smihula. *Long Waves of Technological Innovations. Studia Politica Slovaca* 2 (July 2013), pp. 50–68.

[2] Bo Zhang, Huiping Shi, and Hongtao Wang. *Machine Learning and AI in Cancer Prognosis, Prediction, and Treatment Selection: A Critical Approach. Journal of Multidisciplinary Healthcare* 16 (June 2023), pp. 1779–1791. DOI: 10.2147/JMDH.S410301.

[3] Awais Bajwa, Neelam Nosheen, Khalid Iqbal Talpur, and Sheeraz Akram. *A Prospective Study on Diabetic Retinopathy Detection Based on Modify Convolutional Neural Network Using Fundus Images at Sindh Institute of Ophthalmology & Visual Sciences. Diagnostics* 13.3 (Jan. 2023), p. 393. DOI: 10.3390/diagnostics13030393.

[4] Margarita Martínez-Díaz and Francesc Soriguera. *Autonomous vehicles: theoretical and practical challenges. Transportation Research Procedia* 33 (2018). XIII Conference on Transport Engineering, CIT2018, pp. 275–282. DOI: https://doi.org/10.1016/j.trpro.2018.10.103.

[5] Anton Chernyavskiy, Dmitry Ilvovsky, and Preslav Nakov. *Transformers: "The End of History" for NLP?* Apr. 2021. DOI: doi.org/10.48550/arXiv.2105.00813.

[6] Marta Montenegro-Rueda, José Fernández-Cerero, José María Fernández-Batanero, and Eloy López-Meneses. *Impact of the Implementation of ChatGPT in Education: A Systematic Review. Computers* 12.8 (2023). DOI: 10.3390/computers12080153.

[7] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank. *A Survey of Neuromorphic Computing and Neural Networks in Hardware.* May 2017. DOI: doi.org/10.48550/arXiv.1705.06963.

[8] Catherine D. Schuman, Shruti R. Kulkarni, Maryam Parsa, J. Parker Mitchell, Prasanna Date, and Bill Kay. *Opportunities for neuromorphic computing algorithms and applications. Nature Computational Science* 2 (Jan. 2022), pp. 10–19. DOI: doi.org/10.1038/s43588-021-00184-y.

[9] Abderahman Rejeb, Alireza Abdollahi, Karim Rejeb, and Horst Treiblmaier. *Drones in agriculture: A review and bibliometric analysis. Computers and Electronics in Agriculture* 198 (July 2022), pp. 107–124. DOI: https://doi.org/10.1016/j.compag.2022.107017.

[10] Xueping Li, Jose Tupayachi, Aliza Sharmin, and Madelaine Martinez Ferguson. *Drone-Aided Delivery Methods, Challenge, and the Future: A Methodological Review. Drones* 7.3 (Mar. 2023). DOI: 10.3390/drones7030191.

[11] Elia Kaufmann, Leonard Bauersfeld, Antonio Loquercio, Matthias Muller, Vladen Koltun, and Davide Scaramuzza. *Champion-level drone racing using deep reinforcement learning. Nature* 620 (Aug. 2023), pp. 982–987. DOI: doi.org/10.1038/s41586-023-06419-4.

[12] Zainab Zaheer, Atiya Usmani, Ekram Khan, and Mohammed A. Qadeer. *Aerial surveillance system using UAV. 2016 Thirteenth International Conference on Wireless and Optical Communications Networks (WOCN).* IEEE, July 2016, pp. 1–7. DOI: 10.1109/WOCN.2016.7759885.

[13] S. Sanz-Martos, M. D. López-Franco, C. Álvarez-García, N. Granero-Moya, J. M. López-Hens, S. Cámara-Anguita, P. L. Pancorbo-Hidalgo, and I. M. Comino-Sanz. *Drone applications for emergency and urgent care: A systematic review. Prehospital and Disaster Medicine* 37.4 (Aug. 2022), pp. 502–508. DOI: 10.1017/s1049023x22000887.

[14] A. Izyumov, E. Vasilyeva, O. Ostapovich, and E. Alentsov. *Energy Efficiency Analysis of modern delivery drones. E3S Web of Conferences* 279 (2021). DOI: 10.1051/e3sconf/202127901028.

[15] Bian Sizhen, Schulthess Lukas, Rutishauser Georg, Di Mauro Alfio, Benini Luca, and Magno Michele. *ColibriUAV: An Ultra-Fast, Energy-Efficient Neuromorphic Edge Processing UAV-Platform with Event-Based and Frame-Based Cameras.* May 2023. DOI: doi.org/10.48550/arXiv.2305.18371.

[16] Wolfgang Maass and Christopher M. Bishop. *Pulsed Neural Networks.* Boston, Massachusetts: MIT Press, 1999. ISBN: 9780262278768.

[17] Timo Wunderlich, Akos F. Kungl, Eric Müller, Andreas Hartel, Yannik Stradmann, Syed Ahmed Aamir, Andreas Grübl, Arthur Heimbrecht, Korbinian Schreiber, David Stöckel, Christian Pehle, Sebastian Billaudelle, Gerd Kiene, Christian Mauch, Johannes Schemmel, Karlheinz Meier, and Mihai A. Petrovici. *Demonstrating Advantages of Neuromorphic Computation: A Pilot Study. Frontiers in Neuroscience* 13 (Mar. 2019). DOI: 10.3389/fnins.2019.00260.

[18] Kashu Yamazaki, Viet-Khoa Vo-Ho, Darshan Bulsara, and Ngan Le. *Spiking Neural Networks and Their Applications: A Review. Brain Sciences* 12.7 (June 2022). DOI: 10.3390/brainsci12070863.

[19] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B. Kuang, Rajit Manohar, William P. Risk, Bryan Jackson, and Dharmendra S. Modha. *TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (2015), pp. 1537–1557. DOI: doi.org/10.1109/TCAD.2015.2474396.

[20] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Prasad Joshi, Andrew Lines, Andreas Wild, Hong Wang, and Deepak Mathaikutty. *Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. IEEE Micro* 99 (Jan. 2018), pp. 1–10. DOI: doi.org/10.1109/MM.2018.112130359.

[21] Lin Wang, I.S. Mohammad Mostafavi, Yo-Sung Ho, and Kuk-Jin Yoon. *Event-Based High Dynamic Range Image and Very High Frame Rate Video Generation Using Conditional Generative Adversarial Networks. 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* IEEE, June 2019, pp. 10073–10082. DOI: 10.1109/CVPR.2019.01032.

[22] Robin Ferede, Guido C. H. E. de Croon, Christophe De Wagter, and Dario Izzo. *End-to-end Neural Network Based Quadcopter control.* June 2023. DOI: 10.48550/arXiv.2304.13460.

[23] Robin Ferede, Guido C. H. E. de Croon, Christophe De Wagter, and Dario Izzo. *End-to-end Reinforcement Learning for Time-Optimal Quadcopter Flight.* Nov. 2023. DOI: doi.org/10.48550/arXiv.2311.16948.

[24] A. M. Turing. *On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society* 42.1 (Feb. 1937), pp. 230–265. DOI: doi:10.1112/plms/s2-42.1.230.

[25] John von Neumann. *First Draft of a Report on the EDVAC.* June 1945. URL: http://abelgo.cn/cs101/papers/Neumann.pdf.

[26] Alan Turing. *Intelligent Machinery (1948). The Essential Turing.* Oxford University Press, Sept. 2004. DOI: 10.1093/oso/9780198250791.003.0016.

[27] Donald O. Hebb. *The Organization of Behavior: A Neuropsychological Theory.* Psychology Press, 1949. ISBN: 978-0805843002.

[28] F. Rosenblatt. *The perceptron: a probabilistic model for information storage and organization in the brain. Psychological Review* 65.6 (1958), pp. 386–408. DOI: doi.org/10.1037/h0042519.

[29] D. Rumelhart, G. Hinton, and R. Williams. *Learning representations by back-propagating errors. Nature* 323 (Oct. 1986), pp. 533–536. DOI: doi.org/10.1038/323533a0.

[30] J J Hopfield. *Neural networks and physical systems with emergent collective computational abilities. Proceedings of the National Academy of Sciences* 79.8 (1982), pp. 2554–2558. DOI: 10.1073/pnas.79.8.2554.

[31] B Widrow. *The original adaptive neural net broom-balancer. Proceedings of the IEEE International Symposium on Circuits and Systems.* Vol. 2. May 1987, pp. 351–357.

[32] L. Chua. *Memristor—the missing circuit element. IEEE Transactions on Circuit Theory* 18.5 (Sept. 1971), pp. 507–519. DOI: doi.org/10.1109/TCT.1971.1083337.

[33] E. L. Bienenstock, L. N. Cooper, and P. W. Munro. *Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. The Journal of neuroscience : the official journal of the Society for Neuroscience* 2.1 (Jan. 1982), pp. 32–48. DOI: doi.org/10.1523/JNEUROSCI.02-01-00032.

[34] Carver Mead and Lynn Conway. *Introduction to VLSI systems. Reading, MA, Addison-Wesley Publishing Co., 1980. 426 p.* -1 (Jan. 1980).

[35] E. Farquhar, C. Gordon, and P. Hasler. *A field programmable neural array. 2006 IEEE International Symposium on Circuits and Systems.* 2006, 4 pp.–4117. DOI: doi.org/10.1109/ISCAS.2006.1693534.

[36] Greg S. Snider. *Spike-timing-dependent learning in memristive nanodevices. 2008 IEEE International Symposium on Nanoscale Architectures.* 2008, pp. 85–92. DOI: doi.org/10.1109/NANOARCH.2008.4585796.

[37] Chi-Sang Poon and Kuan Zhou. *Neuromorphic Silicon Neurons and Large-Scale Neural Networks: Challenges and Opportunities. Frontiers in Neuroscience* 5 (2011). DOI: doi.org/10.3389/fnins.2011.00108.

[38] Steve Furber. *Large-scale neuromorphic computing systems. Journal of Neural Engineering* 13.5 (Aug. 2016), pp. 1–15. DOI: doi.org/10.1088/1741-2560/13/5/051001.

[39] Kai Malcom and Josue Casco-Rodriguez. *A Comprehensive Review of Spiking Neural Networks: Interpretation, Optimization, Efficiency, and Best Practices.* Mar. 2023. DOI: doi.org/10.48550/arXiv.2303.10780.

[40] Zheng Shengjie, Qian Lang, Li Pingsheng, He Chenggang, Qin Xiaoqin, and Li Xiaojian. *An Introductory Review of Spiking Neural Network and Artificial Neural Network: From Biological Intelligence to Artificial Intelligence.* Apr. 2022. DOI: doi.org/10.48550/arXiv.2204.07519.

[41] Xiangwen Wang, Xianghong Lin, and Xiaochao Dang. *Supervised learning in spiking neural networks: A review of algorithms and evaluations. Neural Networks* 125 (May 2020), pp. 258–280. DOI: doi.org/10.1016/j.neunet.2020.02.011.

[42] Kai Malcom and Josue Casco-Rodriguez. *A Comprehensive Review of Spiking Neural Networks: Interpretation, Optimization, Efficiency, and Best Practices* (Mar. 2023). DOI: doi.org/10.48550/arXiv.2303.10780.

[43] Eustace Painkras, Luis A. Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R. Lester, Andrew D. Brown, and Steve B. Furber. *SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation. IEEE Journal of Solid-State Circuits* 48.8 (2013), pp. 1943–1953. DOI: doi.org/10.1109/JSSC.2013.2259038.

[44] Junyi Wang. *A Review of Spiking Neural Networks. SHS Web of Conferences* 144 (Aug. 2022). DOI: doi.org/10.1051/shsconf/202214403004.

[45] Christian Pehle, Sebastian Billaudelle, Benjamin Cramer, Jakob Kaiser, Korbinian Schreiber, Yannik Stradmann, Johannes Weis, Aron Leibfried, Eric Müller, and Johannes Schemmel. *The BrainScaleS-2 Accelerated Neuromorphic System With Hybrid Plasticity. Frontiers in Neuroscience* 16 (2022). DOI: doi.org/10.3389/fnins.2022.795876.

[46] Lei Deng, Guanrui Wang, Guoqi Li, Shuangchen Li, Ling Liang, Maohua Zhu, Yujie Wu, Zheyu Yang, Zhe Zou, Jing Pei, Zhenzhi Wu, Xing Hu, Yufei Ding, Wei He, Yuan Xie, and Luping Shi. *Tianjic: A Unified and Scalable Chip Bridging Spike-Based and Continuous Neural Computation. IEEE Journal of Solid-State Circuits* 55.8 (2020), pp. 2228–2246. DOI: 10.1109/JSSC.2020.2970709.

[47] Saber Moradi, Ning Qiao, Fabio Stefanini, and Giacomo Indiveri. *A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs). IEEE Transactions on Biomedical Circuits and Systems* 12.1 (2018), pp. 106–122. DOI: 10.1109/TBCAS.2017.2759700.

[48] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R. Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V. Arthur, Paul A. Merolla, and Kwabena Boahen. *Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations. Proceedings of the IEEE* 102.5 (2014), pp. 699–716. DOI: 10.1109/JPROC.2014.2313565.

[49] James B. Aimone. *A Roadmap for Reaching the Potential of Brain-Derived Computing. Advanced Intelligent Systems* 3.1 (Jan. 2021). DOI: doi.org/10.1002/aisy.202000191.

[50] Mingyue Zeng, Yongli He, Chenxi Zhang, and Qing Wan. *Neuromorphic Devices for Bionic Sensing and Perception. Frontiers in Neuroscience* 15 (June 2021). DOI: doi.org/10.3389/fnins.2021.690950.

[51] T. Sarkar, K. Lieberth, A. Pavlou, T. Frank, V. Mailaender, I. McCulloch, P. W. W. Blom, F. Toricelli, and P. Gkoupidenis. *An organic artificial spiking neuron for in situ neuromorphic sensing and biointerfacing. Nature Electronics* 5 (Nov. 2022), pp. 774–783. DOI: doi.org/10.1038/s41928-022-00859-y.

[52] Guillermo Gallego, Tobi Delbrück, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew J. Davison, Jörg Conradt, Kostas Daniilidis, and Davide Scaramuzza. *Event-Based Vision: A Survey. IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.1 (Jan. 2022), pp. 154–180. DOI: doi.org/10.1109/TPAMI.2020.3008413.

[53] Justas Furmonas, John Liobe, and Vaidotas Barzdenas. *Analytical Review of Event-Based Camera Depth Estimation Methods and Systems. Sensors* 22.3 (Feb. 2022). DOI: doi.org/10.3390/s22031201.

[54] Patricia P. Parlevliet, Andrey Kanaev, Chou P. Hung, Andreas Schweiger, Frederick D. Gregory, Ryad Benosman, Guido C. H. E. de Croon, Yoram Gutfreund, Chung-Chuan Lo, and Cynthia F. Moss. *Autonomous Flying With Neuromorphic Sensing. Frontiers in Neuroscience* 15 (May 2021). DOI: doi.org/10.3389/fnins.2021.672161.

[55] Marina González-Álvarez, Julien Dupeyroux, Federico Corradi, and Guido C.H.E. De Croon. *Evolved neuromorphic radar-based altitude controller for an autonomous open-source blimp. 2022 International Conference on Robotics and Automation (ICRA).* July 2022, pp. 85–90. DOI: doi.org/10.1109/ICRA46639.2022.9812149.

[56] Kevin Y. Ma, Pakpong Chirarattananon, Sawyer B. Fuller, and Robert J. Wood. *Controlled Flight of a Biologically Inspired, Insect-Scale Robot. Science* 340.6132 (May 2013), pp. 603–607. DOI: doi.org/10.1126/science.1231806.

[57] Taylor S. Clawson, Silvia Ferrari, Sawyer B. Fuller, and Robert J. Wood. *Spiking neural network (SNN) control of a flapping insect-scale robot. 2016 IEEE 55th Conference on Decision and Control (CDC).* Dec. 2016, pp. 3381–3388. DOI: doi.org/10.1109/CDC.2016.7798778.

[58] Matěj Karásek, Florian T. Muijres, Christophe De Wagter, Bart D. W. Remes, and Guido C. H. E. de Croon. *A tailless aerial robotic flapper reveals that flies use torque coupling in rapid banked turns. Science* 361.6407 (Sept. 2018), pp. 1089–1094. DOI: doi.org/10.1126/science.aat0350.

[59] K. N. McGuire, C. De Wagter, K. Tuyls, H. J. Kappen, and G. C. H. E. de Croon. *Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment. Science Robotics* 4.35 (Oct. 2019), pp. 1–14. DOI: doi.org/10.1126/scirobotics.aaw9710.

[60] Andreas Pflaum and Franz Eutermoser. *Quadrocopter Stabilization using Neuromorphic Embedded Dynamic Vision Sensors (eDVS).* Research Practice. Linprunstr. 06, 80335, Munchen: Technische Universit¨at M¨unchen, Nov. 2012.

[61] A. Mitrokhin, P. Sutor, C. Fermüller, and Y. Aloimonos. *Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception. Science Robotics* 4.30 (May 2019), pp. 1–19. DOI: doi.org/10.1126/scirobotics.aaw6736.

[62] Mohammad Omar Khyam, David Alexandre, Ananya Bhardwaj, Ruihao Wang, and Rolf Müller. *Neuromorphic Computing for Autonomous Mobility in Natural Environments. Proceedings of the 7th Annual Neuro-Inspired Computational Elements Workshop.* New York, NY, USA: Association for Computing Machinery, Mar. 2019. DOI: doi.org/10.1145/3320288.3320297.

[63] Stein Stroobants, Julien Dupeyroux, and Guido C.H.E. de Croon. *Neuromorphic computing for attitude estimation onboard quadrotors.* Apr. 2023. DOI: doi.org/10.48550/arXiv.2304.08802.

[64] Rasmus Stagsted, Antonio Vitale, Jonas Binz, Alpha Renner, Leon Bonde Larsen, and Yulia Sandamirskaya. *Towards neuromorphic control: A spiking neural network based PID controller for UAV. Robotics: Science and Systems.* July 2020. DOI: doi.org/10.15607/rss.2020.xvi.074.

[65] Stein Stroobants, Christophe de Wagter, and Guido C.H.E. de Croon. *Neuromorphic Control using Input-Weighted Threshold Adaptation.* Apr. 2023. DOI: doi.org/10.48550/arXiv.2304.08778.

[66] Stein Stroobants, Julien Dupeyroux, and Guido C.H.E. de Croon. *Design and implementation of a parsimonious neuromorphic PID for onboard altitude control for MAVs using neuromorphic processors.* Sept. 2021. DOI: doi.org/10.48550/arXiv.2109.10199.

[67] Rasmus Karnøe Stagsted, Antonio Vitale, Alpha Renner, Leon Bonde Larsen, Anders Lyhne Christensen, and Yulia Sandamirskaya. *Event-based PID controller fully realized in neuromorphic hardware: a one DoF study. 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* Oct. 2020, pp. 10939–10944. DOI: doi.org/10.1109/IROS45743.2020.9340861.

[68] Junjie Jiang, Delei Kong, Kuanxu Hou, Xinjie Huang, Hao Zhuang, and Zheng Fang. *Neuro-Planner: A 3D Visual Navigation Method for MAV With Depth Camera Based on Neuromorphic Reinforcement Learning. IEEE Transactions on Vehicular Technology* 72.10 (Sept. 2023), pp. 12697–12712. DOI: doi.org/10.1109/TVT.2023.3278097.

[69] Tim Landgraf, Benjamin Wild, Tobias Ludwig, Philipp Nowak, Lovisa Helgadottir, Benjamin Daumenlang, Philipp Breinlinger, Martin Nawrot, and Raúl Rojas. *NeuroCopter: Neuromorphic Computation of 6D Ego-Motion of a Quadcopter. Biomimetic and Biohybrid Systems*. Ed. by Nathan F. Lepora, Anna Mura, Holger G. Krapp, Paul F. M. J. Verschure, and Tony J. Prescott. Berlin, Heidelberg: Springer Berlin Heidelberg, Nov. 2013, pp. 143–153.

[70] Simon D. Levy. *Robustness Through Simplicity: A Minimalist Gateway to Neurorobotic Flight. Frontiers in Neurorobotics* 14 (Mar. 2020). DOI: doi.org/10.3389/fnbot.2020.00016.

[71] Brent Komer, Pawel Jaworski, Steven Harbour, Chris Eliasmith, and Travis DeWolf. *BatSLAM: Neuromorphic Spatial Reasoning in 3D Environments. 2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*. Sept. 2022, pp. 1–8. DOI: 10.1109/DASC55683.2022.9925773.

[72] F. Paredes-Vallés, Kirk Y. W. Scheper, and Guido C. H. E. de Croon. *Unsupervised Learning of a Hierarchical Spiking Neural Network for Optical Flow Estimation: From Events to Global Motion Perception. IEEE Transactions on Pattern Analysis and Machine Intelligence* 42 (July 2018), pp. 2051–2064.

[73] Julien Dupeyroux, Federico Paredes-Vallés, Jesse Hagenaars, and Guido C.H.E. de Croon. *Neuromorphic control for optic-flow-based landings of MAVs using the Loihi processor*. Nov. 2020. DOI: doi.org/10.48550/arXiv.2011.00534.

[74] Jesse Hagenaars, Federico Paredes-Vallés, Sander Bohte, and Guido Croon. *Evolved Neuromorphic Control for High Speed Divergence-Based Landings of MAVs. IEEE Robotics and Automation Letters* PP (July 2020), pp. 1–8. DOI: doi.org/10.1109/LRA.2020.3012129.

[75] Federico Paredes-Vallés, Jesse Hagenaars, Julien Dupeyroux, Stein Stroobants, Yingfu Xu, and Guido C.H.E. de Croon. *Fully neuromorphic vision and control for autonomous drone flight*. Mar. 2023. DOI: doi.org/10.48550/arXiv.2303.08778.

[76] LIM H TAEK, Jia Hoong Ong, Karan Singh, and Jung-Seok Choi. *Implementing Event-Driven Vision and Control on a Neuromorphic Chip*. May 2023. DOI: doi.org/10.31219/osf.io/akpxt.

[77] Antonio Vitale, Alpha Renner, Celine Nauer, Davide Scaramuzza, and Yulia Sandamirskaya. *Event-driven Vision and Control for UAVs on a Neuromorphic Chip. 2021 IEEE International Conference on Robotics and Automation (ICRA)* (May 2021), pp. 103–109. DOI: doi.org/10.1109/ICRA48506.2021.9560881.

[78] Sourav Sanyal, Rohan Kumar Manna, and Kaushik Roy. *EV-Planner: Energy-Efficient Robot Navigation via Event-Based Physics-Guided Neuromorphic Planner*. July 2023. DOI: doi.org/10.48550/arXiv.2307.11349.

