university of groningen

faculty of science and engineering

Modelling and Simulation of a CMOS Synapse Implementing Two-phase Plasticity

Author:
Martin Alex NICHOLSON
s4704088

Supervisor:
Dr. Erika COVI
Second examiner :
Prof. Elisabetta CHICCA

Bachelor's Thesis
To fulfill the requirements for the degree of
Bachelor of Science in Applied Physics
at the University of Groningen

July 12, 2024

Contents

List of Figures

Glossary

ANN Artificial neural network.

CMOS Complementary metal-oxide-semiconductor.

DC Direct current.

DPI Differential pair integrator.

EPW Early-phase weight.

GPU Graphis processing unit.

LIF Leaky integrate-and-fire.

LPW Late-phase weight.

MNIST Modified National Institute of Standards and Technology.

RAM Random access memory.

SNN Spiking neural network.

STC Synaptic tagging and capture.

Abstract

Spiking neural networks are promising candidates in the search for energy efficient neuromorphic systems. This work serves as a milestone in the development of in-silicon emulations of spiking neural networks by evaluating the behaviour of a CMOS synapse, implementing the multi-timescale learning rule synaptic tagging and capture, at a network level. Synaptic tagging and capture is a biologically plausible explanation of memory storage and recall in the mammalian neocortex that operates on two time scales: the early- and the late-phases of synaptic weight change. A set of CMOS circuits have been proposed, performing synaptic tagging and capture, which this thesis replicates in a Python environment. The validity of this Python replication is corroborated by comparisons of the circuit-based script to another Python script, which was written to solve the differential equations governing synaptic tagging and capture, and by comparisons to simulations of the CMOS circuits performed in Cadence Virtuoso. The circuit-based Python script is then incorporated in a virtual spiking neural network trained on the MNIST dataset, achieving an accuracy of 96% when testing is performed on the early-phase timescale, and 93% when performed on the late-phase timescale. This is used as evidence of the functionality of the proposed CMOS synapse on a network level.

Acknowledgments

I would like to express my deepest gratitude to Dr. Erika Covi, and by extension CogniGron and the entire BICS group (chaired by Prof. Dr. Elisabetta Chicca), for providing me with the opportunity to carry out this work. I also thank Dr. Erika Covi for her unwavering support, feedback and input throughout this project, alongside Dr. Fernando M. Quintana for his invaluable insights into the field of computer science. My thanks also extend to Luca Fehlings, Paolo Gibertini, Bojian Zhang, Egidio Angelo Gallicchio and Pedro Francisco Ortiz Costa for their eagerness to help and the truly welcoming environment they provide.

Finally, I would like to offer special thanks to Jorge Navarro Quijada, who is responsible for the very basis on which this work stands, as he is the designer of the proposed CMOS synapse implementing two-phase plasticity.

1 Introduction

Neural networks have become an invaluable tool in resolving issues in natural language processing, computer vision and autonomous systems, even capable of outperforming the human brain at a variety of tasks [1, 2, 3, 4, 5, 6]. However, despite their many successes, neural networks are yet to approach the unparalleled efficiency of the human brain [7]. The brain consumes about 20 W of power [7], accumulating to about 18 000 kWh over the course of 100 years. This is a full order of magnitude lower than the estimated energy it took to train OpenAI's GPT-3 language model [7], and although GPT-3 may outperform the brain in terms of knowledge, it certainly does not outperform it in terms of functionality. A potential approach to bridging this efficiency gap lies in the transition to more biologically plausible neural networks, which aim to connect the performance of artificial intelligence with the power efficiency of a biological system. Among the promising candidates for this transition are spiking neural networks (SNNs) [8], which will be the focus of this work.

SNNs aim to mimic the behaviour of the brain, making use of biologically plausible neuron and synapse models. Through doing so, SNNs exhibit the potential for increased efficiency relative to their predecessor, the artificial neural network (ANN) [9]. The primary strength of SNNs lies in their asynchronous, event-driven operation. They process information using spikes, for instance in the form of binary digital signals, and hence, neurons within the network only activate when necessary [10], reducing energy consumption [9]. This is in contrast to synchronous digital systems, including ANNs, that rely on periodic clock pulses and thus are continuously active [10].

This work investigates a potential path towards high efficiency SNNs, through the incorporation of the biologically inspired multi-timescale learning rule synaptic tagging and capture (STC) [11] in an SNN. STC is a biologically plausible explanation of how synaptic weights in the brain are updated, describing how memories are stored and recalled [11]. A set of complementary metal-oxide-semiconductor (CMOS) circuits have been proposed, implementing STC in circuitry. Such circuits would make it possible to construct physical neural networks performing this biologically plausible learning rule [12].

The aim of this work is to replicate these circuits in a Python environment and use this emulation to demonstrate their functionality at a network level. To achieve this, a complementary Python script was first developed to solve the differential equations governing STC, which was used in conjunction with circuit simulations to validate the Python replication of the circuitry. Subsequently, a virtual SNN was constructed incorporating the circuit-based Python replication to simulate the behavior of the circuits at the network level.