[79] Luca Zanatta, Alfio Di Mauro, Francesco Barchi, Andrea Bartolini, Luca Benini, and Andrea Acquaviva. *Directly-trained Spiking Neural Networks for Deep Reinforcement Learning: Energy efficient implementation of event-based obstacle avoidance on a neuromorphic accelerator. Neurocomputing* 562 (Dec. 2023). DOI: doi.org/10.1016/j.neucom.2023.126885.

[80] Llewyn Salt, David Howard, Giacomo Indiveri, and Yulia Sandamirskaya. *Differential Evolution and Bayesian Optimisation for Hyper-Parameter Selection in Mixed-Signal Neuromorphic Circuits Applied to UAV Obstacle Avoidance.* Apr. 2017. DOI: doi.org/10.48550/arXiv.1704.04853.

[81] Llewyn Salt, Giacomo Indiveri, and Yulia Sandamirskaya. *Obstacle avoidance with LGMD neuron: Towards a neuromorphic UAV implementation. IEEE International Symposium on Circuits and System.* May 2017, pp. 1–6. DOI: doi.org/10.1109/ISCAS.2017.8050976.

[82] Sizhen Bian, Lukas Schulthess, Georg Rutishauser, Alfio Di Mauro, Luca Benini, and Michele Magno. *ColibriUAV: An Ultra-Fast, Energy-Efficient Neuromorphic Edge Processing UAV-Platform with Event-Based and Frame-Based Cameras.* June 2023, pp. 287–292. DOI: doi.org/10.1109/IWASI58316.2023.10164354.

[83] Catherine D. Schuman, James S. Plank, Grant Bruer, and Jeremy Anantharaj. *Non-Traditional Input Encoding Schemes for Spiking Neuromorphic Systems. 2019 International Joint Conference on Neural Networks (IJCNN).* Sept. 2019, pp. 1–10. DOI: doi.org/10.1109/IJCNN.2019.8852139.

[84] Jianxiang He, Yanzi Li, Yingtian Liu, Jiyang Chen, Chaoqun Wang, Rui Song, and Yibin Li. *The development of Spiking Neural Network: A Review. 2022 IEEE International Conference on Robotics and Biomimetics (ROBIO).* Dec. 2022, pp. 385–390. DOI: doi.org/10.1109/ROBIO55434.2022.10012028.

[85] Boudjelal Meftah, Olivier Lezoray, Soni Chaturvedi, Aleefia Khurshid, and Abdelkader Benyettou. *Image Processing with Spiking Neuron Networks.* Vol. 427. Jan. 2013, pp. 525–544. ISBN: 9783642296949. DOI: doi.org/10.1007/978-3-642-29694-9_20.

[86] E.D. Adrian and Yngve Zotterman. *The impulses produced by sensory nerve-endings. The Journal of Physiology* 61.2 (Apr. 1926), pp. 151–171. DOI: doi.org/10.1113/jphysiol.1926.sp002281.

[87] Julien Dupeyroux, Stein Stroobants, and Guido C.H.E. De Croon. *A toolbox for neuromorphic perception in robotics. 2022 8th International Conference on Event-Based Control, Communication, and Signal Processing (EBCCSP).* June 2022, pp. 1–7. DOI: doi.org/10.1109/EBCCSP56922.2022.9845664.

[88] David Heeger. *Poisson Model of Spike Generation.* Tech. rep. New York University, Sept. 2000.

[89] Michael Hough, Michael Korkin, Felix Gers, and Norberto Nawa. *SPIKER: Analog Waveform to Digital Spiketrain Conversion in ATR&apos;s Artificial Brain (CAM-Brain) Project* (Sept. 1999).

[90] B. Schrauwen and J. Van Campenhout. *BSA, a fast and accurate spike train encoding scheme. Proceedings of the International Joint Conference on Neural Networks, 2003.* Vol. 4. July 2003, pp. 2825–2830. DOI: doi.org/10.1109/IJCNN.2003.1224019.

[91] Yuval Zaidel, Albert Shalumov, Alex Volinski, Lazar Supic, and Elishai Ezra Tsur. *Neuromorphic NEF-Based Inverse Kinematics and PID Control. Frontiers in Neurorobotics* 15 (Feb. 2021). DOI: doi.org/10.3389/fnbot.2021.631159.

[92] Wulfram Gerstner and Werner M. Kistler. *Because the sequence of action potentials generated by a given stimulus varies from trial to trial, neuronal responses are typically treated statistically or probabilistically.* Cambridge, UK: Cambridge University Press, 2022. ISBN: 9780511815706.

[93] F. Theunissen and J.P. Miller. *Temporal encoding in nervous systems: A rigorous definition. Journal of Computational Neuroscience* 2 (Feb. 1995), pp. 149–162. DOI: doi.org/10.1007/BF00961885.

[94] Wenzhe Guo, Mohammed E. Fouda, Ahmed Eltawil, and Khaled Salama. *Neural Coding in Spiking Neural Networks: A Comparative Study for Robust Neuromorphic Systems. Frontiers in Neuroscience* 15 (Mar. 2021). DOI: doi.org:10.3389/fnins.2021.638474.

[95] P. Lichtsteiner and T. Delbruck. *A 64x64 aer logarithmic temporal derivative silicon retina. Research in Microelectronics and Electronics, 2005 PhD*. Vol. 2. July 2005, pp. 202–205. DOI: doi.org/10.1109/RME.2005.1542972.

[96] Kasabov Nikola, Matthew Scott Nathan, Tu Enmei, Marks Stefan, Sengupta Neelava, Capecci Elisa, Othman Muhaini, Gholami Doborjeh Maryam, Murli Norhanifah, Hartono Reggio, Israel Espinosa-Ramos Josafath, Zhou Lei, Bashir Alvi Fahad, Wang Grace, Taylor Denise, Feigin Valery, Gulyaev Sergei, Mahmoud Mahmoud, Hou Zeng-Guang, and Yang Jie. *Evolving spatio-temporal data machines based on the NeuCube neuromorphic framework: Design methodology and selected applications. Neural Networks* 78 (June 2016), pp. 1–14. DOI: doi.org/10.1016/j.neunet.2015.09.011.

[97] Wei Fang, Zhaofei Yu, Yanqi Chen, Timothée Masquelier, Tiejun Huang, and Yonghong Tian. *Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks. 2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021, pp. 2641–2651. DOI: doi.org/10.1109/ICCV48922.2021.00266.

[98] Sander M. Bohté, Joost N. Kok, and Han La Poutré. *SpikeProp: backpropagation for networks of spiking neurons. The European Symposium on Artificial Neural Networks*. Apr. 2000, pp. 419–424.

[99] Si Wu, Shun-ichi Amari, and Hiroyuki Nakahara. *Population Coding and Decoding in a Neural Field: A Computational Study. Neural Computation* 14.5 (May 2002), pp. 999–1026. DOI: doi.org/10.1162/089976602753633367.

[100] D.H. Hubel and T.N. Wiesel. *Receptive fields of single neurones in the cat's striate cortex. The Journal of Physiology* 148.3 (Oct. 1959), pp. 574–591. DOI: doi.org/10.1113/jphysiol.1959.sp006308.

[101] Robert Patton, Catherine Schuman, Shruti Kulkarni, Maryam Parsa, J. Parker Mitchell, N. Quentin Haas, Christopher Stahl, Spencer Paulissen, Prasanna Date, Thomas Potok, and Shay Snyder. *Neuromorphic Computing for Autonomous Racing. International Conference on Neuromorphic Systems 2021*. New York, NY, USA: Association for Computing Machinery, 2021. DOI: doi.org/10.1145/3477145.3477170.

[102] Iulia-Maria Comşa, Luca Versari, Thomas Fischbacher, and Jyrki Alakuijala. *Spiking Autoencoders With Temporal Coding. Frontiers in Neuroscience* 15 (Aug. 2021). DOI: doi.org/10.3389/fnins.2021.712667.

[103] Justus F. Hübotter, Pablo Lanillos, and Jakub M. Tomczak. *Training Deep Spiking Autoencoders without Bursting or Dying Neurons through Regularization* (Sept. 2021). DOI: doi.org/10.48550/arXiv.2109.11045.

[104] Ruoshi Li, Xin Xu, Lihong Wan, Qingdu Li, Ye Yuan, and Na Liu. *Autoencoder Induced Deep Spiking Neural Network. Proceedings of the 2023 7th International Conference on Innovation in Artificial Intelligence*. New York, NY, USA: Association for Computing Machinery, July 2023, pp. 147–153. DOI: doi.org/10.1145/3594409.3594437.

[105] Hiromichi Kamata, Yusuke Mukuta, and Tatsuya Harada. *Fully Spiking Variational Autoencoder*. Dec. 2021. DOI: doi.org/10.48550/arXiv.2110.00375.

[106] Sanaullah, Shamini Koravuna, Ulrich Rückert, and Thorsten Jungeblut. *Exploring spiking neural networks: a comprehensive analysis of mathematical models and applications. Frontiers in Computational Neuroscience* 17 (Aug. 2023). DOI: doi.org/10.3389/fncom.2023.1215824.

[107] A. L. Hodgkin and A.F. Huxley. *A quantitative description of membrane current and its application to conduction and excitation in nerve. The Journal of Physiology* 117.4 (Aug. 1952), pp. 500–544. DOI: doi.org/10.1113/jphysiol.1952.sp004764.

[108] Alla Borisyuk. *Morris–Lecar Model. Encyclopedia of Computational Neuroscience*. Ed. by Dieter Jaeger and Ranu Jung. New York, NY: Springer New York, 2015, pp. 1758–1764. ISBN: 978-1-4614-6675-8.

[109] J. Nagumo, S. Arimoto, and S. Yoshizawa. *An Active Pulse Transmission Line Simulating Nerve Axon. Proceedings of the IRE* 50.10 (Oct. 1962), pp. 2061–2070. DOI: doi.org/10.1109/JRPROC.1962.288235.

[110] S.L. Hindmarsh and R.M. Rose. *A model of neuronal bursting using three coupled first order differential equations. The Royal Society*. Vol. 221. 1222. Mar. 1984, pp. 87–102. DOI: doi.org/10.1098/rspb.1984.0024.

[111] E.M. Izhikevich. *Simple model of spiking neurons. IEEE Transactions on Neural Networks* 14.6 (Nov. 2003), pp. 1569–1572. DOI: doi.org/10.1109/TNN.2003.820440.

[112] S. Mihalaș and E. Niebur. *A generalized linear integrate-and-fire neural model produces diverse spiking behaviors. Neural computation* 21.3 (Mar. 2009), pp. 704–718. DOI: doi.org/10.1162/neco.2008.12-07-680.

[113] A. N. Burkitt. *A Review of the Integrate-and-fire Neuron Model: I. Homogeneous Synaptic Input. Biological Cybernetics* 95 (Apr. 2006), pp. 1–19. DOI: doi.org/10.1007/s00422-006-0068-6.

[114] Renaud Jolivet, Timothy J. Lewis, and Wulfram Gerstner. *Generalized Integrate-and-Fire Models of Neuronal Activity Approximate Spike Trains of a Detailed Model to a High Degree of Accuracy. Journal of Neurophysiology* 92.2 (Aug. 2004), pp. 959–976. DOI: doi.org/10.1152/jn.00190.2004.

[115] Romain Brette and Wulfram Gerstner. *Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity. Journal of Neurophysiology* 94.5 (Nov. 2005), pp. 3637–3642. DOI: doi.org/10.1152/jn.00686.2005.

[116] Nicolas Brunel and Peter E. Latham. *Firing Rate of the Noisy Quadratic Integrate-and-Fire Neuron. Neural Computation* 15.10 (Oct. 2003), pp. 2281–2306. DOI: doi.org/10.1162/089976603322362365.

[117] Guido Gigante, Paolo Del Giudice, and Maurizio Mattia. *Frequency-dependent response properties of adapting spiking neurons. Mathematical Biosciences* 207.2 (June 2007), pp. 336–351. DOI: doi.org/10.1016/j.mbs.2006.11.010.

[118] Wulfram Gerstner. *Spike-response model. Scholarpedia* 3.12 (Dec. 2008). DOI: doi:10.4249/scholarpedia.1343.

[119] Sam McKennoch, Thomas Voegtlin, and Linda Bushnell. *Spike-Timing Error Backpropagation in Theta Neuron Networks. Neural Computation* 21.1 (Jan. 2009), pp. 9–45. DOI: doi.org/10.1162/neco.2009.09-07-610.

[120] Mohamed Sadek Bouanane, Dalila Cherifi, Elisabetta Chicca, and Lyes Khacef. *Impact of spiking neurons leakages and network recurrences on event-based spatio-temporal pattern recognition. Frontiers in Neuroscience* 17 (Nov. 2023). DOI: doi.org/10.3389/fnins.2023.1244675.

[121] R. Gütig and H. Sompolinsky. *The tempotron: a neuron that learns spike timing–based decisions. Nature of Neuroscience* 9 (Feb. 2006), pp. 420–428. DOI: doi.org/10.1038/nn1643.

[122] X. Zhang, G. Foderaro, C. Henriquez, A. M. J. van Dongen, and S. Ferrari. *A Radial Basis Function Spike Model for Indirect Learning via Integrate-and-Fire Sampling and Reconstruction Techniques. Advances in Artificial Neural Systems* 2012 (Oct. 2012). DOI: doi.org/10.1155/2012/713581.

[123] Hyeryung Jang, Osvaldo Simeone, Brian Gardner, and Andre Gruning. *An Introduction to Probabilistic Spiking Neural Networks: Probabilistic Models, Learning Rules, and Applications. IEEE Signal Processing Magazine* 36.6 (Nov. 2019), pp. 64–77. DOI: doi.org/10.1109/MSP.2019.2935234.

[124] Yufei Guo, Xuhui Huang, and Zhe Ma. *Direct learning-based deep spiking neural networks: a review. Frontiers in Neuroscience* 17 (June 2023). DOI: doi.org/10.3389/fnins.2023.1209795.

[125] Peter Diehl and Matthew Cook. *Unsupervised learning of digit recognition using spike-timing-dependent plasticity. Frontiers in Computational Neuroscience* 9 (Aug. 2015). DOI: doi.org10.3389/fncom.2015.00099.

[126] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. *Generative Adversarial Networks.* June 2014. DOI: doi.org/10.48550/arXiv.1406.2661.

[127] Bleema Rosenfeld, Osvaldo Simeone, and Bipin Rajendran. *Spiking Generative Adversarial Networks With a Neural Network Discriminator: Local Training, Bayesian Models, and Continual Meta-Learning.* Nov. 2021. DOI: doi.org/10.48550/arXiv.2111.01750.

[128] Vineet Kotariya and Udayan Ganguly. *Spiking-GAN: A Spiking Generative Adversarial Network Using Time-To-First-Spike Coding.* June 2021. DOI: doi.org/10.48550/arXiv.2106.15420.

[129] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need.* June 2017. DOI: doi.org/10.48550/arXiv.1706.03762.

[130] Zhaokun Zhou, Yuesheng Zhu, Chao He, Yaowei Wang, Shuicheng Yan, Yonghong Tian, and Li Yuan. *Spikformer: When Spiking Neural Network Meets Transformer.* Sept. 2022. DOI: doi.org/10.48550/arXiv.2209.15425.

[131] Murat Isik, Hiruna Vishwamith, Kayode Inadagbo, and I. Can Dikmen. *HPCNeuroNet: Advancing Neuromorphic Audio Signal Processing with Transformer-Enhanced Spiking Neural Networks.* Nov. 2023. DOI: doi.org/10.48550/arXiv.2311.12449.

[132] Qingyu Wang, Tielin Zhang, Minglun Han, Yi Wang, Duzhen Zhang, and Bo Xu. *Complex Dynamic Neurons Improved Spiking Transformer Network for Efficient Automatic Speech Recognition. Proceedings of the AAAI Conference on Artificial Intelligence* 37.1 (June 2023), pp. 102–109. DOI: doi.org/10.1609/aaai.v37i1.25081.

[133] Anonymous. *Spike-driven Transformer V2: Meta Spiking Neural Network Architecture Inspiring the Design of Next-generation Neuromorphic Chips. The Twelfth International Conference on Learning Representations.* Jan. 2024.

[134] Emmanouil Angelidis, Emanuel Buchholz, Jonathan Arreguit, Alexis Rougé, Terrence Stewart, Axel von Arnim, Alois Knoll, and Auke Ijspeert. *A spiking central pattern generator for the control of a simulated lamprey robot running on SpiNNaker and Loihi neuromorphic boards. Neuromorphic Computing and Engineering* 1.1 (Aug. 2021). DOI: doi.org/10.1088/2634-4386/ac1b76.

[135] Mathias Gehrig, Sumit Bam Shrestha, Daniel Mouritzen, and Davide Scaramuzza. *Event-Based Angular Velocity Regression with Spiking Networks.* Mar. 2020. DOI: doi.org/10.48550/arXiv.2003.02790.

[136] Friedemann Zenke and Surya Ganguli. *SuperSpike: Supervised Learning in Multilayer Spiking Neural Networks. Neural Computation* 30.6 (June 2018), pp. 1514–1541. DOI: doi.org/10.1162/neco_a_01086.

[137] Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, and Luping Shi. *Spatio-Temporal Backpropagation for Training High-Performance Spiking Neural Networks. Frontiers in Neuroscience* 12 (May 2018). DOI: doi.org/10.3389/fnins.2018.00331.

[138] Yan Xu, Xiaoqin Zeng, and Shuiming Zhong. *A New Supervised Learning Algorithm for Spiking Neurons. Neural Computation* 25.6 (June 2013), pp. 1472–1511. DOI: doi.org/10.1162/NECO_a_00450.

[139] Filip Ponulak and Andrzej Kasiński. *Supervised Learning in Spiking Neural Networks with ReSuMe: Sequence Learning, Classification, and Spike Shifting. Neural Computation* 22.2 (Feb. 2010), pp. 467–510. DOI: doi.org/10.1162/neco.2009.11-08-901.

[140] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. *Training Deep Spiking Neural Networks Using Backpropagation. Frontiers in Neuroscience* 10 (Nov. 2016). DOI: doi.org/10.3389/fnins.2016.00508.

[141] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. *Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks. IEEE Signal Processing Magazine* 36.6 (2019), pp. 51–63. DOI: 10.1109/MSP.2019.2931595.

[142] Sumit Bam Shrestha and Garrick Orchard. *SLAYER: Spike Layer Error Reassignment in Time.* Sept. 2018. DOI: doi.org/10.48550/arXiv.1810.08646.

[143] T. Bekolay, C. Kolbeck, and C. Eliasmith. *Simultaneous unsupervised and supervised learning of cognitive functions in biologically plausible spiking neural networks. Proceedings of the Annual Meeting of the Cognitive Science Society.* Vol. 35. University of Waterloo. Mar. 2013. URL: https://escholarship.org/uc/item/9k64b389.

[144] Yingyezhe Jin, Wenrui Zhang, and Peng Li. *Hybrid Macro/Micro Level Backpropagation for Training Deep Spiking Neural Networks. Advances in Neural Information Processing Systems.* Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc., 2018.

[145] Lin Xiang-hong, Wang Xiang-wen, and Dang Xiao-chao. *A New Supervised Learning Algorithm for Spiking Neurons Based on Spike Train Kernels. ACTA ELECTONICA SINICA* 44.12 (Dec. 2016), pp. 2877–2886. DOI: doi.org/10.3969/j.issn.0372-2112.2016.12.010.

[146] Aditya Gilra and Wulfram Gerstner. *Predicting non-linear dynamics by stable local learning in a recurrent spiking neural network. eLife* 6 (Nov. 2017). Ed. by Peter Latham. DOI: doi.org/10.7554/eLife.28295.

[147] W. Severa, C.M. Vineyard, R. Dellana, S. J. Verzi, and J. B. Aimone. *Training deep neural networks for binary communication with the Whetstone method. Nature Machine Intelligence* 1 (Jan. 2019), pp. 86–94. DOI: doi.org/10.1038/s42256-018-0015-y.

[148] S. Song, K. Miller, and L. Abbott. *Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. Nature Neuroscience* 3 (Sept. 2000), pp. 919–926. DOI: doi.org/10.1038/78829.

[149] Kendra S. Burbank. *Mirrored STDP Implements Autoencoder Learning in a Network of Spiking Neurons. PLOS Computational Biology* 11.12 (Dec. 2015), pp. 1–25. DOI: doi.org/10.1371/journal.pcbi.1004566.

[150] Jean-Pascal Pfister and Wulfram Gerstner. *Triplets of Spikes in a Model of Spike Timing-Dependent Plasticity. Journal of Neuroscience* 26.38 (Sept. 2006), pp. 9673–9682. DOI: doi.org/10.1523/JNEUROSCI.1425-06.2006.

[151] Timothée Masquelier and Simon J Thorpe. *Unsupervised Learning of Visual Features through Spike Timing Dependent Plasticity. PLOS Computational Biology* 3.2 (Feb. 2007), pp. 1–11. DOI: doi.org/10.1371/journal.pcbi.0030031.

[152] Peter Jedlicka. *Synaptic plasticity, metaplasticity and BCM theory. Bratislavské lekárske listy* 103 (Feb. 2002), pp. 137–143.

[153] John J. Wade, Liam J. McDaid, Jose A. Santos, and Heather M. Sayers. *SWAT: A Spiking Neural Network Training Algorithm for Classification Problems. IEEE Transactions on Neural Networks* 21.11 (Sept. 2010), pp. 1817–1830. DOI: doi.org/10.1109/TNN.2010.2074212.

[154] Răzvan V. Florian. *Reinforcement Learning Through Modulation of Spike-Timing-Dependent Synaptic Plasticity. Neural Computation* 19.6 (June 2007), pp. 1468–1502. DOI: doi.org/10.1162/neco.2007.19.6.1468.

[155] Sergio F. Chevtchenko, Yeshwanth Bethi, Teresa B. Ludermir, and Saeed Afshar. *A Neuromorphic Architecture for Reinforcement Learning from Real-Valued Observations.* July 2023. DOI: doi.org/10.48550/arXiv.2307.02947.

[156] Nicolas Frémaux and Wulfram Gerstner. *Neuromodulated Spike-Timing-Dependent Plasticity, and Theory of Three-Factor Learning Rules. Frontiers in Neural Circuits* 9 (Feb. 2016). DOI: doi.org/10.3389/fncir.2015.00085.

[157] Junqi Lu, Xinning Wu, Su Cao, Xiangke Wang, and Huangchao Yu. *An Implementation of Actor-Critic Algorithm on Spiking Neural Network Using Temporal Coding Method. Applied Sciences* 12.20 (Oct. 2022). DOI: doi.org/10.3390/app122010430.

[158] Ding Chen, Peixi Peng, Tiejun Huang, and Yonghong Tian. *Deep Reinforcement Learning with Spiking Q-learning.* Jan. 2022. DOI: doi.org/10.48550/arXiv.2201.09754.

[159] Lei Deng, Yujie Wu, Xing Hu, Ling Liang, Yufei Ding, Guoqi Li, Guangshe Zhao, Peng Li, and Yuan Xie. *Rethinking the performance comparison between SNNS and ANNS. Neural Networks* 121 (Jan. 2020), pp. 294–307. DOI: doi.org/10.1016/j.neunet.2019.09.005.

[160] Edgar Lemaire, Loïc Cordone, Andrea Castagneti, Pierre-Emmanuel Novac, Jonathan Courtois, and Benoît Miramond. *An Analytical Estimation of Spiking Neural Networks Energy Efficiency.* Oct. 2022. DOI: doi.org/10.48550/arXiv.2210.13107.

[161] Shuo Li, Erik van der Horst, Philipp Duernay, Guido C. H. E. de Croon, and Christophe De Wagter. *Visual Model-predictive Localization for Computationally Efficient Autonomous Racing of a 72-gram Drone.* May 2019. DOI: doi.org/10.48550/arXiv.1905.10110.

[162] Jelle Westenberger, Christophe De Wagter, and Guido C. H. E. de Croon. *Efficient Bang-Bang Model Predictive Control for Quadcopters. Unmanned Systems* 10.04 (2022), pp. 395–405. DOI: doi.org/10.1142/S2301385022410060.

[163] Federico Paredes-Vallés. *Self-Supervised Neuromorphic Perception for Autonomous Flying Robots.* Dissertation (TU Delft). Delft University of Technology, 2023. ISBN: 978-94-6366-755-5. DOI: doi.org/10.4233/uuid:f5f62ff1-28d8-4c5f-93f5-3f9d90d06d28.

[164] Craig Iaboni, Thomas Kelly, and Pramod Abichandani. *NU-AIR – A Neuromorphic Urban Aerial Dataset for Detection and Localization of Pedestrians and Vehicles.* Feb. 2023. DOI: doi.org/10.48550/arXiv.2302.09429.

Part III

Additional Work

A

Encoder

The aim of the current chapter is to detail what work has been performed to answer the first research question of the literature study: **How can the input/output values be transformed into spikes such that the lowest resources are used?**. For this, encoding procedure was analysed in detail but, in the end, it was discovered that it does not improve the general performance of the controller so it was discarded completely. Even though the work performed will be presented here including how the dataset was generated for this stage in section A.1. Secondly, the sensitivity analysis performed on the autoencoder will be detailed in section A.2 while the work required to answer the research question will be presented in section A.3. Lastly, the results that led to the discard of the encoder will be presented in section A.4. Even though not clearly part of the encoder work, it was decided to include here also the work of generating a training dataset with reinforcement learning focused on learning to fly to a hover position as was done with the energy optimal control problem. The work required and the results obtained can be found in section A.5. Similarly, in order to improve the performance of the SNN controller, weight decaying was implemented during the training phase and the analysis can be seen in section A.6.

A.1 Generating Dataset with RL

When the work for the current study started, the encoder was the initial concern. The feasibility of switching from floating point values to spiking values was the first question to be answered by the research. For this reason, to analyse the capability of encoders (and namely population encoding in the current case), a general dataset had to be created that contained as little bias as possible. The quality of the dataset for learning an SNN controller was not of great importance during this phase. With this mindset, it was decided to develop the dataset with the reinforcement learning approach created by Ferede et al.[23].[1] In order to generate it, a model had to be learned through reinforcement learning, and with its help, after running it through the environment several times, the dataset could be generated containing multiple parameters at every timestep. Another reason for choosing this approach compared to the supervised learning one was that the current reinforcement learning model could produce more parameters for the dataset. Thus, 24 inputs for every timestep were saved instead of 19 as the supervised learning model was producing.

In order to make the dataset more general and with as little bias as possible for a better verification of the encoder abilities, the creation of a more complex map is required that has to be learned through reinforcement learning. Thus, as presented during the literature study, the map shown in Figure A.1 - A.2 will be used to generate the dataset. With this, a model is learned to fly the circuit as fast as possible and then, on the same circuit, the drone flight is simulated

[1]https://github.com/tudelft/optimal_quad_control_RL

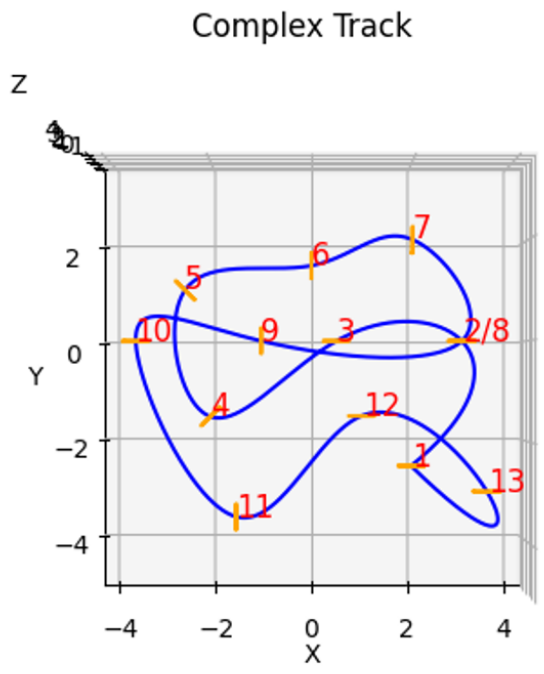


Figure A.1: The track used to create the encoder dataset, seen from above

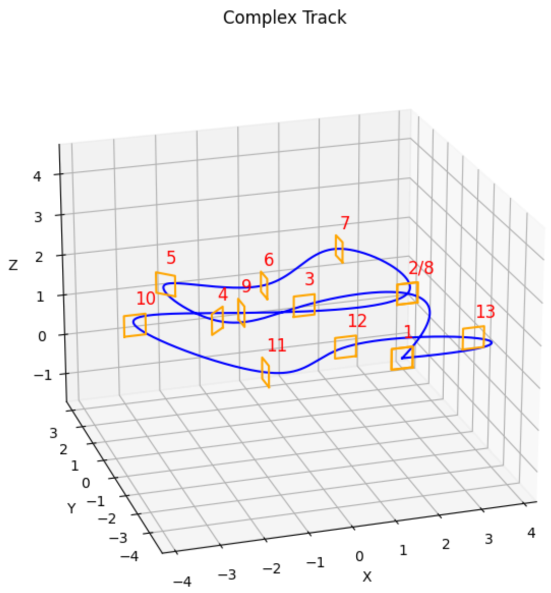


Figure A.2: The track used to create the encoder dataset, seen from perspective

for 2048 times for 10 seconds. With a timestep of 0.005 seconds, this leads to 2000 samples for each simulation. As mentioned above, each sample contains the input (that is the 3-dimensional position (x, y and z), velocity (v_x, v_y, v_z), angular displacement (ϕ, θ, ψ) and angular velocity (p, q, r) of the drone as well as the actual angular velocity of the propellers of the drone (w_1, w_2, w_3, w_4) and a 3-dimensional disturbance momentum ($M_{ext,x}$, $M_{ext,y}$, $M_{ext,z}$) and force in the vertical z-direction ($F_{ext,z}$) as well as the next gate 3-dimensional position ($gate_x$, $gate_y$, $gate_z$) and yaw ($gate_{yaw}$)). Moreover, the target output is also saved at every timestep containing the output command between 0 and 1 for each propeller (u_1, u_2, u_3, u_4).

For the encoder the following structure has been chosen. The dataset is initially normalised between 0 and 1 for all the parameters and structured in such as manner that one dataset sample will contain the 2000 timesteps mentioned above to perform recursion with SNN. Following this, the dataset is then split in 2 for training and validation following a 9 to 1 ratio. Later, to train the network, a similar structure as presented in section B.1 was used. Thus, with the ADAM optimiser and with a learning rate of 0.01 but no learning rate scheduler, the SNN started training for several number of epochs. Using the MSE loss to quantify the performance, the model with the lowest error on the validation dataset is chosen as the best one. The SNN is formed of one spiking layer between input and output. The input is formed of all the parameters that have to be encoded while the target output is simply the same input list which has to be reproduced. The structure of the resulting network can be seen in Figure A.3.

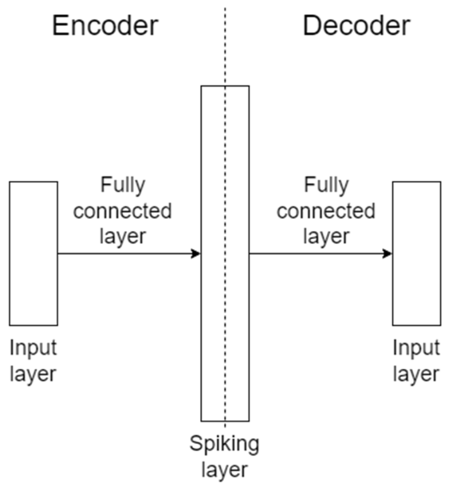


Figure A.3: The structure of the encoder SNN

With the created dataset and the equivalent trained controller, the encoding results could be simulated for all the input and output parameters. The results can be visualised in Figure A.4

where the signal was simulated for a new validation dataset which is comprised of 5 samples, each of 2000 timesteps. As it can be observed, the output of the spiking network is not very accurate. Even though for some parameters, the network learns how to encode the input such as the position and velocity signals, for the angles and angular rates the performance is much worse and shows no clear learning capabilities. Moreover, the performance of encoding the disturbance moments and gates is very noisy which was not expected due to the discrete nature of the parameters.

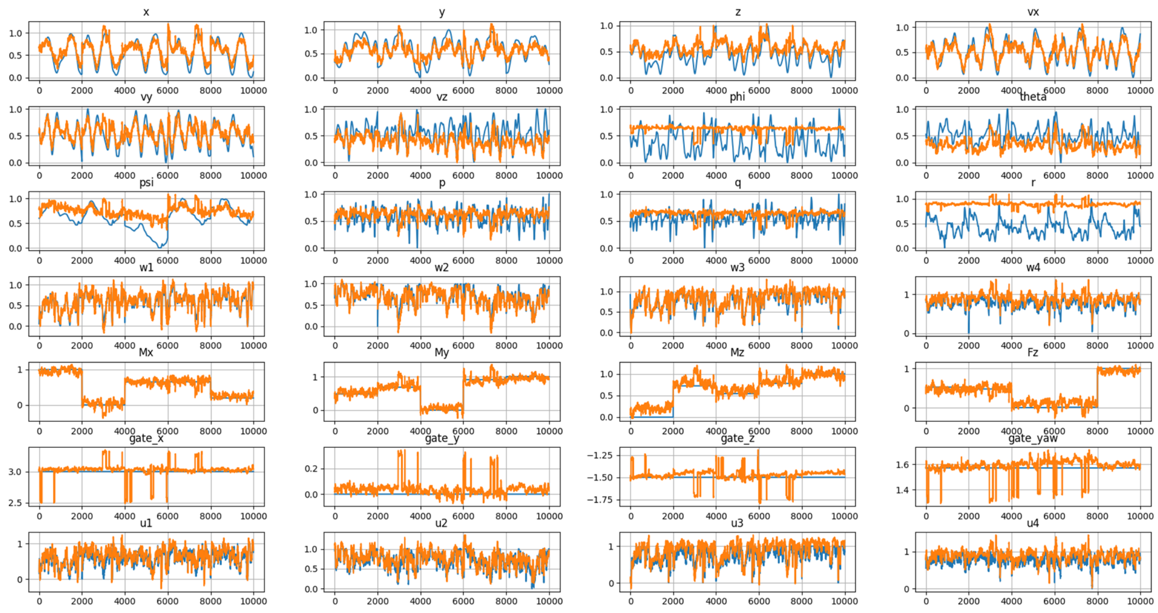


Figure A.4: The results of the first encoder trained on the dataset generated with reinforcement learning. The figure contains both the input and output parameters. In blue there is the target signal while in orange there is the reproduced signal with SNN;

Once the 2 problems were identified, the search for solving them started. It was discovered that the poor performance of the angles and angular rates was given by a poor normalisation technique as well as by a bad dataset which included some samples that were close to crashing. This pushed the minimum and maximum values of the angles and angular rates to extremes which, after normalisation, forced the output to be in a very small interval of values. This small interval is not favourable for SNN as it requires a very large number of spiking neurons with small weights for accurately identifying the small differences present in the dataset now. Moreover, with the help of large weight values, several neurons have to be used for extreme cases and can be considered wasted as repeating them is improbable. Thus, to solve this issue, a new dataset was generated with all the samples having stable intervals and not being close to crashing. This meant the target signal as well as the output of the SNN could explore a larger spectrum of values more accurately. However, the problem of the moment and force disturbances and of the gate position and yaw values poor performance, could only be rooted to the influence of other parameters. As it will be analysed in section A.3, the solution for this problem involved the usage of different training for groups of parameters with specific characteristics.

A.2 Sensitivity Analysis

The current section will focus on the sensitivity analysis performed on the built encoder. Before the study of the controller started, an initial sensitivity analysis was done on the encoder procedure for 2 main reasons. Firstly, the performance of the encoder had to be increased as, before attaching it to the controller, accurate results ought to be expected. Secondly, as the work was pioneering a novel domain, a better understanding of the SNNs and how they function was done with the help of this step. Moreover, it is important to mention that this step required the implementation of the TU Delft's Delftblue supercomputer. As several programs had to be run in parallel, for a better time efficiency and a more thorough analysis, it was decided to use the capabilities of the Delftblue supercomputer. The obtained results and models were then analysed on the personal computer and can be visualised below.

First analysis results focused on the influence of initial leak and threshold values upon the performance of the network. It is important to note that both the leak current coefficient ($leak_i$) as well as the leak voltage coefficient ($leak_v$) are considered as one instance and thus were modelled in a similar manner and will be analysed together. To reiterate, the values are generated in a random manner with the help of a normal distribution function where the mean and variance are passed as can be seen in the table below. Again, as mentioned in the main thesis paper work, it is important to note that the leak values presented represent the value passed to the sigmoid function to be bounded. For this reason, a low leak value of -2, after passed through the sigmoid function, is closer to 0 meaning that a great part of the past information is forgotten. This translates actually, in a contradictory manner, to a large leak of information every timestep. Similarly, the opposite happens when a large leak value is passed to the sigmoid function, leading, in the end, to a small leak of information over every timestep. Compared to the analysis done in the main paper work, the current one is focused more on the general behaviour of the parameters. That is the reason why during this phase the standard deviation (STD) was analysed as well as large intervals were used for the mean of the threshold and leak values as the general trend should be observed. Tweaking the parameters was not analysed in detail during this step.

With these observations in mind, the performance of the SNN as a function of the mean and standard deviation of initial leak and threshold values can be observed in Table A.1. As can be seen, the best performance is achieved when the leak value is modelled with a normal distribution with a low mean of -2 and a standard deviation of 1.5 while the threshold also needs a low mean of 1 leading to an MSE loss of 3.529e-3 of the encoder. Overall some trends can be observed. For example, low leak values are preferred which means that the past information is not very important for the encoder. This means that the SNN does not use recursion a lot and just prefers to transform the input directly to output for every timestep. This can be explained by the little correlation over time observed in the dataset as it is the case for the output command and the propeller rotational speed parameters (as observed in Figure A.4) which are generally very noisy and do not depend on previous timesteps. Thus, with a high variance in the dataset it is more difficult to learn a relation over time and the model just focuses to predict only during the current timestep. As it was observed in the controller sensitivity analysis, a slightly higher value is observed closer to 0. This can be rooted to a similar cause as the dataset output command is smoother in the energy optimal dataset and thus easier to follow.

Regarding the standard deviation of the leak values a clear trend cannot be observed as the performances fluctuates irrespective of the value of the mean. Instead, it can be observed that when a wrong combination of initial threshold and leak values is chosen, a higher standard deviation will lead to a smaller MSE loss. This can be explained by the fact that a higher standard deviation will lead to some values closer to the correct leak and threshold optimal values which will boost the performance in the end. Similarly, when the error is relatively low, a standard deviation of 0 will keep the error low. Thus if a standard deviation of 0 is preferred, it can be used as a confirmation of the right selection of threshold and leak initial values mean. Regarding the analysis of the threshold value, a similar observation as done for the controller can be done for the encoder as well. Thus, a smaller value is definitely preferred which translates in the need of the SNN to react fast to different changes. This was expected as the input signal changes very fast due to the noisy behaviour observed for output command and propeller rotational speed.

Table A.1: Influence of threshold and leak initial value selection on MSE loss of the encoder

Leak Value \ Threshold Value	Mean=1 STD=0.5	Mean=5 STD=0.5
Mean=-2; STD=0	3.931e-3	4.443e-3
Mean=-2; STD=0.5	3.712e-3	4.685e-3
Mean=-2; STD=1.5	3.529e-3	4.962e-3
Mean=0; STD=0	3.606e-3	2.512e-2
Mean=0; STD=0.5	4.157e-3	2.194e-2
Mean=0; STD=1.5	4.974e-3	8.236e-3
Mean=2; STD=0	7.056e-3	2.727e-2
Mean=2; STD=0.5	6.826e-3	2.538e-2
Mean=2; STD=1.5	6.744e-3	1.988e-2

The next parameters considered during the sensitivity analysis were the dataset size and the number of epochs required for training and their influence on the performance. The results can be observed in Figure A.5 where the performance of models trained on various dataset sizes is shown on a separate validation dataset along different epochs. As expected, the higher the dataset size and the more epochs are being used, the higher the performance of the SNN model. Moreover, the convergence trend along different epochs can be seen the most clear for the smallest dataset which does not seem to have converged fully not even after 5 epochs. Otherwise, the performance line of the models varies greatly when close to convergence with various up and downs shifts that depend solely on the optimisation procedure and efficiency. An important thing to mention during the analysis was that the maximum number of epochs was chosen to be 5 as it was discovered during the previous study that most models converge very fast and do not require more than 3-5 epochs to achieve convergence. After this convergence is reached, the model performance fluctuates around the convergence MSE loss for the remaining number of epochs.

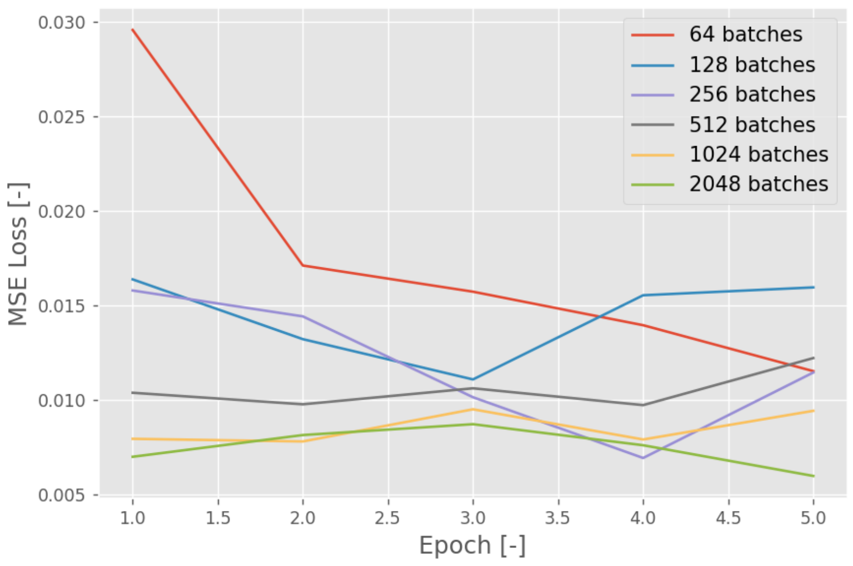


Figure A.5: The influence of the dataset size upon the network performance over multiple epochs

Last but not least, a special analysis of the trained models was done on a longer dataset than trained on. As mentioned previously, one sample of dataset contained 10 seconds of simulation but the current analysis wanted to verify how the model would perform if simulated on a much longer time, namely 50 seconds (or 10000 timestep). The resulting behaviour can be observed in Figure A.6 - A.7. First of all, by looking at the encoding of position and velocity, it can be seen that having a higher timestep than trained on does not influence the results. This was expected as the SNN prefers encoding every timestep separately as seen in the analysis of the initial leak values and does not get influenced by the length of the sample. On the other hand, doing the sample longer, the value for the heading angle ψ keeps increasing over the maximum allowed value. This makes the encoder SNN to not be able to encode the value any longer and flattens at a stable maximum value. It is important to see that this problem affects all the other angles and angular velocity parameters as well with a decrease in performance after the maximum value is overpassed. Moreover, the model showing the current plots was trained to encode all parameters at once showing that there is a strong correlation between the angles and angular velocities but no correlation when comparing also to the position and velocities. This observation that grouping parameters for a better encoding was used during the following section when the number of neurons had to be optimised and a better performance was achieved with this approach.

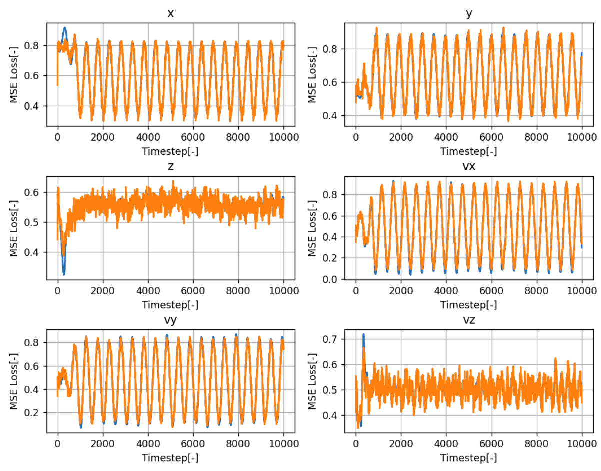


Figure A.6: The comparison between target encoder output (blue) and predicted output by the encoder (orange) on the position and velocity parameters of a dataset longer that the sample size;

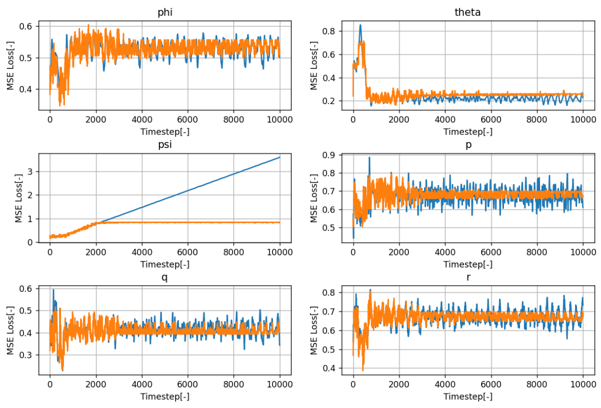


Figure A.7: The comparison between target encoder output (blue) and predicted output by the encoder (orange) on the angles and angular velocity parameters of a dataset longer that the sample size;

A.3 Number of Neurons Optimisation

The current section will present the sensitivity analysis performed during the encoding to understand how the number of spiking neurons can be reduced to achieve a good performance out of the list of floating point values. The first approach considered for encoding was to send all the input parameters at once through the neuromorphic autoencoder. In the current approach, the sensitivity analysis involved changing the number of neurons in the layer with spiking neurons. After several models were trained with various number of spiking neurons, the resulting performance plot can be seen in Figure A.8. The outcome confirms the results found while analysing the controller performance (namely 7-8 spiking neurons per conventional neuron) and suggests that the optimal number to achieve good performance is to use 200 neurons for 28 floating point values. At 200 neurons, the performance flattens and adding more neurons will either slightly increase or decrease the performance loss. Decreasing the number of spiking neurons will drastically raise the performance loss which is also undesired.