2 State of the Art

This work emulates in Python code the proposed CMOS circuits implementing the biologically inspired learning rule, STC, and then tests the Python code in an SNN. The following section provides an overview of SNNs, an explanation of STC, and finally an introduction as to how STC was implemented in the CMOS circuits.

2.1 Spiking neural networks:

2.1.1 Introduction to SNNs

SNNs are a type of biologically inspired neural network, developed with the purpose of matching the performance of ANNs with the efficiency of the human brain. The biological equivalents of the components comprising an SNN are illustrated in Figure 1. Among the key differences between ANNs and SNNs is that the former operates using floating point numbers to transfer information between neurons, whereas the latter uses a more biologically plausible approach [13]. Biological neurons transmit information via action potentials. Action potentials are the rapid rise and fall of a neuron's membrane potential in response to its electrical and chemical inputs surpassing certain thresholds [14]. Owing to this rapidness, action potentials can be treated as discrete, binary spikes: an all-or-nothing event communicating the state of one neuron to its surrounding neurons via the synapses that connect them [7]. This allows the behaviour of a neuron to be described using binary arrays, with '1's representing a spike in discrete time, and '0's representing inactivity.

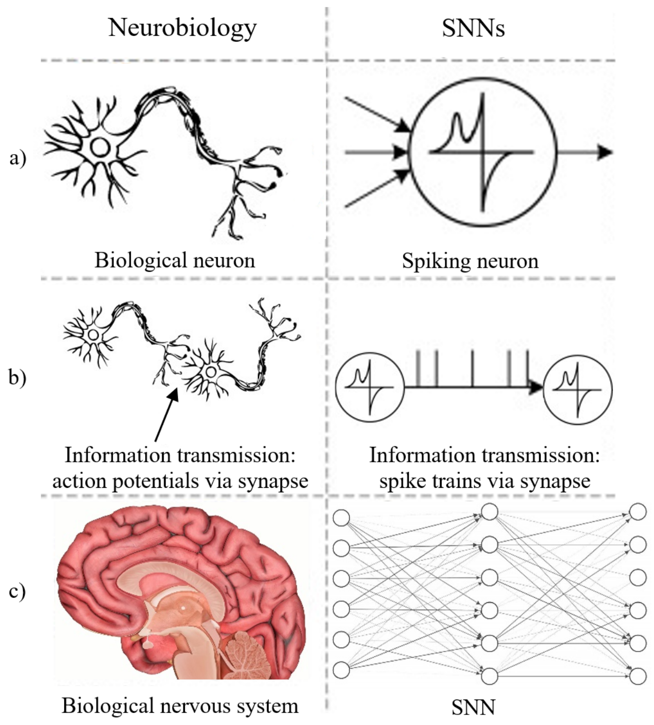

Neurobiology
SNNs
a)
Biological neuron
Spiking neuron
b)
Information transmission: action potentials via synapse
Information transmission: spike trains via synapse
c)
Biological nervous system
SNN

Figure 1: A comparison of the components of SNNs and their biological counterparts. (a) A biological neuron is modeled using a spiking neuron, as described in Section 2.1.2. (b) Information is transmitted by synapses. Biological synapses transmit action potentials, which are modeled as discrete spikes in SNNs. (c) SNNs aim to mimic the behaviour of the brain. Figure adapted from [15, 16, 17].

Since biological neurons spend the majority of their time inactive, the spiking activity of the biologically inspired neurons in SNNs tends to be limited [7]. As spiking activity is limited, the binary arrays describing neuron behaviours are sparse, meaning they contain far more zeroes than ones. This allows for data compression techniques such as run-length encoding to be incorporated into the SNNs, reducing their memory requirements, alongside their power consumption [7]. Hence, although ANNs still outperform their younger counterpart in certain areas, the improved energy and spatial efficiency of SNNs make them more suitable in edge computing applications, where power availability, space and access to a central cloud-based platform are limited. These types of applications were the initial motivation for the development of the CMOS circuits reproduced in this work [7].

2.1.2 SNN architecture: spiking neuron models

The leaky integrate-and-fire (LIF) neuron model was used in the creation of the SNN, chosen for its prevalence in neuromorphic computing and the balance it achieves between biological accuracy and computational efficiency [18]. Like biological neurons, LIF neurons are mostly inactive: they only spiking once the membrane voltage of the neuron surpasses a certain threshold. The behaviour of the membrane voltage of a LIF neuron i is given by:

$$\tau_m \frac{d}{dt} V_i(t) = -V_i(t) + V_{rev} + \sum_j \sum_{t_j^k} w_{ji} \cdot exp[-(t - t_j^k - t_{ax,delay})/\tau_{syn}] + R_m(I_{bg}(t) + I_{stim}(t)), \quad (1)$$

with membrane time constant τ_m; synaptic time constant τ_{syn}; reversal potential V_{rev}, which is the membrane potential at equilibrium; synaptic weights w_{ji} (with j being the index of presynaptic neurons and i the index of the neuron being described); presynaptic spike times t_j^k; axon delay time $t_{ax,delay}$; membrane resistance R_m; background current $I_{bg}(t)$; and stimulus current $I_{stim}(t)$ [11, 19]. The neuron i spikes once $V_i(t)$ surpasses the spiking threshold V_{th}, after which the neuron's membrane potential is set to V_{reset}. Following each spike, the potential remains at V_{reset} for the length of the refractory period t_{ref} [11, 19]. This behaviour is depicted in Figure 2.