However, another approach considered during the encoding phase was to encode parameters in groups based on similarity. The similarity means that in the group the parameters should have

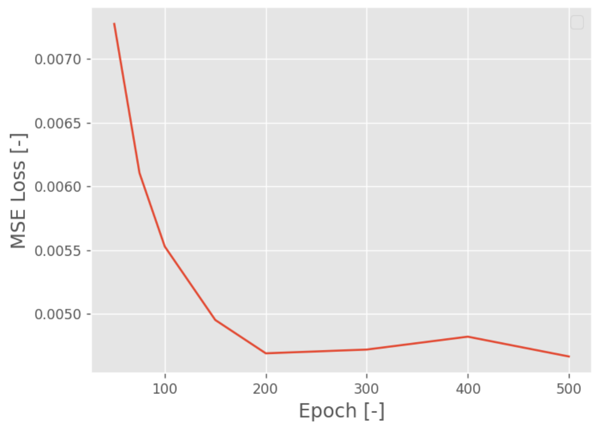


Figure A.8: The influence of the number of neurons on the training performance of the encoder when all input parameters are trained at once;

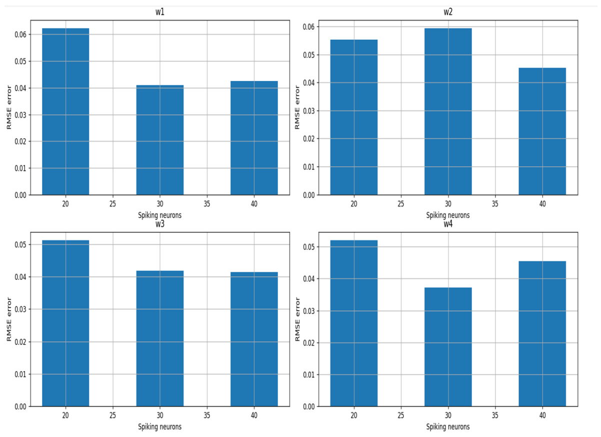


Figure A.9: The influence of the number of neurons on the training performance of the encoder when only the propeller angular velocity is considered;

a correlation between them. For example, position and velocities can be considered similar as velocities represent the direct derivative in time of the position. This behaviour was observed in Figure A.6 and Figure A.7 when a problem in the heading angle ψ led to bad encoding only for other angles and angular velocities but not for position and velocities. For this reason, the following groups of parameters were created:

- **Positions and velocities:** Including the positions x, y and z as well as the equivalent velocities v_x, v_y and v_z;
- **Angles and angular velocities:** Including the angles ϕ, θ and ψ as well as the equivalent angular velocities p, q and r;
- **Propeller angular velocities:** Even though not having a lot of correlation between them, it was chosen to group these parameters together to not affect the other parameters. Including the angular velocities of all 4 propellers w_1, w_2, w_3, w_4;
- **Next gate positions and yaw:** Including the positions of the next gate $gate_x$, $gate_y$ and $gate_z$ as well as the yaw $gate_{yaw}$;
- **Disturbances:** Even though not having a lot of correlation between them, it was chosen to group these parameters together to not affect the other parameters. Including the 3 moment disturbances on x-axis $M_{ext,x}$, on y-axis $M_{ext,y}$ and on z-axis $M_{ext,z}$ and the disturbance force on z-direction $F_{ext,z}$;
- **Output commands:** Separate to use these parameters for decoding. Including the output commands for all 4 propellers u_1, u_2, u_3, u_4;

With the groups of parameters realised (batches), it was important to find how many spiking neurons to assign for each group to have a well-performing encoding. Unfortunately, the sensitivity analysis was not so thorough for the groups of parameters and it involved just a small sensitivity analysis for the propeller angular velocities. The plot can be seen in Figure A.9 and it was chosen to continue with 30 spiking neurons as the performance loss seems to plateau at that value with only a very slight decrease in performance when 40 neurons are used. Based on this finding, the following number of spiking neurons were proposed for each group of parameters:

- **Positions and velocities:** 45 neurons chosen based on proportionality as 6 parameters are used compared to 4;

- **Angles and angular velocities:** 45 neurons chosen based on proportionality as 6 parameters are used compared to 4;
- **Propeller angular velocities:** 30 neurons;
- **Next gate positions and yaw:** 20 neurons as the task to encode the values is more trivial as there is little fluctuation in the dataset;
- **Disturbances:** 20 neurons as the task to encode the values is more trivial as there is little fluctuation in the dataset;
- **Output commands:** 30 neurons chosen based on proportionality as a similar number of parameters is used;

As can be seen, summing up all the spiking neurons of all the groups leads to a value of 190 neurons which is below the value of 200 neurons when parameters are encoded all together. It was decided to continue with less neurons as the performance is already better as will follow from the following analysis and further sensitivity was not required. Moreover, a relatively fair comparison between the 2 approaches had to be done and thus having a similar number of spiking neurons helped. In order to train the encoder and keep the groups of parameters separated a mask was applied to both weight matrices before and after the encoding layer. The mask matrix was built of the same size as the weight matrix and the multiplication was done element wise. Based on the input (and implicitly the output) order, the mask matrices were build of block matrices of 1s on the main diagonal. In total there were 6 block matrices and their size was given by the number of floating point parameters and by the number of expected spiking neurons. For a better understanding of the implementation check section B.1 where the code required for masking the parameters was built.

However, before showing the final results and their analysis, it is important to explain how the disturbance group of parameters was actually encoded. When encoding the parameters in batches, it was discovered that the performance was terrible and this can be visualised in Figure A.10. This could be explained for several reasons. First of all, the disturbances are generated randomly and thus there is no temporal correlation between consecutive timesteps which does not benefit SNNs. For this reason, the fluctuating behaviour (the big orange blocks in the image are a continuous up and down behaviour in the signal) is happening because the SNN does not know which value to converge to. Secondly, the disturbances are constant over the whole sample of 2000 timesteps. The SNN is not very accurate in keeping constant random values as it is used to behave similar to a signal that changes over time due to the inherent nature of spiking neurons to not spike continuously. Thus, passing constant random values is the worst feature that can be processed by SNNs. For this reason, it was considered that this group of parameters will be encoded differently with binning. This encoding method is also a population encoding method that assigns for every sub-interval a unique combination of spikes. What makes binning special is that the interval is split in equal sub-intervals. In order to build this encoding method, equal number of spiking neurons was assigned for each of the 4 parameters (5 spiking neurons per parameter). This leads to 2^5 combinations per parameter which means that the 0 to 1 interval was split in 32 sub-intervals. Thus to encode a value, the number should fall into a sub-interval and thus gets assigned a combination of spikes. The resulting improvement can be seen in Figure A.15.

With the encoding procedure clarified, it is important now to analyse the encoder output closely as well as to compare the performance of training the encoder using groups of parameters or training all the parameters at once. The first group of parameters considered are the positions and the velocities. As seen in Figure A.11, training in batches is clearly preferred. By looking at the plot on the left side, it can be seen that the encoder manages to follow the target signals

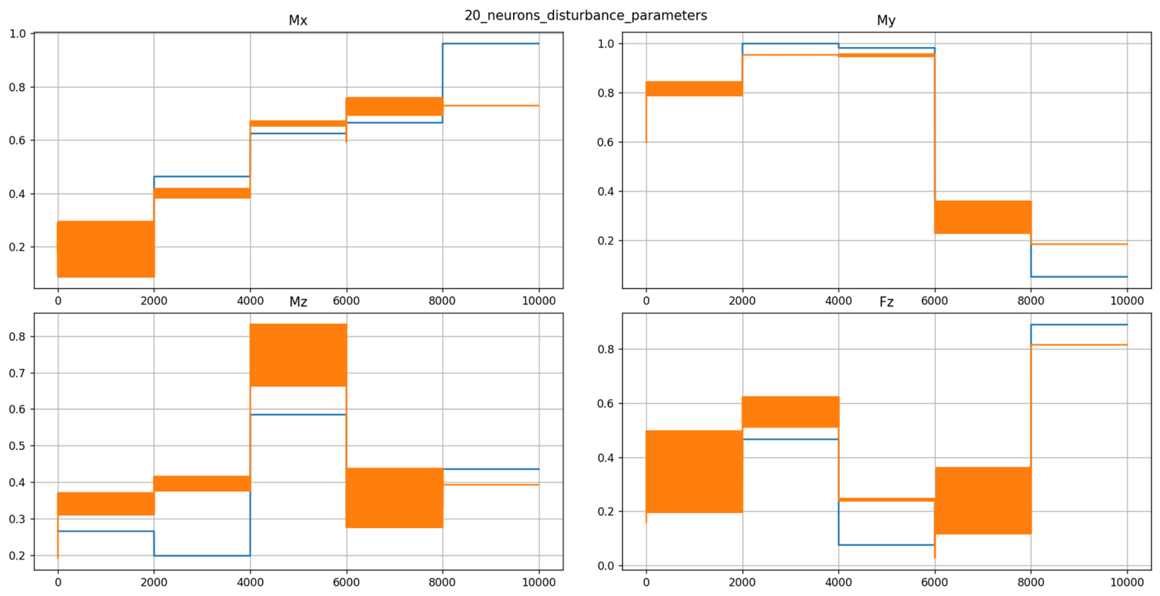


Figure A.10: The performance of the encoder model trained only on the disturbance parameters before binning is applied;

very close for all the parameters. A peculiar thing can be observed for the signal of z which is much noisier than the other signals. This can be rooted to the dataset which has a very small range for the values of z as the target z-position of all the gates is the same at -1 and very few room for flying around is allowed. Moreover, this signal is not periodic as the one for the x and y which follow a sinusoidal graph. Thus, the movement in z-axis is very irregular which makes the signal look more noisy. The error shown on the right side is the absolute error between signals and is most likely caused by the "noisy" behaviour specific to SNNs.

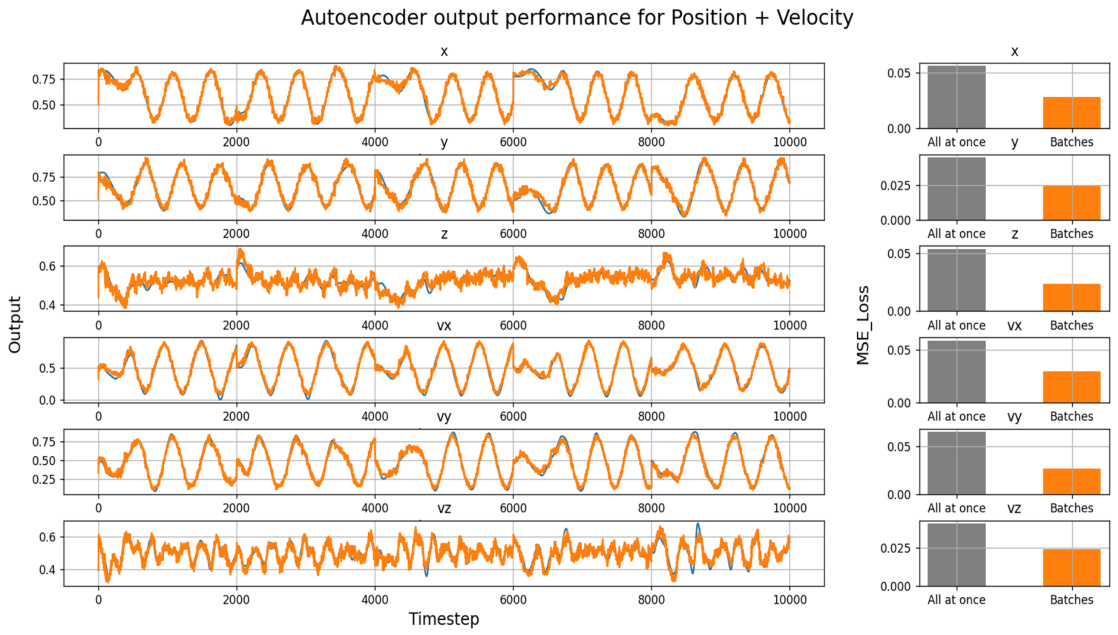


Figure A.11: The comparison between target output (blue) of the position and velocity parameters and the encoder estimation (orange) on the left. On the right the performance comparison between the 2 encoder models where the parameters are trained as one batch versus when all parameters are encoded all together. The left plot uses the encoder model trained only on the position and velocity parameters;

The same behaviour can be seen when analysing the encoding of angles and angular velocities as seen in Figure A.12. Thus, training in batches is preferred and the encoding signal is followed closely irrespective of the parameter. Even though, a weird thing happens for all parameters at 8000 timesteps. In order to understand this weird behaviour, the spiking activity of all neurons was plotted and can be seen in Figure A.13. The peculiar behaviour was discovered to be caused by one neuron (namely 30) which behaves normally as a bias neuron for all parameters. However, when 8000 timesteps gets reached, the neuron does not spike for one timestep. The reason for this could be that a new sample is being started which means translates in a sudden change in the input signal which further causes this weird behaviour. However, this is not consistent as previous sample changes happened but neuron 30 did not stop spiking.

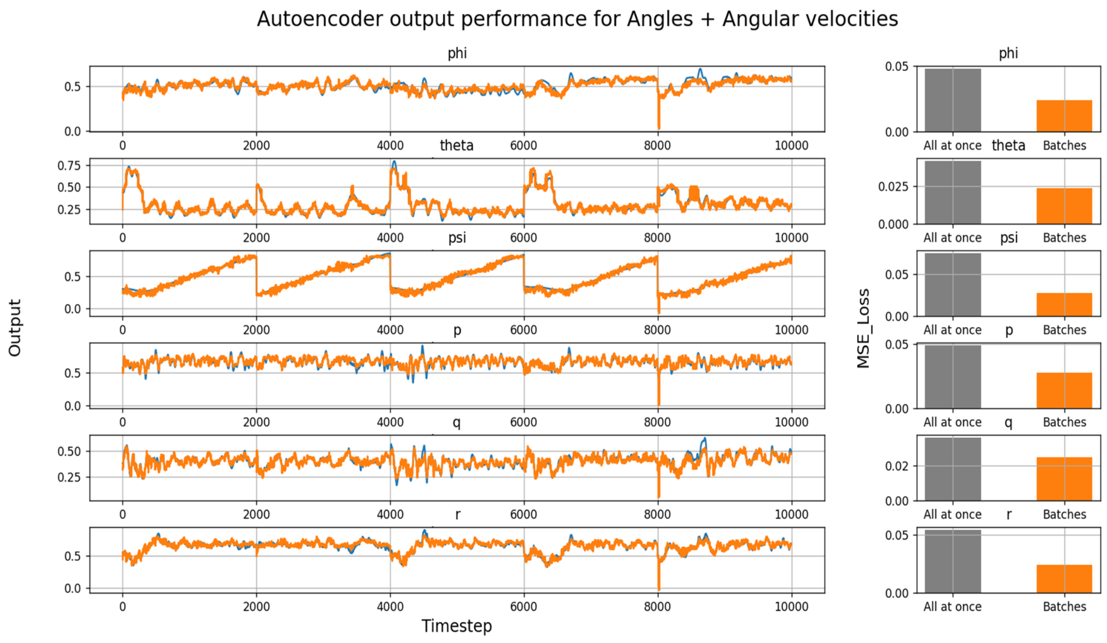


Figure A.12: The comparison between target output (blue) of the angles and angular velocity parameters and the encoder estimation (orange) on the left. On the right the performance comparison between the 2 encoder models where the parameters are trained as one batch versus when all parameters are encoded all together. The left plot uses the encoder model trained only on the angles and angular velocities parameters;

Figure A.14 shows a similar behaviour as shown also for the previous plots. Thus, training in batches is again preferred and the performance of encoding is very good. However, this time, the improvement shown by training in batches compared to training all parameters at once is much smaller. This can be explained as there is little correlation between parameters generally. Thus training in batches does not bring a great advantage within itself just that the encoding is not affected by the values of the other parameters.

Next we have the disturbances which can be seen in Figure A.15. This time the comparison is done between training all parameters at once and using binning as described above. As expected, using binning helps a lot the controller achieving very low errors while the disturbances encoded all at once suffer from the same problem as mentioned above.

Next the gate positions and yaw performance can be seen in Figure A.16. As expected training in batches benefits these parameters the most which are no longer affected by the inputs of other parameters. Thus, the absolute error gets very close to 0 as the input is periodic over time and has some discrete predefined values. This behaviour benefits SNN the most and can be easily encoded by it. However, in the output plot, weird spikes can be seen at 0, 6000 and 8000

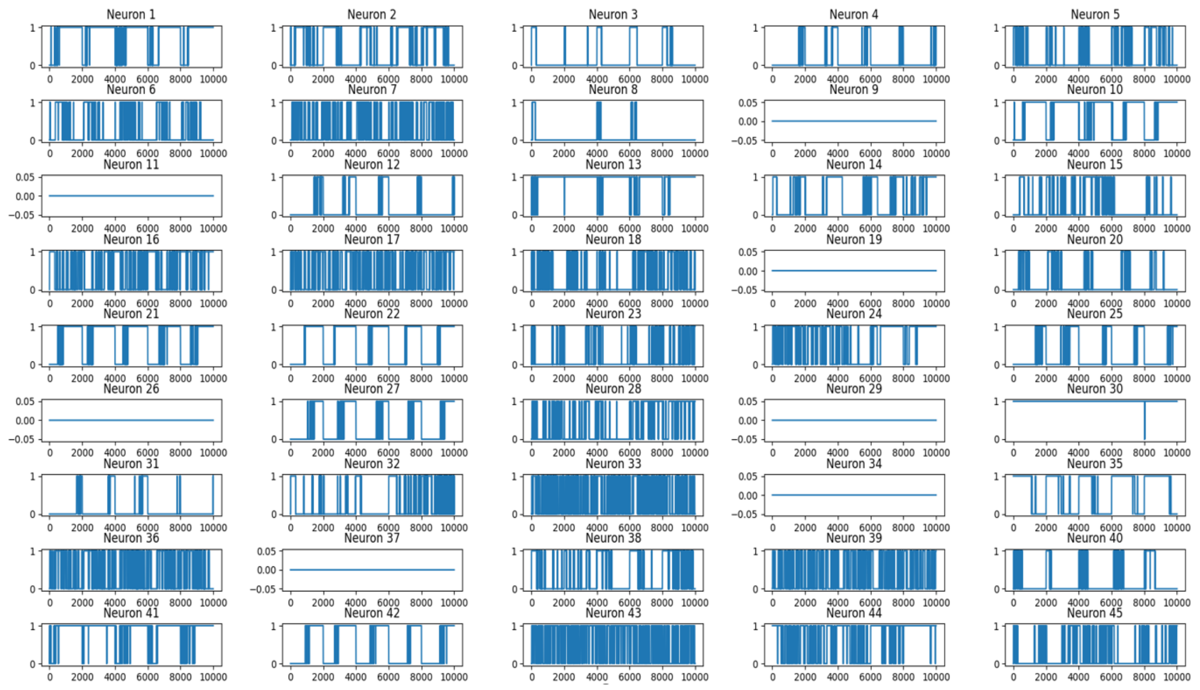


Figure A.13: The spiking activity of the neurons required to encode the parameters for angles and angular velocity when the masked model is used;

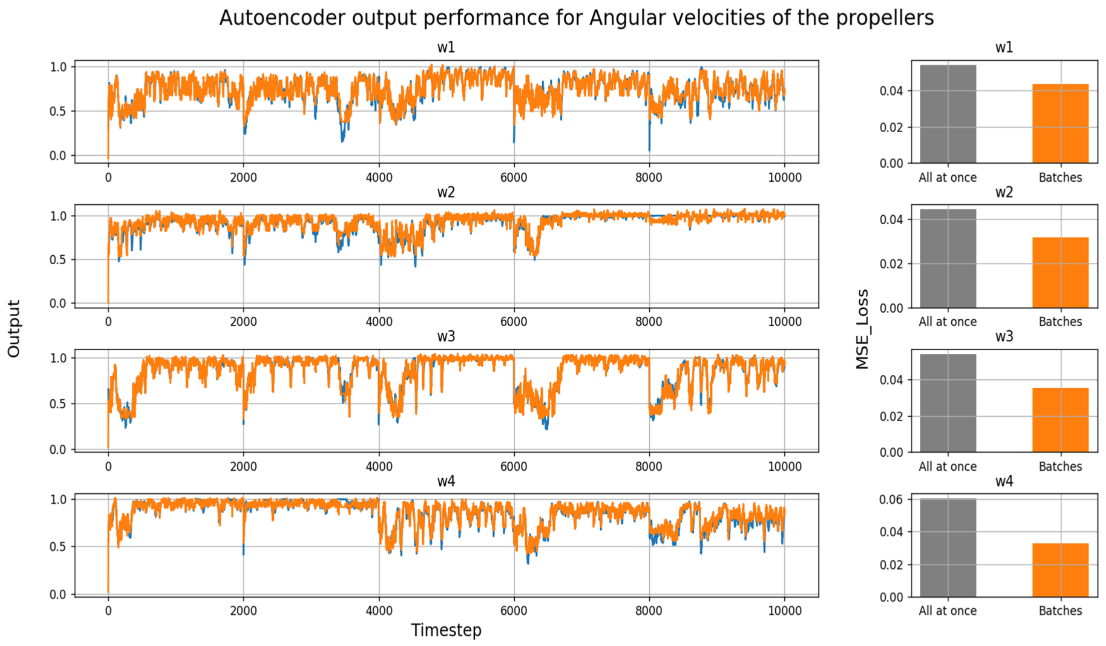


Figure A.14: The comparison between target output (blue) of the propeller angular velocity parameters and the encoder estimation (orange) on the left. On the right the performance comparison between the 2 encoder models where the parameters are trained as one batch versus when all parameters are encoded all together. The left plot uses the encoder model trained only on the propeller angular velocity parameters;

timesteps. These are again probably caused by the switch of samples in the dataset which leads to a sudden change in the input parameters values. However, this time the spikes are not caused by a special neuron which decides to spike, when analysing the spiking activity in Figure A.17. Instead, the spike is caused by a slight delay between 2 neurons. In other words, a neuron is not synchronised and starts 1 timestep later compared to the others.

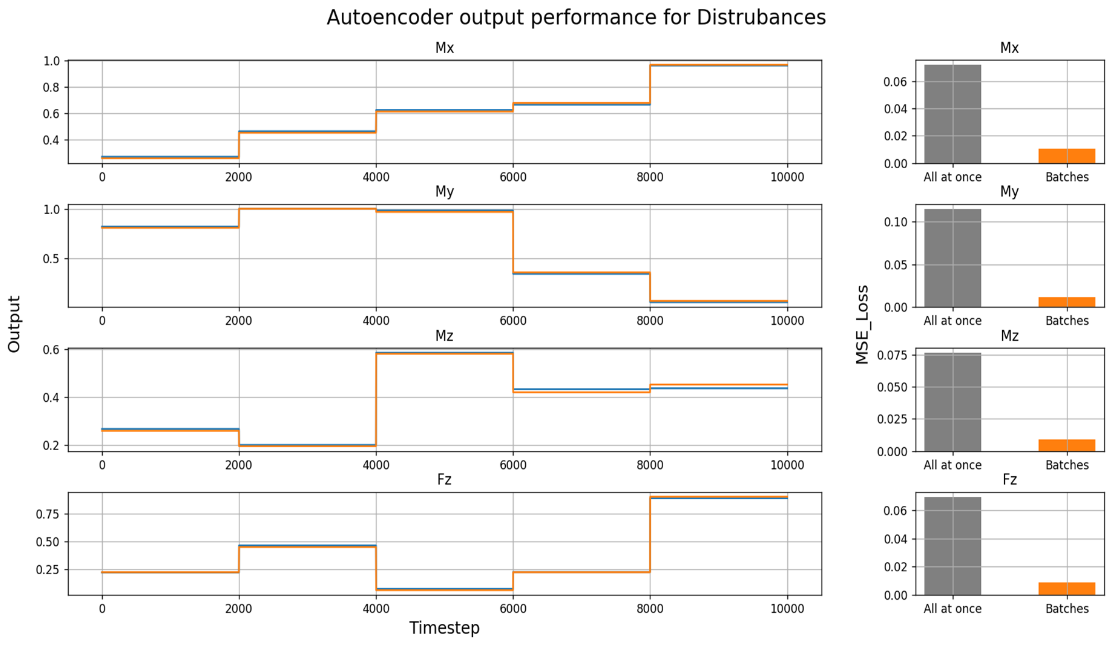


Figure A.15: The comparison between target output (blue) of the disturbance parameters and the encoder estimation (orange) on the left. On the right the performance comparison between the 2 encoder models where the parameters are trained as one batch versus when all parameters are encoded all together. The left plot uses the encoder model trained only on the disturbance parameters;

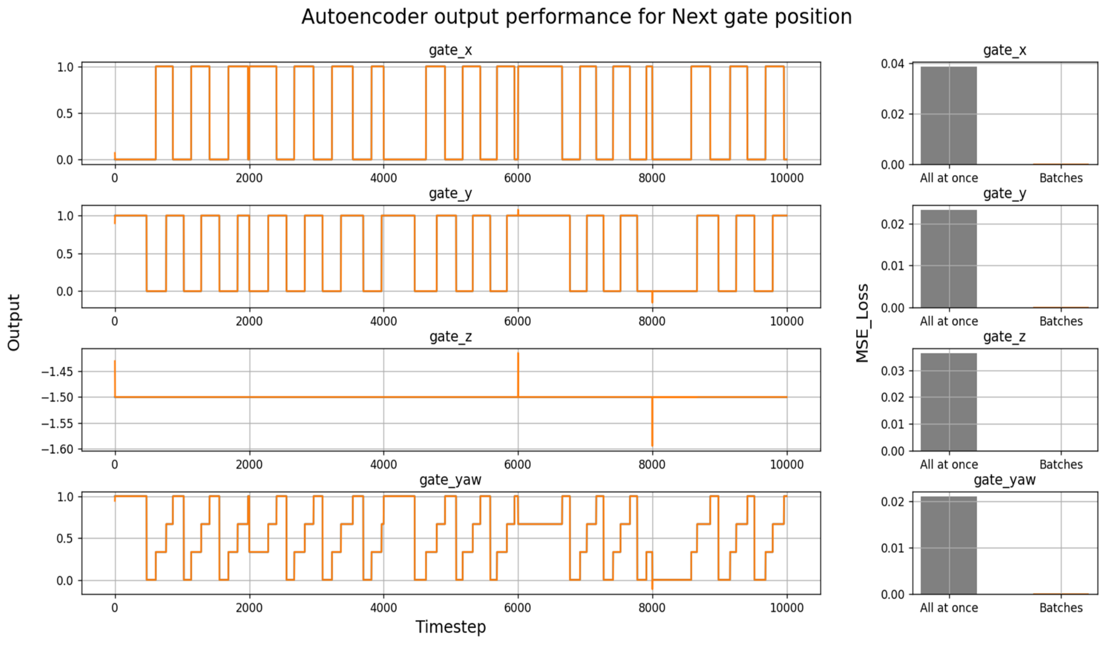


Figure A.16: The comparison between target output (blue) of the next gate position and yaw parameters and the encoder estimation (orange) on the left. On the right the performance comparison between the 2 encoder models where the parameters are trained as one batch versus when all parameters are encoded all together. The left plot uses the encoder model trained only on the next gate position and yaw parameters;

Last but not least, the output commands were encoded and the comparison can be seen in Figure A.18. As opposed to all the previous comparisons, training all parameters at once shows a

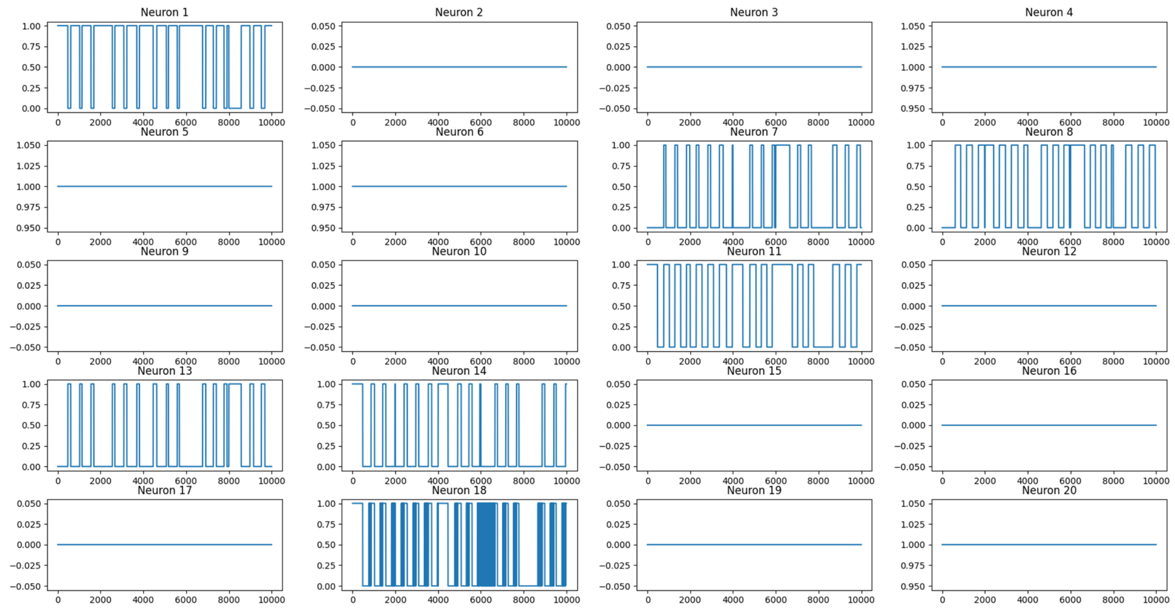


Figure A.17: The spiking activity of the neurons required to encode the parameters for next gate position and yaw when the masked model is used;

better performance than training into batches. Even though, for both encoders, the performance is pretty weak which can be caused by the very stochastic and noisy behaviour of the signals which is hard to be understood and followed over time. However, training all the parameters at once helps the encoding of the output commands. This can be caused by 2 explanations. Firstly, as was the case for the angular velocities of the propeller, there is little correlation between the 4 output commands which does not benefit training in batches. Secondly, as opposed to the angular velocities of the propeller case, it seems receiving information from the other parameters helps the performance of the encoder. This translates further that there is a correlation to be learned between all the parameters and the output commands. Thus this step showed initially that learning a controller is feasible as well as there might be no need for an encoding phase for the neuromorphic controller to learn the information.

With the performance analysed, the study continued by checking how the population encoder works. For this, for every group of parameter, the neuron activity was plotted along with the weights of the decoder to understand which neuron is responsible for every parameter and in which manner. The analysis will begin with the positions and velocities and can be visualised in Figure A.19 and Figure A.20 respectively. One important observation that can be observed directly as it was the case for the analysis of angles and angular velocities is that a bias neuron is being used for all parameters (neuron 17). This neuron acts as the mean calculator of all the parameters as it requires to bring the signal to its average through its high weight attached to it. Several neurons are not used at all (neurons 12, 15, 28, 33, 38, 41). Even though they might seem useless and discarding them could be a solution to save more neurons, they may have been trained for extreme cases which is also explained by the relatively large weights attached to them. Otherwise, it was observed that the z parameter has the smallest weights and this can be explained by the low variability in this parameter which requires little change.

Other general observations found from the encoding procedure of the previous batch of parameters that apply generally to all the encoding procedures include:

- It happens rarely that some neuron is tweaked to influence only one parameter but generally the weights are shared between parameters. In other words, it is very common that one neuron has non-zero weights for all parameters but the values varies greatly. This may

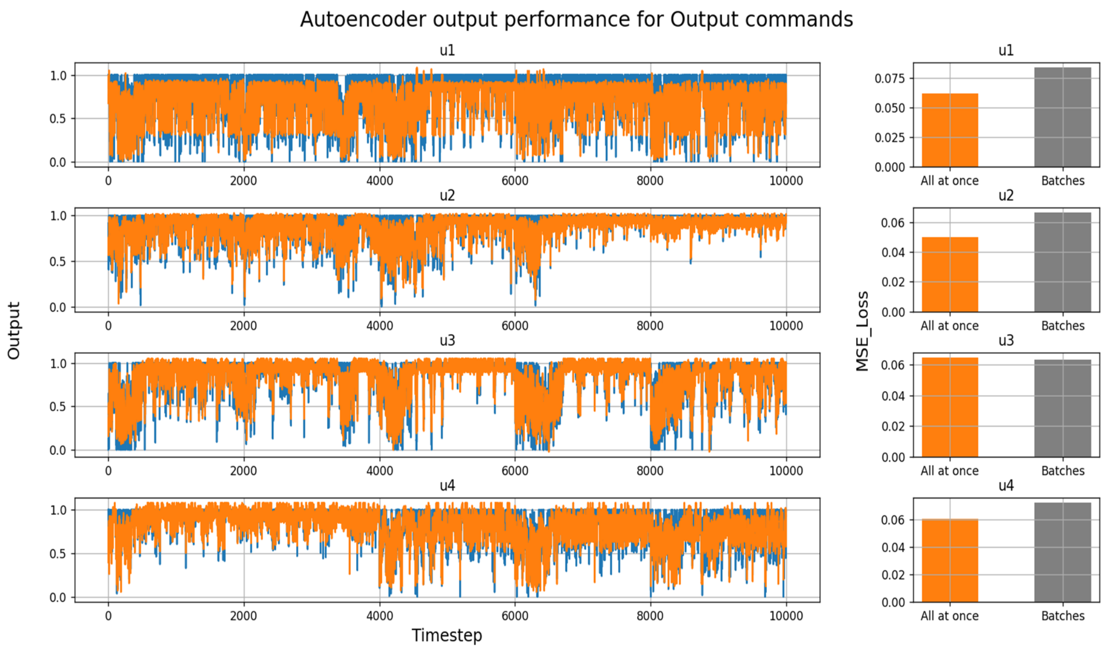


Figure A.18: The comparison between target output (blue) of the output parameters and the encoder estimation (orange) on the left. On the right the performance comparison between the 2 encoder models where the parameters are trained as one batch versus when all parameters are encoded all together. The left plot uses the encoder model trained on all the parameters;

show some correlation between parameters but could also show that the SNN will learn and encoder that is over complicated;

- There usually exist neurons that do not fire at all. Even though they may seem useless, they may be used when an extreme case happens and thus a sudden change is needed very fast as they generally have relatively higher weights;
- There exist usually one or more bias neurons that find the mean of all parameters and fire continuously;
- Generally, if the neurons are spiking rarely, a larger weight is given to the neuron. Thus that neuron receives a greater importance. The opposite is also true and the neurons with small weights attached usually control through their spiking behaviour the low amplitude ”noisy” behaviour specific to SNN.

The analysis continues with the angles and angular velocities with the focus upon their spiking activity in Figure A.21 and weights in Figure A.22. What this group of parameters has different is that, similar to the analysis done above, the bias neuron (which in this case is 41) suddenly stops spiking most probably due to a sudden switch in the dataset caused by the start of a new sample. An interesting behaviour is observed regarding how r is encoded. For this, neurons 32 and 33 are mostly being used. Thus neuron 33 is kept as an addition to the bias (so constantly spiking) and if a sudden drop is required, neuron 32 fires (with its negative weight) and if the drop is too big followed even by the neuron 33 stopping to fire. The same behaviour and usage of neurons could be observed for θ, p and ψ parameters. Otherwise, it seems neuron 24 seems to be used mainly for the encoding of ϕ

Next, the angular velocities of the propeller spiking activity and weights have been plotted in Figure A.23 and Figure A.24 respectively. Some interesting observation can be made based on the current plots. For example, this time, 3 spiking neurons are used to control the bias with the most important of them being neuron 2. However, it was found that these parameters

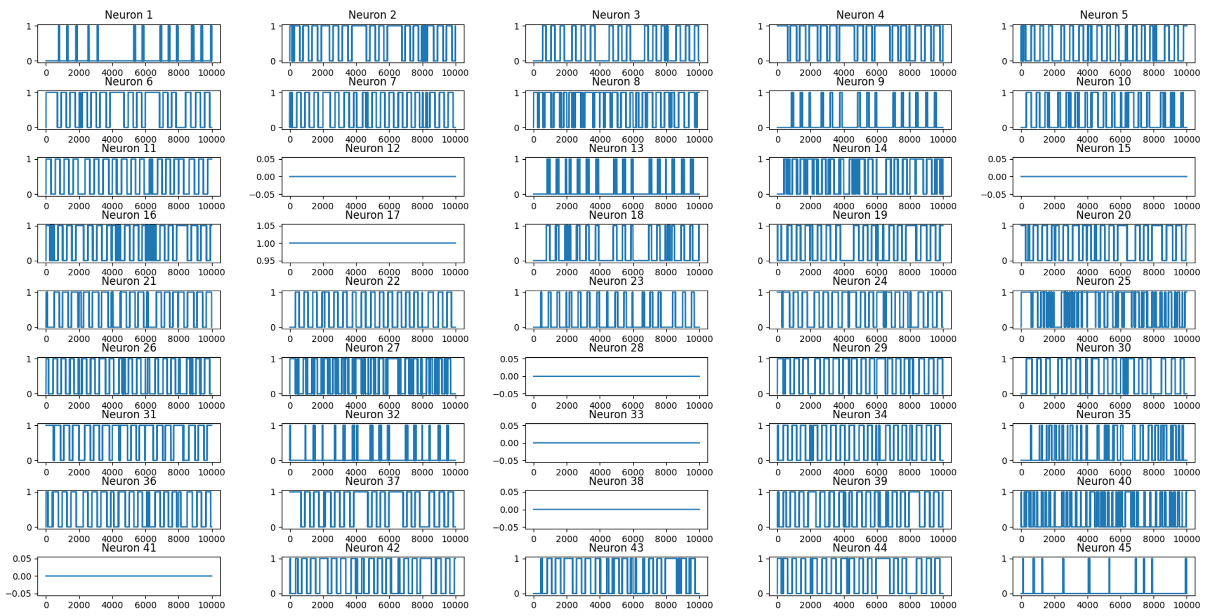


Figure A.19: The activity of all the spiking neurons found in the encoder specialised for positions and velocities;

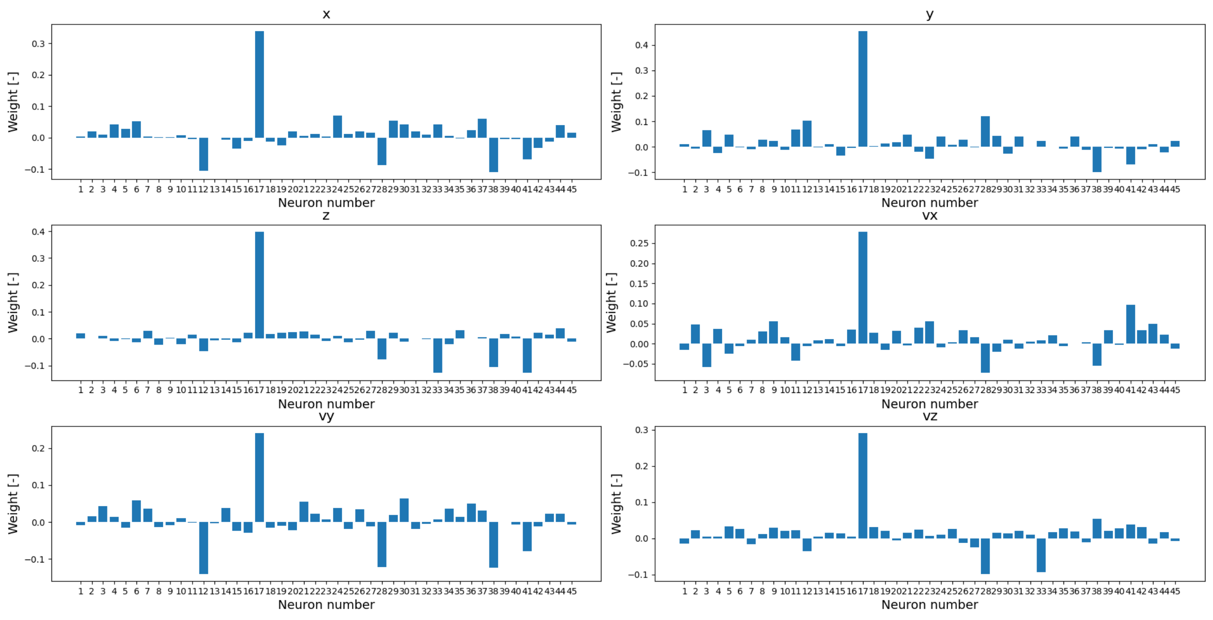


Figure A.20: The influence of every spiking neuron upon the positions and velocities predicted parameters of the encoder by looking at the corresponding weights;

often require sudden changes especially when a new sample begins which explains the relatively frequent stopping in the spiking behaviour observed for these bias neurons. Moreover, the average state of the normalised propeller rotational speed parameters is close to 1 which means that most of the neurons are focused on spiking. Otherwise, as the signal is much more irregular than for the other groups of parameters, there are multiple neurons which spike very aggressively and still have a large weight as rapid adaptation is absolutely required. Moreover, it was observed that 2 neurons are being used almost exclusively to encode w_3 and these are neurons 8 and 30 while they have little influence upon the other parameters.

An important group of parameters that benefited from the current analysis the most was the one of the gate positions and yaw which can be seen in both Figure A.25 and Figure A.26. The current plots show clearly how the encoding procedure functions as the task is much easier for

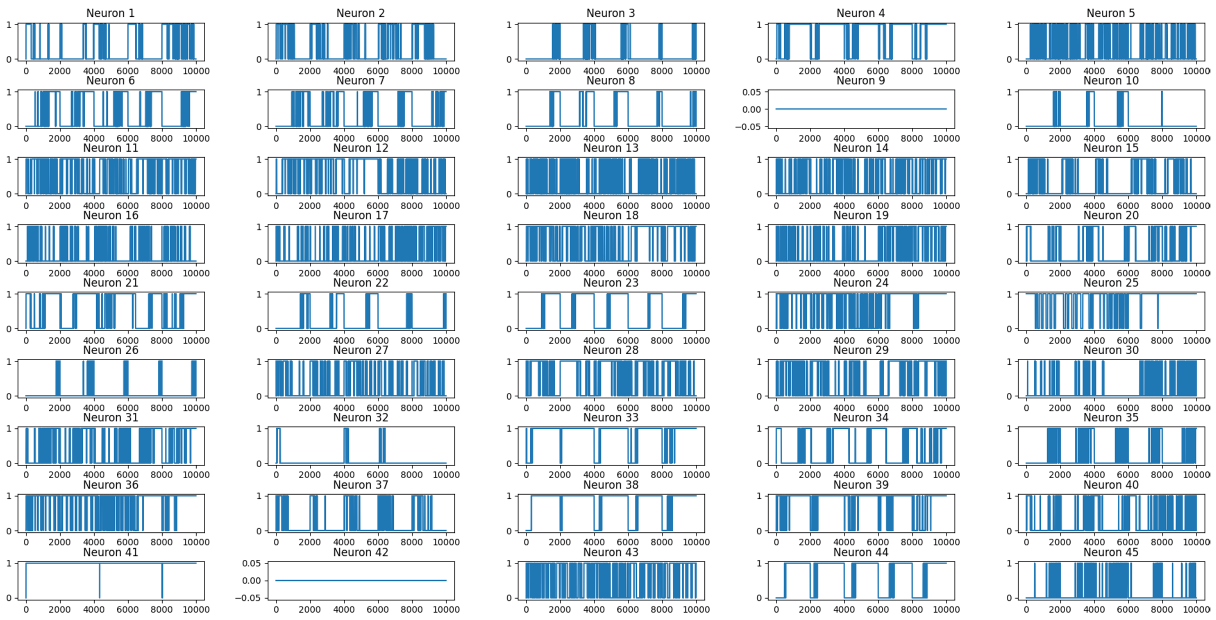


Figure A.21: The activity of all the spiking neurons found in the encoder specialised for angles and angular velocities;

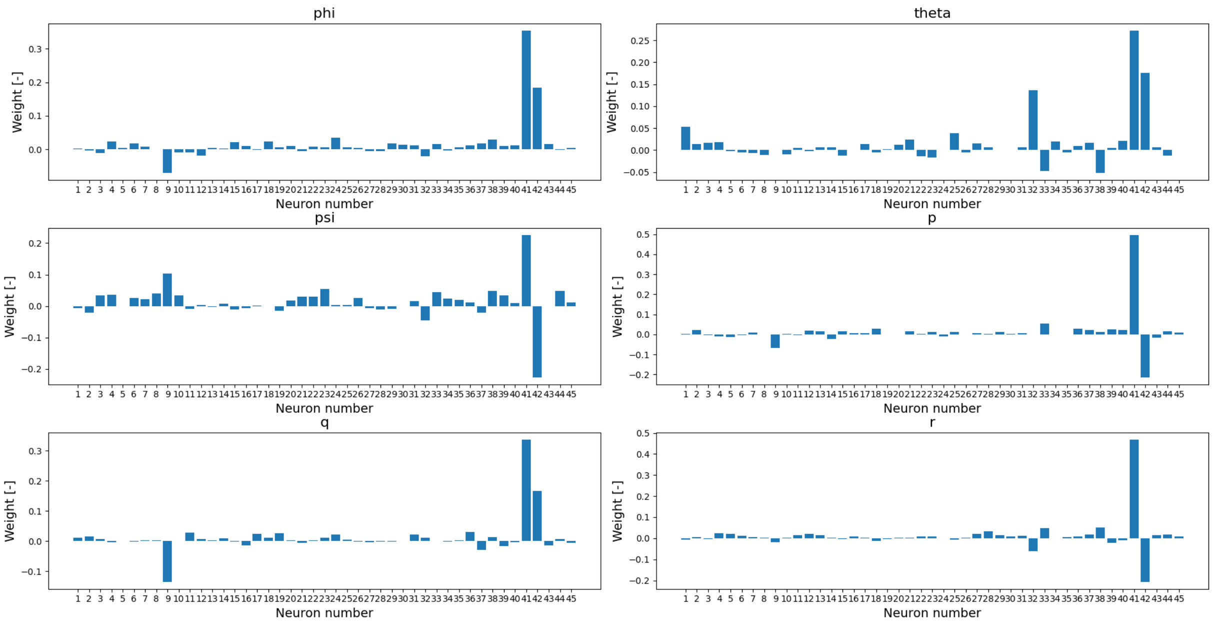


Figure A.22: The influence of every spiking neuron upon the angles and angular velocities predicted parameters of the encoder by looking at the corresponding weights;

this group of parameters. This time the x and y position of the gate can only take 2 values, z position is continuously constant and the value of yaw can only take 4 different values. Moreover, it seems that the current encoder needed much less neurons with 9 of them never firing. There are 4 neurons which regulate the bias (4, 5, 6 and 20). These act in contradictory manner such that they cancel their influence when used for x and y position as well as for the yaw. However, they are mostly used by the z position to keep the constant value. Otherwise neurons 8 and 11 perform most of the encoding as follows: neuron 8 controls the x position, neuron 11 controls the y position and both neurons control the yaw value. Otherwise, neurons 1 and 13 are not used as they counteract their influence. Moreover, it was observed that the delay between these 2 neurons leads to weird spiking behaviour observed previously for the next gate parameters encoding. And last but not the least, neurons 14 and 18 have low weights but spike very often

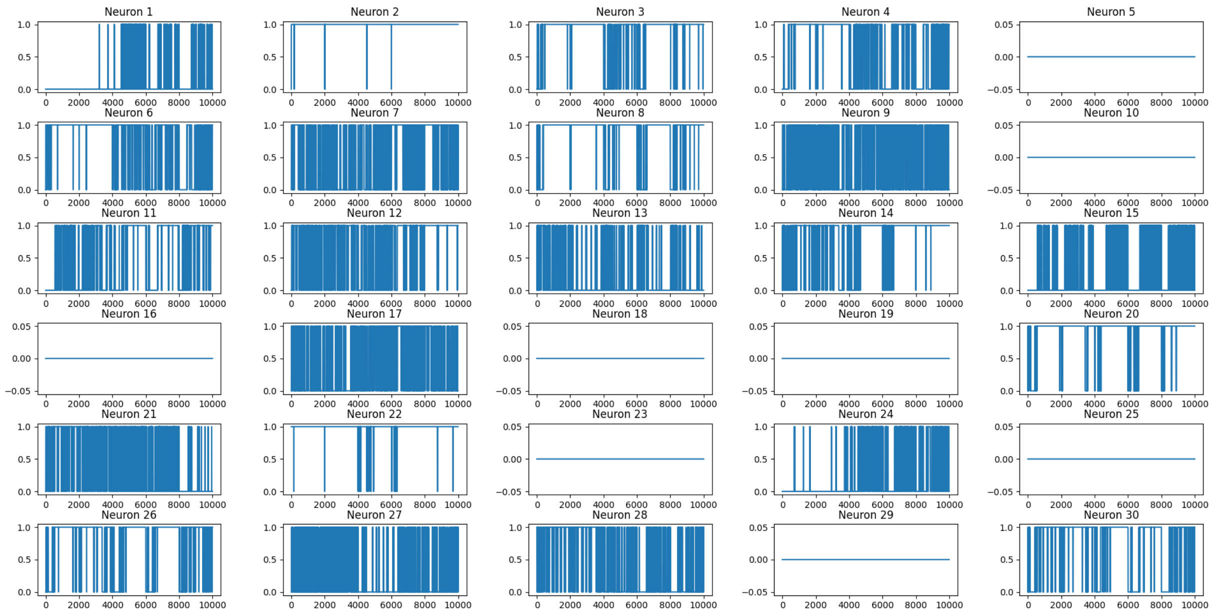


Figure A.23: The activity of all the spiking neurons found in the encoder specialised for the angular velocities of the propellers;

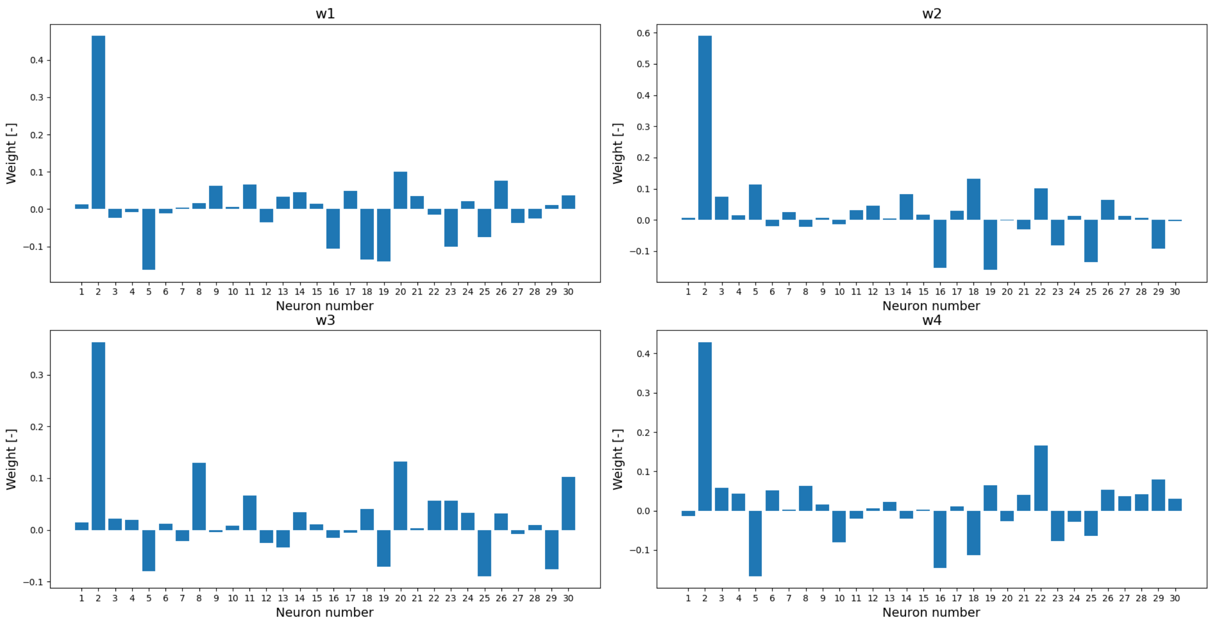


Figure A.24: The influence of every spiking neuron upon the angular velocities of the propellers predicted parameters of the encoder by looking at the corresponding weights;

having low influence upon the encoding procedure. To conclude, the encoding of the next gate parameters can be realised with only 3 neurons, one for the bias and 2 for the encoding of the 4 gate possibilities.

Last but not least, the encoding of the output commands can be visualised in Figure A.27 and Figure A.28. Even though the plots do not show clear information, some conclusion can still be drawn for them. Again, the encoding of output commands requires a special neuron for bias which stops firing if a sudden change is required. The vague spiking behaviour with most of the neurons spiking aggressively suggests that the input data is extremely irregular and ”noisy” which requires for very fast adaptation.

To conclude, it seems there is a sweet spot for the encoder where the ratio of spiking neurons

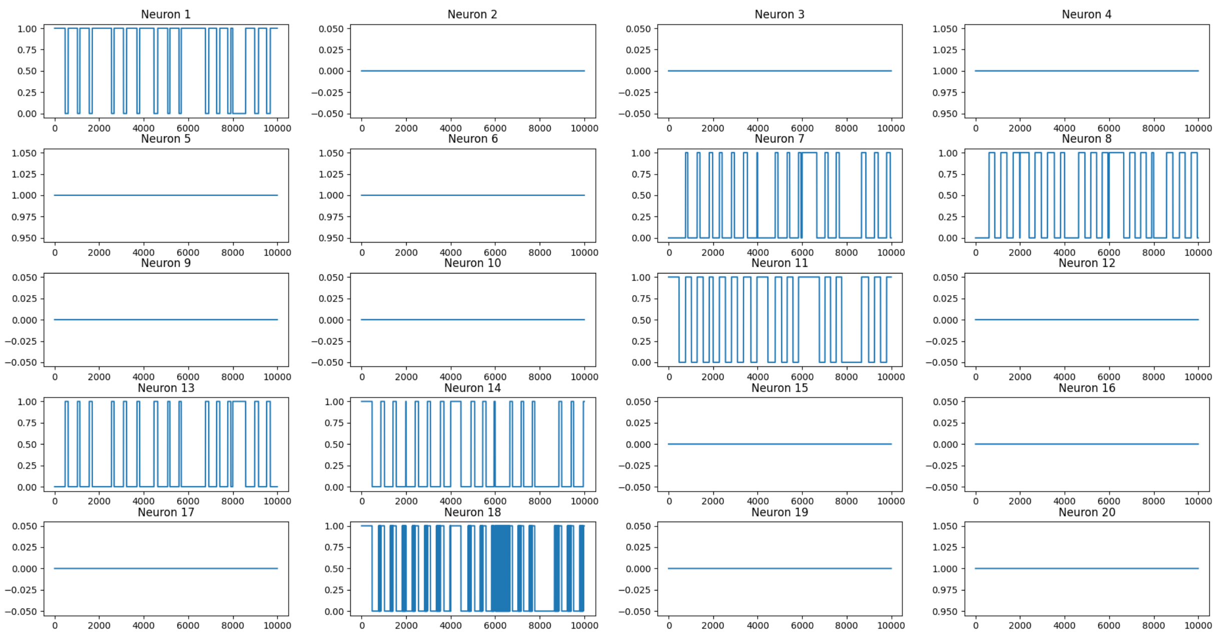


Figure A.25: The activity of all the spiking neurons found in the encoder specialised for the next gate position and yaw;

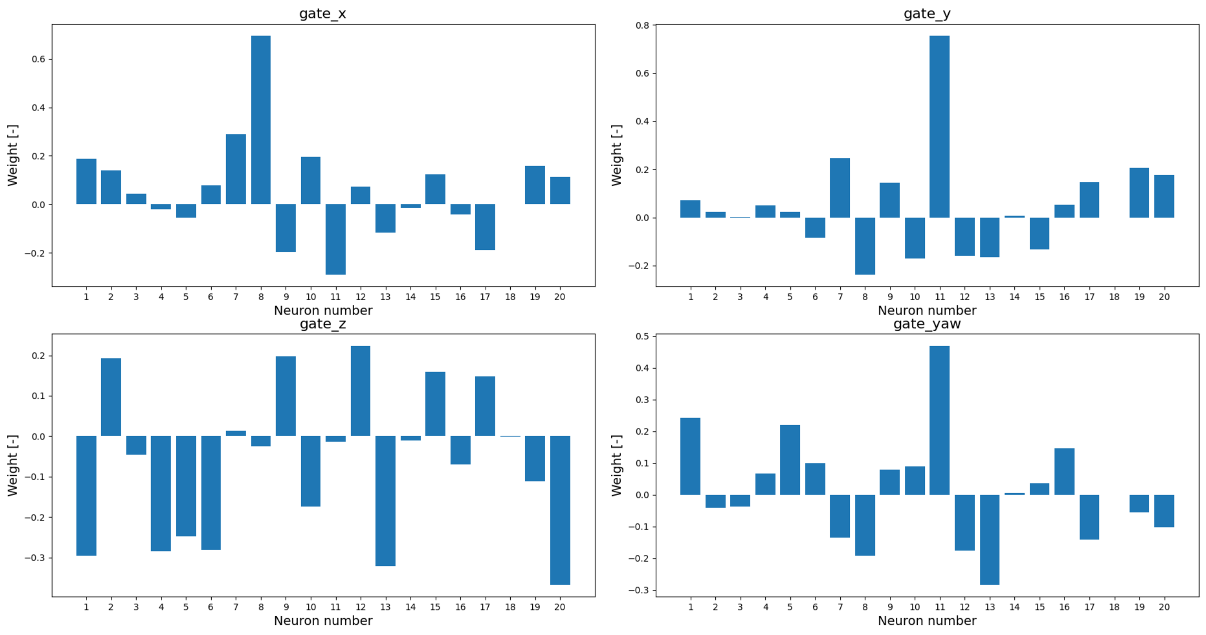


Figure A.26: The influence of every spiking neuron upon the next gate position and yaw predicted parameters of the encoder by looking at the corresponding weights;

per number of floating point neurons is around 7-8. However as specified in the main paper, the current approach cannot fly the drone in a real experiment due to the size of the spiking neural network which requires long processing times. In order to reduce the number of spiking neurons and thus speed up the running time required several techniques have been observed that include the following suggestions:

- If the parameters are discrete and their possible values are known as in the case of next gate parameters, encoding them separately with clear rules will improve definitely the performance;
- Binning or other population encoding techniques might be more beneficial than using

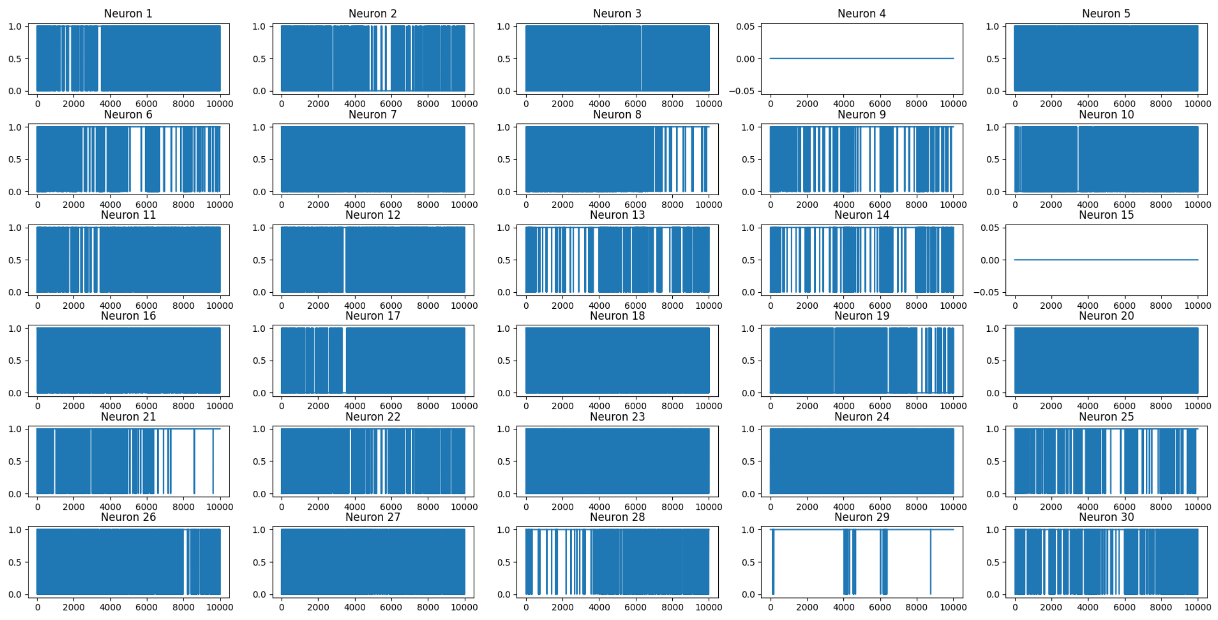


Figure A.27: The activity of all the spiking neurons found in the encoder specialised for the output commands;

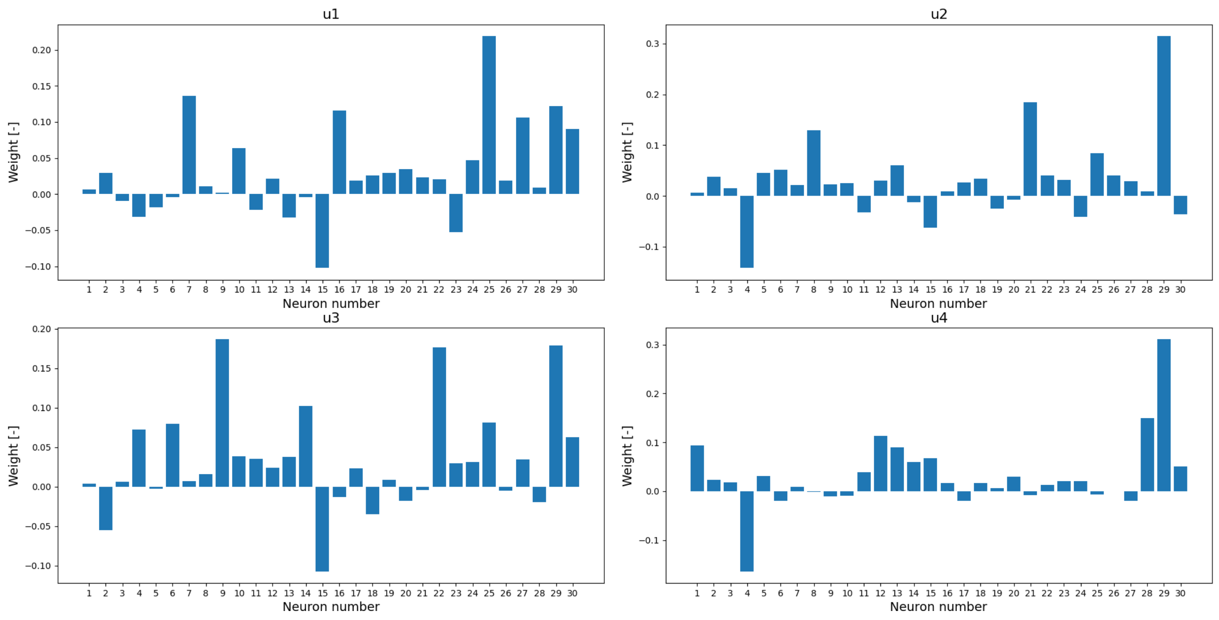


Figure A.28: The influence of every spiking neuron upon the output commands predicted parameters of the encoder by looking at the corresponding weights;

autoencoder as very good performance could be observed for the disturbance values;

- Grouping similar parameters together really helps the performance of the encoder. This could be studied in the future if the introduction of masks in the training procedure of the controllers might bring some new benefits;

A.4 Influence on the Controller

The current section will focus on the influence of the encoder upon the controller performance and the reasons why it was decided not to introduce it in its structure in the end. The analysis involved the creation of different controller structure including the training of both SNN and

MLP models. First 3 models considered were focused on the influence of the encoder and decoder upon the performance of the SNN controller. The structure of the models can be visualised in Figure A.29 - A.31. As can be seen, the structure with 3 layers with 500 neurons per each layer is preserved and was used for all the models considered in the analysis. The difference between the models consists in the usage of the encoder and decoder networks. Thus, one model does not use the encoder or decoder at all and is trained to perform the encoding as well as decoding, another model uses only the encoder and is trained to output the commands directly while the last model of the controller receives spikes as input and outputs spikes which should be decoded in the end with the previously trained models.

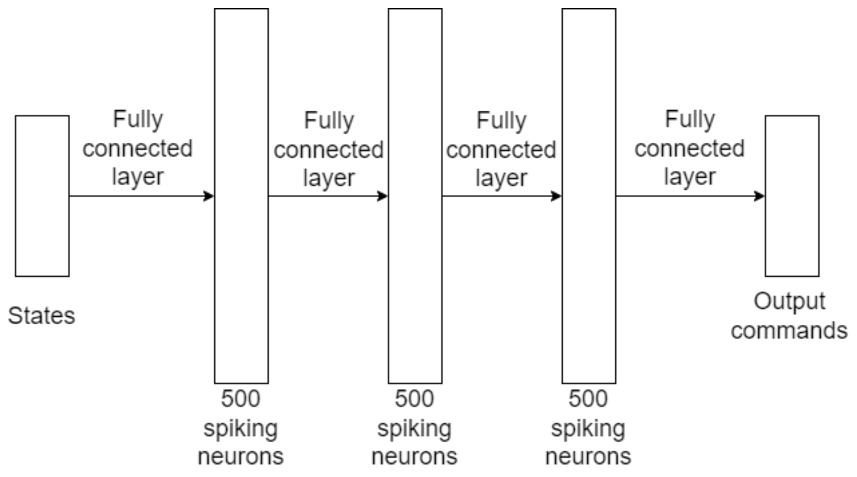


Figure A.29: The end to end controller structure;

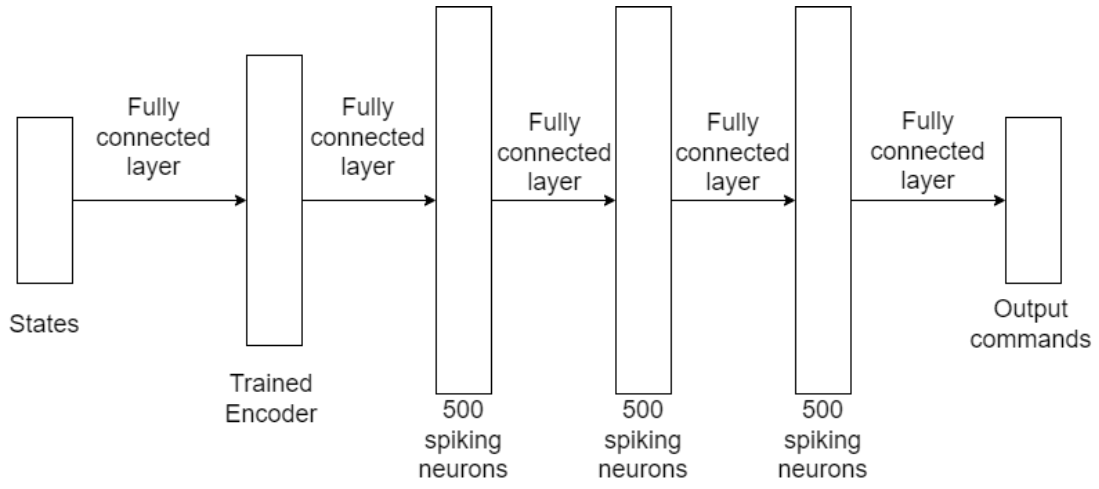


Figure A.30: The spiking to end controller structure that used the best performing encoder to transform input values to spikes;

The performance of the models can be visualised in Table A.2 with the mean absolute error values for the output command of each propeller. The training has been done on the dataset generated with the reinforcement learning approach. As can be seen, the table does not contain the controller model trained from spikes to spikes as this is inherently impossible. The reason for this is that learning a controller to output spikes is very risky. If one spiking neurons fires a spike at the wrong time, the output of the whole network after being passed through the decoder might be very different. This comes as the spikes of the controller as well as the decoder behaviour are not continuous and function in a discrete manner. However, starting from spikes instead should not cause problems. When looking at the comparison of the end to end model to the spiking to end model, it can be seen that the end to end model performs marginally better

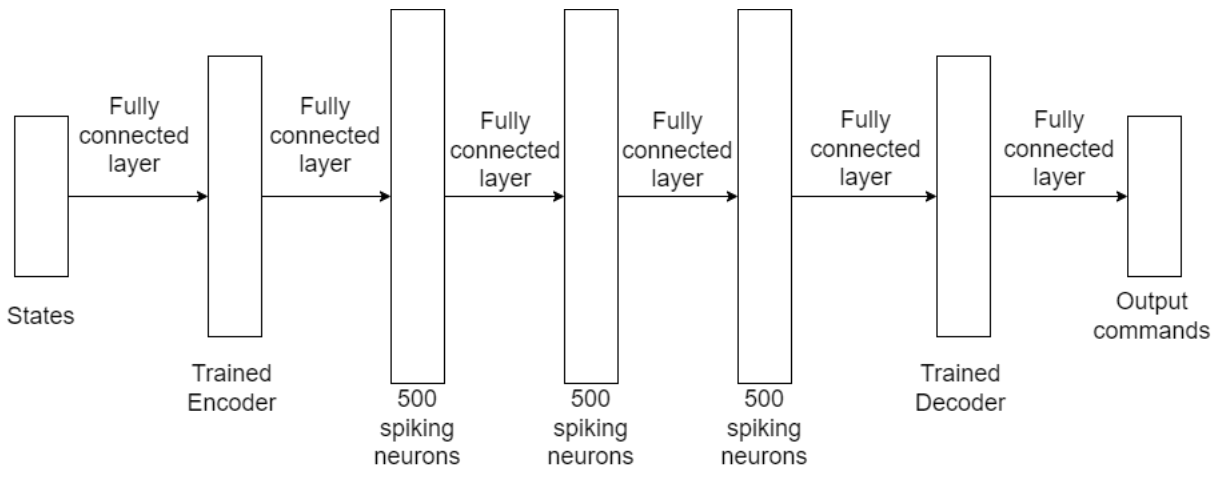


Figure A.31: The spiking to spiking controller structure that used the best performing encoder and decoder to transform input and output values to spikes;

for all the propellers but both models manage to learn a controller. The reason why the model trained without an encoder performs better is rooted in the fact that the encoder will bring an additional error which has to be accounted by the controller. If the encoding procedure would be flawless and consistent with almost no error being caused, the spiking to end controller might be able to perform better as the controller model is not concerned with encoding anymore but this should be left as future recommendation for study.

Table A.2: Absolute error of different methods of building a controller for every output command

Model	**u1**	**u2**	**u3**	**u4**
End to End	0.1534	0.0695	0.0930	0.0981
Spiking to End	0.1589	0.07107	0.1007	0.1064
Controller p&ϕ	0.2126	0.1078	0.1816	0.1713
MLP End to End	0.1501	0.0636	0.0901	0.0915
RNN	0.2241	0.2517	0.2895	0.3334

During the initial analysis of the controller structure, several other controller models have been trained for a better understanding of the training process. Firstly, training an SNN controller with less input parameters was analysed. For this, a pair of related parameters was chosen that is not periodic over time (such as the x or y position) and provides important information about the state of the flight (as opposed to z position or the ψ angle). Thus, it was decided to use only the roll angle ϕ and its derivative p to build a controller. The results are as expected as very few information is being passed to the controller. Even though, the output is discrete and it learns to follow mostly the average of the signal without being capable to model the aggressive behaviour of the signal with sudden upward and downward values. This behaviour can be visualised in Figure A.32.

Following this, some artificial neural networks have been trained to learn the controller as well. Thus 2 such models were created, one that uses a multi-layer perceptron (MLP) structure and learns the controller end to end and a second one that uses recurrent neural network (RNN) also to learn the controller end to end. Comparing their performance with the other models, it can be seen that the MLP achieves the best performance on learning the controller but the improvement compared to SNN is not that evident. Instead, using RNN to learn the controller shows how recursion causes poor performance. This can be explained as the output command is very spiky and the output commands barely depend on previous timesteps. This also explains why the SNN chooses an approach closer to the MLP approach (so little recursion) with very low leak values and thus very little dependency on previous timesteps. This can also be visualised by comparing the predicted controller output to the target output of the 3 models (the RNN, the MLP and the SNN). The RNN output can be seen in Figure A.34, the MLP output in Figure A.33 and the SNN can be visualised in the main paper work but it is very similar to the MLP output and thus it was decided not to reproduce it anymore. From the 2 plots some conclusions can be drawn. First of all, the dataset is so aggressive and spiky that not even with an MLP can be accurately reproduced and thus the performance is still very poor. For this reason, it was decided to make the RL controller output a smoother dataset that can actually be learned as explained in the main paper work. Secondly, the recursion makes the controller run smoother without sudden jumps across the dataset and just being concerned of following the average value of the dataset. Thus it is expected that the controller trained with RNN to achieve great performance at following the average trend as it learns from present and past information.

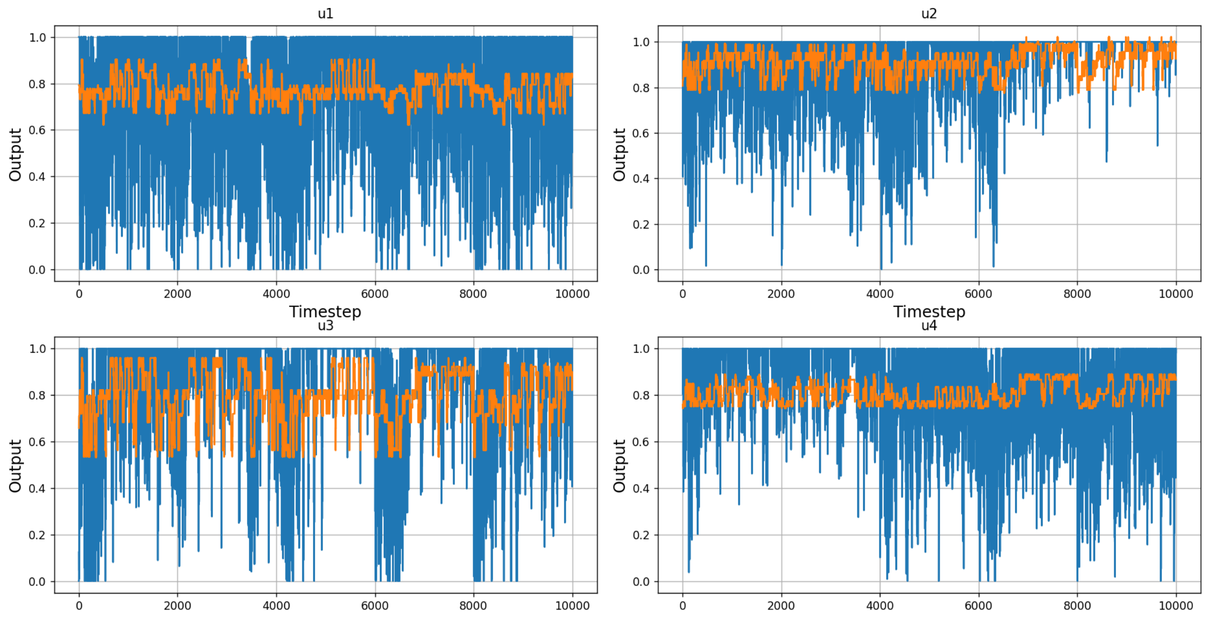


Figure A.32: The performance of the controller that had as inputs only the p and ϕ parameters;

A.5 Hover to Hover with Reinforcement Learning

This section will present how various datasets were generated with the reinforcement learning approach. Even though the title mentions about a hover to hover dataset, the current section will also present how the dataset generated with reinforcement learning was made smoother. The outcomes of the process can be seen in Figure A.35 where the output of the SNN can be seen. It is clear now that when a smoother dataset, that can be followed more accurately, is passed to the controller, its performance gets definitely improved. In order to make the dataset smoother more constraints were imposed in the reward function. Until this moment, the only constraint present was to minimise the distance to the gate as fast as possible. However, to make the dataset smoother, 2 more constraints were added:

- The derivative of the command output should be reduced such that sudden jumps in the output are avoided. In order to combine it with the existing reward system it was found that a ratio factor of 0.00002 should be applied with respect to the existing distance constraint;
- Following a similar approach as the energy optimal control dataset, the command output should be reduced such that the drone flies directly to the gate with the lowest resources

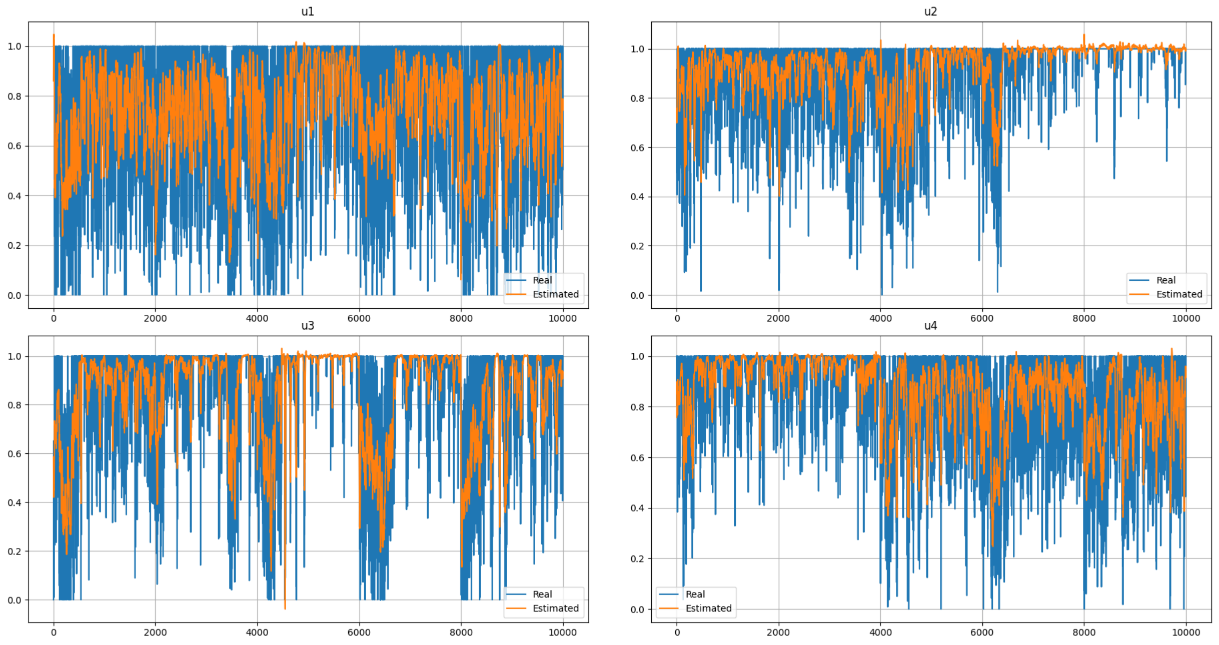


Figure A.33: The performance of the MLP controller trained end-to-end;

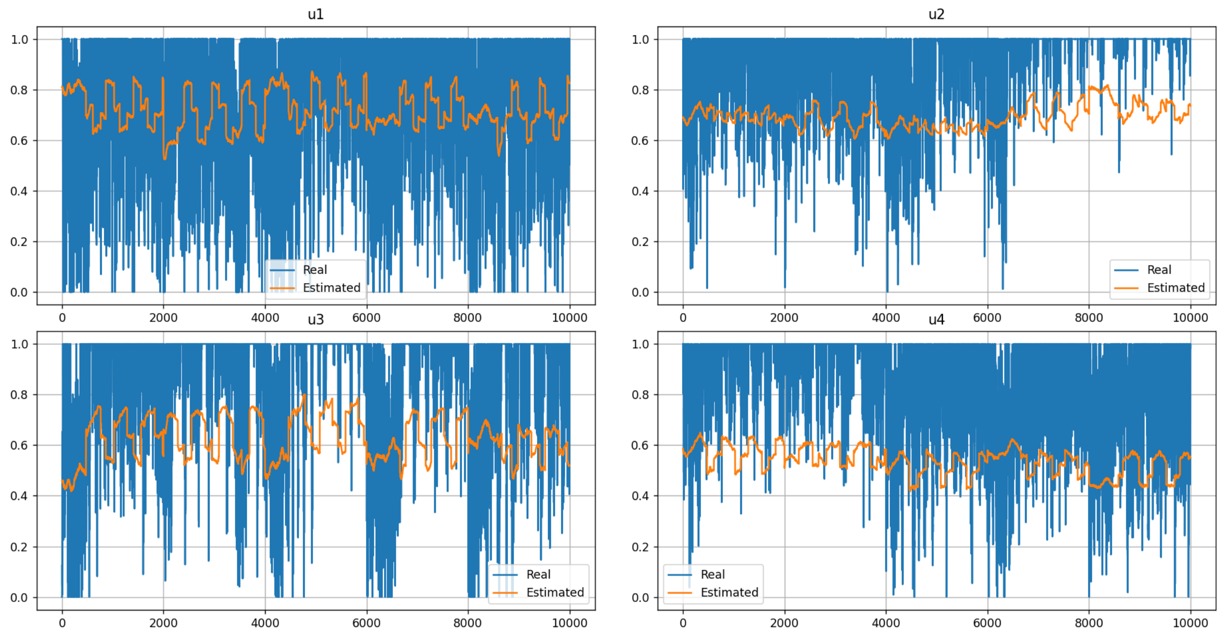


Figure A.34: The performance of the RNN controller;

possible. In order to combine it with the existing reward system it was found that a ratio factor of 0.004 should be applied with respect to the existing distance constraint;

A similar approach was applied for generating a hover to hover dataset with reinforcement learning as done with the energy optimal control problem dataset. Thus a model was learned to fly optimally from one position to a hover state using reinforcement learning. With the resulting model, a dataset containing 10000 samples of 800 timesteps each (equivalent to 8 seconds per sample) was generated. This was passed both to an SNN as well as to an MLP to be learned. Figure A.36 shows the performance of the SNN controller on a testing dataset formed of 10 samples while Figure A.37 shows the performance of the MLP controller in the simulator. Unfortunately, the SNN model could not be flown in the simulator as the controller crashed immediately. As can be seen, the SNN manages to follow the target signal relatively close but

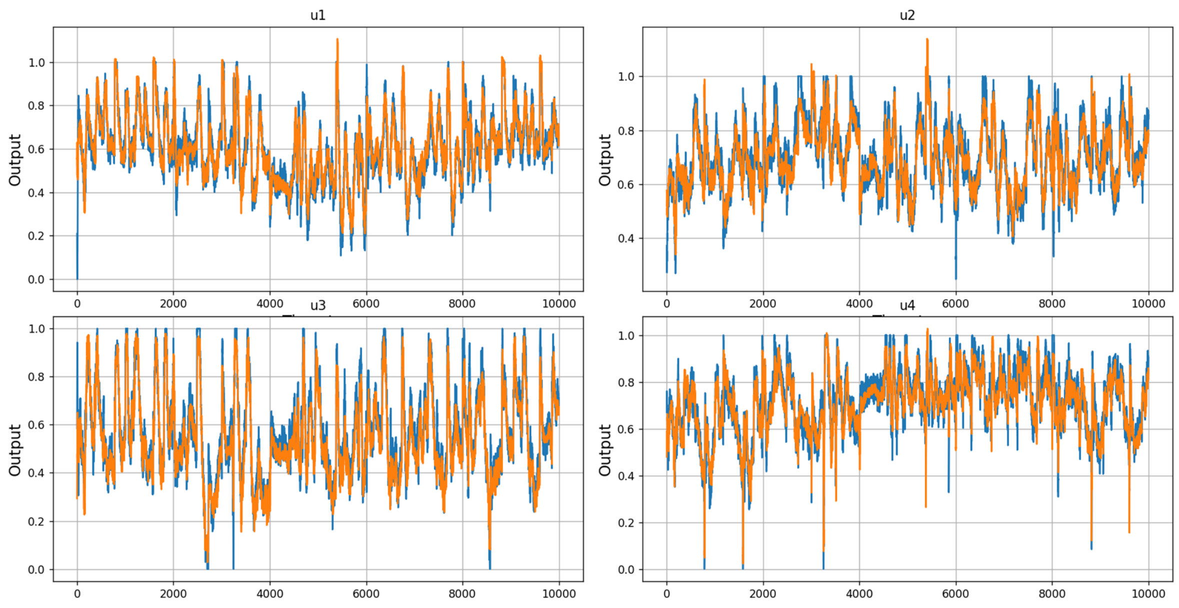


Figure A.35: The trained SNN model performance on the smooth dataset generated with reinforcement learning;

there seems to be a slight bias in the output command as the value is slightly higher than it should be. The reason for the bias was not found and it requires further research but overall the performance seems very good and able to perform. The bias might also be the reason behind the SNN failure in the simulator. On the other hand, the MLP model manages to learn the controller even better as it can fly around the racing track. As observed and analysed through the current paper, the SNN should be capable of getting close to the MLP performance. This leads to a future research recommendation as it should be possible to train an SNN controller with a dataset generated with reinforcement learning.

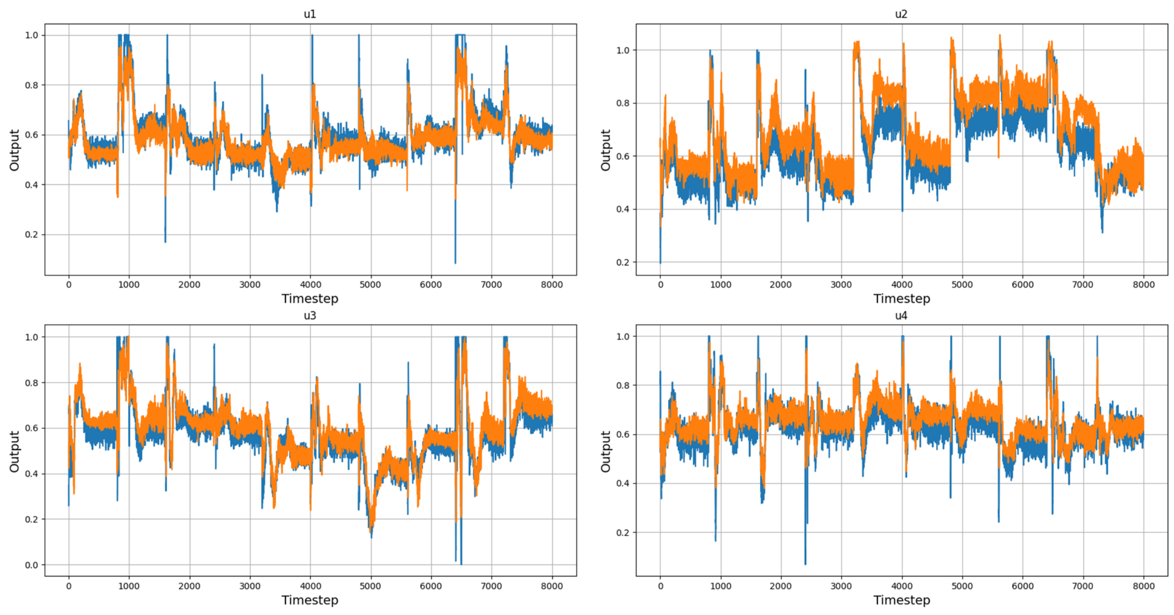


Figure A.36: The trained SNN model performance after being trained on a hover to hover dataset that was generated with the reinforcement learning approach;

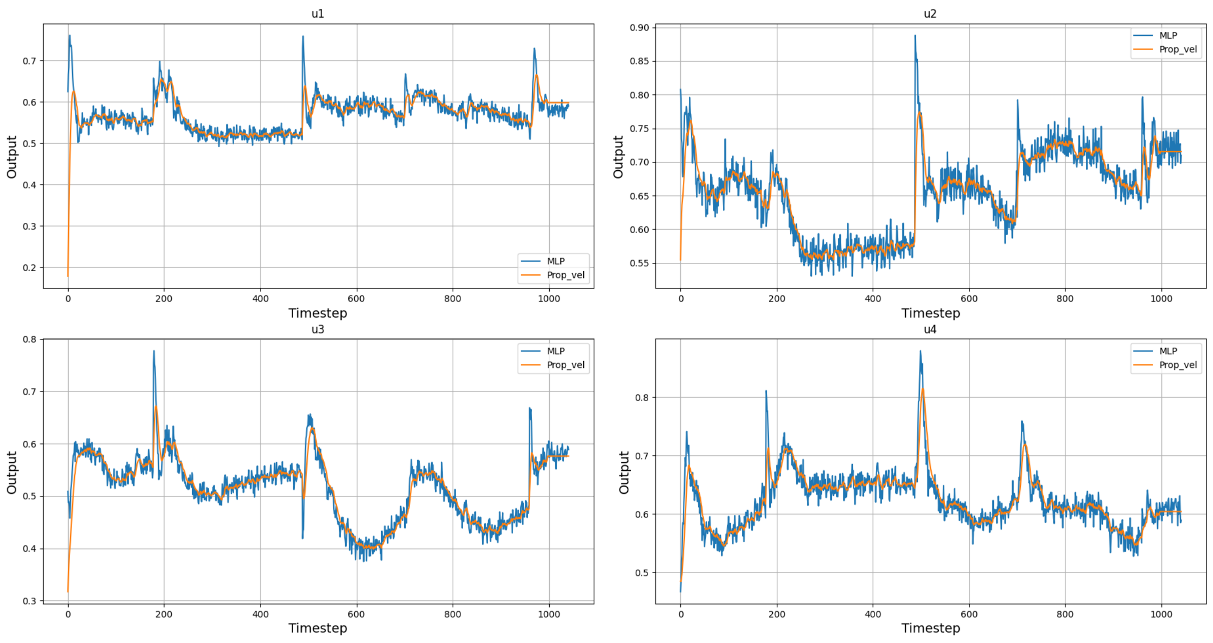


Figure A.37: The trained MLP model performance in the flight simulator around the racing track after being trained on a hover to hover dataset that was generated with the reinforcement learning approach;

A.6 Weight Decaying

Even though this section is not related to the study of the encoder, it was still included in the current chapter as the analysis reveals important conclusions with regard to the learning performance of spiking neural networks not included in the main paper work. The weight decaying was applied to the SNN controller as an attempt to reduce the heavy jumps present in the output commands. In order to apply it, L2 normalisation technique has been used. This tries to minimise the following loss function:

$$\text{MSE} = \sum_{i=1}^{4}(u_i - \bar{u_i})^2 + \lambda \sum_{j=1}^{N_{param}} \beta_j \tag{A.1}$$

where the first part of the equation is simply the mean squared error value of the network to be minimised. The second part of the equation includes the regularisation term which implies that all the N_{param} parameters β_j should be minimised. The term λ is a factor which controls how strong the regularisation should be which in the case of the current analysis was initialised to 0.001.

In the end, using regularisation did not achieve the intended outcome making the SNN controller less stable as can be observed in Figure A.38. The MSE loss of the model with weight decay increased to 2.256e-2 compared to without decay where the MSE loss is only 5.387e-3 on the validation dataset. This decrease in performance can be attributed to the weight decay constraints which just imposes more limits on finding the best performance as it is concerned with minimising the values of all the parameters throughout the model. Even though a weaker performance is achieved during training, when comparing the lap times achieved by the 2 SNN models, better performance can be generally attributed to the controller trained using weight decay. So, the current analysis shows that adding normalisation to the loss function improves marginally the lap times but leads to less stable laps with the several outliers for other laps times and a high standard deviation for first lap times. The reason for this might be that with less constraints, better training is achieved and thus a more stable controller as well. The faster lap times might be caused by the fact that the controller with weight decay learned to fly closer to the inside of the racing circuit. This generally leads to faster lap times but also brings a riskier approach with a higher chance of missing the gates explaining the outlier lap times as well.

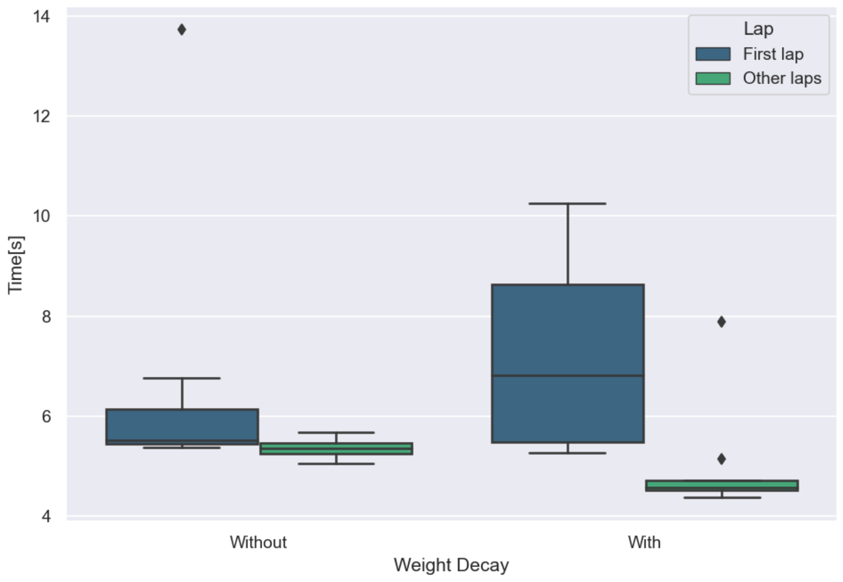


Figure A.38: The influence of the weight decay on the lap robustness and aggressiveness of the SNN model trained on the hover to gate dataset with 500 neurons per layer

B

Coding Contributions

The aim of this chapter is to present the extra work performed while developing the programming code required for the current study. The focus of the current chapter will be on explaining what has been adapted and what innovations have been brought to achieve the final goal of the research. The work done during the current thesis can be found on a public repository[1]. Thus, the chapter has been split in 2 sections, each responsible for a programming part of the current project. Firstly, section B.1 was focused on the simulator part of the code and will explain how the SNN model was built and trained using Python programming language. Secondly, section B.2 will detail how the best SNN models were adapted to C programming language such that they can be implemented in the experiment on the Bebop 1 drone.

B.1 Spiking library

For training an SNN, an existing repository in Python offered by TU Delft was used.[2] Through the usage of PyTorch library, the repository allowed for the training of an SNN using neuromorphic approach with the creation of CuBa LIF neuron modules and surrogate gradient descent backpropagation methods. The existing repository was specialised on image recognition which required the need of restructuring to be adapted to the current problem of guidance and control. Thus this included the normalisation of data between 0 and 1, the use of MSE Loss for calculating the error and optimising the model as well as the inclusion of a learning rate scheduler which decreases the use of learning rate when the learning plateau is reached. Moreover, the division between training and validation dataset is still used and the selection of the optimal model is done based on the best performing model on validation dataset. For this reason, the number of epochs have been increased and the model is left to converge slowly to the optimal result. However, some choices were still left to the existing code such as the ADAM optimiser, or the arctangent as the function used for surrogate gradient descent.

With all these in mind, some changes had to be brought to the code as well. For example, as mentioned above, the existing code was specialised on image recognition which required the usage of convolutional synapses. However, for learning a controller with SNN using supervised learning, a new module was required that would combine the existent CuBa LIF neurons module with the linear synapse module. Moreover, as explained in section A.3, the research required the analysis of the optimal number of neurons by grouping similar parameters together. To achieve this during the training procedure, a mask was imposed upon the weight matrices to allow for the grouping of parameters in the correct manner. This led to the creation of a new type of synapse module which was added to the existing framework named `MaskedLazyLinear` which can be visualised below:

[1]https://github.com/TudorAvarvarei/Code
[2]https://github.com/tudelft/spiking

```python
class MaskedLazyLinear(nn.LazyLinear):
    cls_to_become = None
    def __init__(self, out_features, bias, mask=None):
        super().__init__(out_features, bias)  # Input size is set to 0 initially
        self.mask = nn.Parameter(mask, requires_grad=False)

    def forward(self, input):
        # Apply the mask to the weight matrix
        masked_weight = self.mask * self.weight
        input = nn.functional.linear(input, masked_weight, self.bias)
        return input, [input]

    def initialize_parameters(self, input, *__):  # discard state argument
        super().initialize_parameters(input=input)  # add delay dimension
```

The inclusion of this module required the change of several functions. This changes included the creation of a new layer module that combined the usage of the CuBa LIF neuron module to the `MaskedLazyLinear` synapse module as well as the change of the function that reads the SNN model which required a new input, namely the mask matrix. Using the current library required further changes in other modules such as the simulator developed by Ferede et al.[22, 23] for the supervised learning and reinforcement learning code. However, the changes were generally local and spread across the code so a detail of the procedure is beyond the scope of the current section.

B.2 Switching to C Code

In the current section a presentation of the work required to transform the existing PyTorch SNN model to C programming language will be done. This requirement is imposed by the experiment and the Bebop 1 drone which can only process the information in real time through the C programming language. In order to do this, the existing repository created for training an MLP with supervised learning by Ferede et al.[22][3] will be used. However, this repository was crafted for traditional artificial neural networks and cannot process directly SNNs. In order to solve this issue, inspired by the work present in another repository[4], the current code was adapted to process SNNs in C programming language. In order to adapt the existing C code spiking repository, the following modules were changed as follows. The connection module that forwards the information between 2 consecutive layers and saves the weights of the synapses is given by the following code:

```c
#include "Connection.h"
#include <stdio.h>
#include <stdlib.h>

// Build connection
Connection build_connection(int const post, int const pre) {
  // Connection struct
  Connection c;

  // Set shape
  c.post = post;
  c.pre = pre;

  // Allocate memory for weight array
  c.w = calloc(post * pre, sizeof(*c.w));
  return c;
}
// Init connection
```

[3]https://github.com/tudelft/optimal_quad_control_SL
[4]https://github.com/tudelft/tinysnn

```c
void init_connection(Connection *c) {
  // Loop over weights
  for (int i = 0; i < c->post; i++) {
    for (int j = 0; j < c->pre; j++) {
      c->w[i * c->pre + j] = rand() / (float)RAND_MAX;
    }
  }
}
// Reset connection
// Doesn't actually do anything, just for consistency
void reset_connection(Connection *c) {}

// Load parameters for connection (weights) from a header file
// (using the ConnectionConf struct)
void load_connection_from_header(Connection *c, ConnectionConf const *conf) {
  // Check if same shape
  if ((c->pre != conf->pre) || (c->post != conf->post)) {
    printf("Connection has a different shape than specified in the "
           "ConnectionConf!\n");
    exit(1);
  }
  // Loop over weights
  for (int i=0; i<c->post*c->pre; i++){
    c->w[i] = conf->w[i];
  }
}
// Free allocated memory for connection
void free_connection(Connection *c) {
  // Only one call, so only one free (as opposed to other methods for 2D arrays)
  free(c->w);
}

// Forward
// Spikes as floats to deal with real-valued inputs
void forward_connection_float(Connection *c, float out[], float const in[]) {
  // Loop over weights and multiply with spikes
  for (int i = 0; i < c->post; i++) {
    for (int j = 0; j < c->pre; j++) {
      out[i] += c->w[i*c->pre + j] * in[j];
    }
  }
}
// Forward
// Spikes as ints to deal with real-valued inputs
void forward_connection_int(Connection *c, float out[], int const in[]) {
  // Loop over weights and multiply with spikes
  for (int i = 0; i < c->post; i++) {
    for (int j = 0; j < c->pre; j++) {
      out[i] += c->w[i*c->pre + j] * in[j];
    }
  }
}
```

Similarly, the neuron module responsible for saving the parameters of all the neurons in all the layers such as leak and threshold values but also the voltage and current potentials as well will be detailed below. This module required the highest amount of change as the existing neuron model was the LIF neuron while in the Python spiking library, the CuBa LIF neuron module was used. Thus, both the parameters used as well as the neuron dynamics had to be adapted. For a better understanding, the created module will be shown below:

```c
#include "Neuron.h"
#include "functional.h"

```

```c
// Build neuron
Neuron build_neuron(int const size) {
  // Neuron struct
  Neuron n;

  // Set size
  n.size = size;

  // Allocate memory for arrays: inputs, voltage, threshold, spikes, leaks
  // No need for type casting
  n.x = calloc(size, sizeof(*n.x));
  n.v = calloc(size, sizeof(*n.v));
  n.thresh = calloc(size, sizeof(*n.thresh));
  n.leak_i = calloc(size, sizeof(*n.leak_i));
  n.leak_v = calloc(size, sizeof(*n.leak_v));
  n.s = calloc(size, sizeof(*n.s));
  n.i = calloc(size, sizeof(*n.i));
  // Reset constants
  n.v_rest = 0.0f;

  return n;
}

// Init neuron (addition/decay/reset constants, inputs, voltage, spikes,
// threshold, leaks)
void init_neuron(Neuron *n) {
  // Loop over neurons
  for (int i = 0; i < n->size; i++) {
    // Inputs
    n->x[i] = 0.0f;
    // Voltage
    n->v[i] = n->v_rest;
    // Spikes
    n->s[i] = 0.0f;
    // Trace
    n->i[i] = 0.0f;
    // Leak of current
    n->leak_i[i] = 0.0f;
    // Leak of voltage
    n->leak_v[i] = 0.0f;
    // Threshold
    n->thresh[i] = 1.0f;
  }
  // Spike counter
  n->s_count = 0;
}

// Reset neuron (inputs, voltage, spikes, threshold, trace)
void reset_neuron(Neuron *n) {
  // Loop over neurons
  for (int i = 0; i < n->size; i++) {
    // Inputs
    n->x[i] = 0.0f;
    // Voltage
    n->v[i] = n->v_rest;
    // Spikes
    n->s[i] = 0.0f;
    // Trace
    n->i[i] = 0.0f;
  }
  // Spike counter
  n->s_count = 0;
}
```

```c

// Load parameters for neuron from header file (using the NeuronConf struct)
void load_neuron_from_header(Neuron *n, NeuronConf const *conf) {
  // Check shape
  if (n->size != conf->size) {
    printf("Neuron has a different shape than specified in the NeuronConf!\n");
    exit(1);
  }
  // Loop over neurons
  for (int i = 0; i < n->size; i++) {
    // Constants for addition of voltage, threshold and trace
    n->thresh[i] = conf->thresh[i];
    n->leak_i[i] = conf->leak_i[i];
    n->leak_v[i] = conf->leak_v[i];
  }
  // Constant for resetting voltage
  n->v_rest = conf->v_rest;
}

// Free allocated memory for neuron
void free_neuron(Neuron *n) {
  // calloc() was used for voltage/decay/reset constants, inputs, voltage,
  // threshold, spike and leaks arrays
  free(n->x);
  free(n->v);
  free(n->i);
  free(n->thresh);
  free(n->s);
  free(n->leak_i);
  free(n->leak_v);
}

float sigmoidf(float n) {
    return (1 / (1 + powf(EULER_NUMBER, -n)));
}

float relu(float n){
    return (n < 0.0f ? 0.0f : n);
}

// Check spikes
static void spiking(Neuron *n) {
  // Loop over neurons
  for (int i = 0; i < n->size; i++) {
    // If above/equal to threshold: set spike, else don't
    float thresh = relu(n->thresh[i]);
    n->s[i] = n->v[i] >= thresh ? 1 : 0;
  }
}

// Do refraction
static void refrac(Neuron *n) {
  // Loop over neurons
  for (int i = 0; i < n->size; i++) {
    // If spike, then refraction
    // We don't have a refractory period, so no need to take care of that
    n->v[i] = n->s[i] == 1 ? n->v_rest : n->v[i];
    // Also increment spike counter!
    n->s_count += n->s[i] == 1 ? 1 : 0;
  }
}

// Update voltage
```

```c
static void update_voltage(Neuron *n) {
  // Loop over neurons
  for (int i = 0; i < n->size; i++) {
    // Decay difference with resting potential, then increase for incoming
    // spikes
    float leak_i = sigmoidf(n->leak_i[i]);
    float leak_v = sigmoidf(n->leak_v[i]);
    n->i[i] = (n->i[i] * leak_i) + n->x[i];
    n->v[i] = ((n->v[i] - n->v_rest) * leak_v) + n->i[i];
  }
}

// Update/reset inputs (otherwise accumulation over time)
static void update_inputs(Neuron *n) {
  // Loop over neurons
  for (int i = 0; i < n->size; i++) {
    // Set to zero
    n->x[i] = 0.0f;
  }
}

// Forward: encompasses voltage/trace/threshold updates, spiking and refraction
void forward_neuron(Neuron *n) {
  // Update voltage
  update_voltage(n);
  // Get spikes
  spiking(n);
  // Refraction
  refrac(n);
  // Reset inputs (otherwise we get accumulation over time)
  update_inputs(n);
}
```

With novel connection and neuron modules a new code that encompasses all of them had to be created. This is the network module and should contain one function for initialisation as well as one function for when the neuron network gets forwarded as new output commands are required. To perform this, the following code was created:

```c
    #include "Network.h"

// Build network: calls build functions for children
Network build_network(int const in_size, int const hid_layer_size,
                      int const hid_neuron_size, int const out_size) {
  // Network struct
  Network net;

  net.in_size = in_size;
  net.hid_layer_size = hid_layer_size;
  net.hid_neuron_size = hid_neuron_size;
  net.out_size = out_size;

  // Allocate memory for input placeholders, place cell centers and underlying
  // neurons and connections
  net.input = calloc(in_size, sizeof(*net.input));
  net.input_norm = calloc(in_size, sizeof(*net.input_norm));
  net.in_norm_min = calloc(in_size, sizeof(*net.in_norm_min));
  net.in_norm_max = calloc(in_size, sizeof(*net.in_norm_max));
  net.output = calloc(out_size, sizeof(*net.output));
  net.output_decoded = calloc(out_size, sizeof(*net.output_decoded));
  net.inhid = malloc(sizeof(*net.inhid));
  net.hid1 = malloc(sizeof(*net.hid1));
  net.hid2 = malloc(sizeof(*net.hid2));
  net.hidout = malloc(sizeof(*net.hidout));
  net.layer1 = malloc(sizeof(*net.layer1));
  net.layer2 = malloc(sizeof(*net.layer2));
  net.layer3 = malloc(sizeof(*net.layer3));

  // Call build functions for underlying neurons and connections
  *net.inhid = build_connection(hid_neuron_size, in_size);
  *net.hid1 = build_connection(hid_neuron_size, hid_neuron_size);
  *net.hid2 = build_connection(hid_neuron_size, hid_neuron_size);
  *net.hidout = build_connection(out_size, hid_neuron_size);
  *net.layer1 = build_neuron(hid_neuron_size);
  *net.layer2 = build_neuron(hid_neuron_size);
  *net.layer3 = build_neuron(hid_neuron_size);

  return net;
}

// Init network: calls init functions for children
void init_network(Network *net) {
  // Loop over input placeholders
  for (int i = 0; i < net->in_size; i++) {
    net->input[i] = 0.0f;
    net->input_norm[i] = 0.0f;
    net->in_norm_min[i] = 0.0f;
    net->in_norm_max[i] = 0.0f;
  }

  for (int i = 0; i < net->out_size; i++) {
    net->output[i] = 0.0f;
    net->output_decoded[i] = 0.0f;
  }
  // Call init functions for children
  init_connection(net->inhid);
  init_connection(net->hid1);
  init_connection(net->hid2);
  init_connection(net->hidout);
  init_neuron(net->layer1);
  init_neuron(net->layer2);
  init_neuron(net->layer3);
}

// Reset network: calls reset functions for children
void reset_network(Network *net) {
  reset_connection(net->inhid);
  reset_connection(net->hid1);
  reset_connection(net->hid2);
  reset_connection(net->hidout);
  reset_neuron(net->layer1);
  reset_neuron(net->layer2);
  reset_neuron(net->layer3);
}

// Load parameters for network from header file and call load functions for
// children
void load_network_from_header(Network *net, NetworkConf const *conf) {
  // Check shapess
  if ((net->in_size != conf->in_size) ||
      (net->hid_layer_size != conf->hid_layer_size) ||
      (net->hid_neuron_size != conf->hid_neuron_size) || (net->out_size != conf->
    out_size)) {
    printf(
        "Network has a different shape than specified in the NetworkConf!\n");
    exit(1);
  }
  // Decoding
  net->output_scale_min = conf->output_scale_min;
  net->output_scale_max = conf->output_scale_max;
  // Encoding
  for (int i = 0; i < net->in_size; i++) {
    net->in_norm_min[i] = conf->in_norm_min[i];
    net->in_norm_max[i] = conf->in_norm_max[i];
  }

  // Connection input -> hidden
  load_connection_from_header(net->inhid, conf->inhid);
  // Hidden neuron
  load_connection_from_header(net->hid1, conf->hid1);
  // Hidden neuron
  load_connection_from_header(net->hid2, conf->hid2);
  // Connection hidden -> output
  load_connection_from_header(net->hidout, conf->hidout);
  // Layer 1
  load_neuron_from_header(net->layer1, conf->layer1);
  // Layer 2
  load_neuron_from_header(net->layer2, conf->layer2);
  // Layer 3
  load_neuron_from_header(net->layer3, conf->layer3);
}

// Free allocated memory for network and call free functions for children
void free_network(Network *net) {
  // Call free functions for children
  // Freeing in a bottom-up manner
  // TODO: or should we call this before freeing the network struct members?
  free_connection(net->inhid);
  free_connection(net->hid1);
  free_connection(net->hid2);
  free_connection(net->hidout);
  free_neuron(net->layer1);
  free_neuron(net->layer2);
  free_neuron(net->layer3);
  // calloc() was used for input placeholders and underlying neurons and
  // connections
  free(net->input);
  free(net->input_norm);
  free(net->in_norm_min);
  free(net->in_norm_max);
  free(net->inhid);
  free(net->hid1);
  free(net->hid2);
  free(net->hidout);
  free(net->layer1);
  free(net->layer2);
  free(net->layer3);
}

void preprocess_input(float *input, float *input_norm, const float *in_norm_min,
    const float *in_norm_max, const int in_size)
{
    for(int idx = 0; idx < in_size - 3; idx++)
    {
        input_norm[idx] = (input[idx] - in_norm_min[idx])/(in_norm_max[idx] -
    in_norm_min[idx]);
    }
    input_norm[in_size - 3] = 0.0f;
    input_norm[in_size - 2] = 0.0f;
    input_norm[in_size - 1] = 0.0f;
```

```
}

// Function: Post-process outputs (unscaling layers)
void postprocess_output(float *output, float *output_decoded, const float
    output_scale_min, const float output_scale_max, const int out_size)
{
    int idx;

    for(idx = 0; idx < out_size; idx++)
    {
        // unscale from [0,1] to the interval [control_min, control_max]
        output_decoded[idx] = output_scale_min + output[idx] * (output_scale_max -
    output_scale_min);
    }
}

// Forward network and call forward functions for children
// Encoding and decoding inside
void forward_network(Network *net) {
  for(int i=0; i<net->out_size; i++){
    net->output[i] = 0;
  }
  preprocess_input(net->input, net->input_norm, net->in_norm_min, net->in_norm_max
    , net->in_size);
  // Call forward functions for children
  forward_connection_float(net->inhid, net->layer1->x, net->input_norm);
  forward_neuron(net->layer1);
  forward_connection_int(net->hid1, net->layer2->x, net->layer1->s);
  forward_neuron(net->layer2);
  forward_connection_int(net->hid2, net->layer3->x, net->layer2->s);
  forward_neuron(net->layer3);
  forward_connection_int(net->hidout, net->output, net->layer3->s);
  // Decode output neuron traces to scalar value
  postprocess_output(net->output, net->output_decoded, net->output_scale_min, net
    ->output_scale_max, net->out_size);
}
```

With the SNN modules translated to C programming language, it is important now to understand how the integration in the Paparazzi simulator was done as well as how the SNN parameters were passed to new programming language. Firstly, the code, required to run the Paparazzi simulator, can be found in the following repository.[5] This works simply by sending the initialisation function `build_network` followed by the loading function `load_network_from_header` when the guidance and control module is initialised. When the update of the output commands is required, the `forward_network` function is called at every timestep.

Secondly, it is important to understand how the network parameters are passed to the C module. In short, the parameters are passed within a file as 1D arrays. Thus, all the leaks and threshold values are passed as 1D arrays of the same length as the number of neurons in the layer. Similarly, the normalisation array for the input values are also passed as 1D arrays with one array for the maximum of all the parameters and one array for the minimum of all the parameters. Last but not least, the weight matrices between consecutive layers are passed as 1D arrays as well. Thus, they are built to follow the row-major criterion specific of C programming language which means that, in order to call the weight from the i^{th} row and the j^{th} columns, it is necessary to use the following structure: `i*n_col + j` where n_{col} is the number of columns. Moreover, in the current code, the number of columns is given by number of neurons in the preceding layer while the number of rows is given by the number of neurons in the following layer.

[5]https://github.com/tudelft/paparazzi/tree/race_min_snap2_SNN