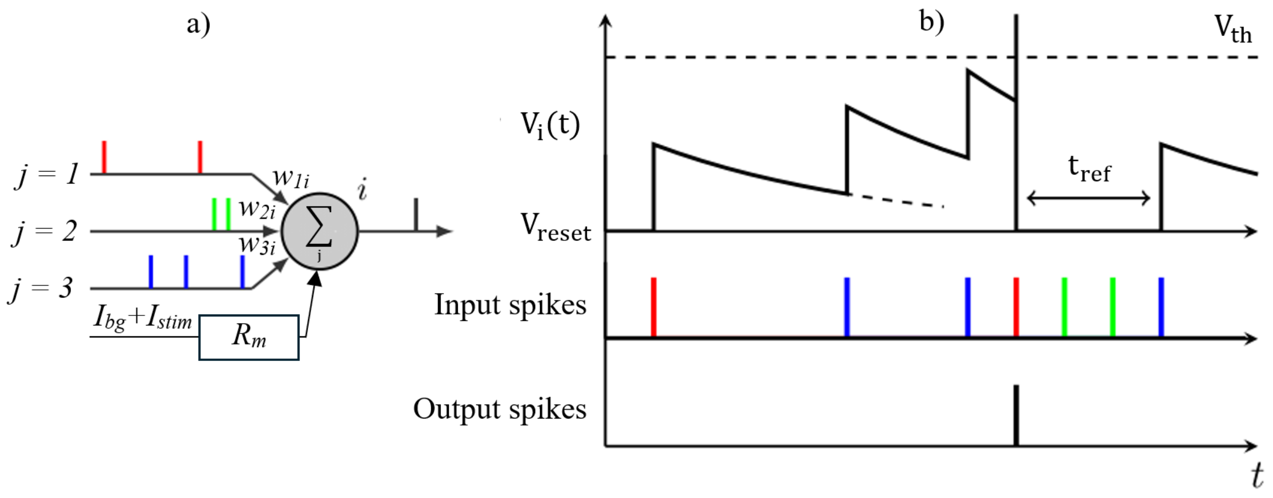


Figure 2: The behaviour of the LIF neuron. (a) A schematic of the LIF neuron model depicting the input spikes arriving to neuron i from neurons j via synapses of weight w_{ji}, and the input stimulation I_{stim} and background I_{bg} currents. The sum of these contributions causes neuron i to spike, represented by the black spike exiting the neuron. (b) The membrane voltage $V_i(t)$ and the output spikes of neuron i in response to a set of input spikes from presynaptic neurons j. In this figure $V_{rev} = V_{reset}$. Figure adapted from [20].

2.2 Synaptic tagging and capture

The STC hypothesis is a biologically plausible learning rule, providing a possible explanation of the mechanisms underlying memory storage and recall in the mammalian neocortex [11, 21, 22, 23]. The hypothesis describes the mechanisms by which synaptic weights change, altering the strength of connections between the neurons within our brains. Through the continual altering of these synaptic weights, memories are consolidated within neural networks through the formation of Hebbian cell assemblies [11]. These are clusters of neurons displaying strong synaptic connections, linking their activations. Stimulation of a portion of neurons within a Hebbian cell assembly can cause the entire assembly to spike, resulting in memory recall. This behavior can be leveraged to train neural networks to recall specific memories in response to distinct inputs [11, 19].

A critical aspect of the STC hypothesis is its operation on two distinct phases: the early- and the late-phases of long-term synaptic changes (also referred to as long-term synaptic plasticity). The early-phase describes synaptic weight changes that only last a few hours; they decay to their equilibrium values in the absence of further change-inducing inputs [11]. For the influence of the early-phase weight (EPW) changes of a synapse to be maintained, a consolidation process must occur. This consolidation process involves the change of the synapse's late-phase weight (LPW). For the LPW of a synapse to change, the EPW must meet the conditions required for a synaptic tag to be formed at the synapse, allowing it to capture protein. Protein capture at a synapse is necessary for changes in its LPW to take place [11]. The LPW configuration of a synapse is stored indefinitely [21], and is added to the EPW to produce the total synaptic weight [12].

An example of the behaviour of a synapse's weight over time, as determined by STC, is illustrated in Figure 3, which depicts the response of the synaptic weight following a rapid increase in the EPW at $t = 0$. The response demonstrates the operation of the two separate phases, with the EPW decaying to its reference value over a seven hour period, and the LPW consolidating the initial change to the EPW.

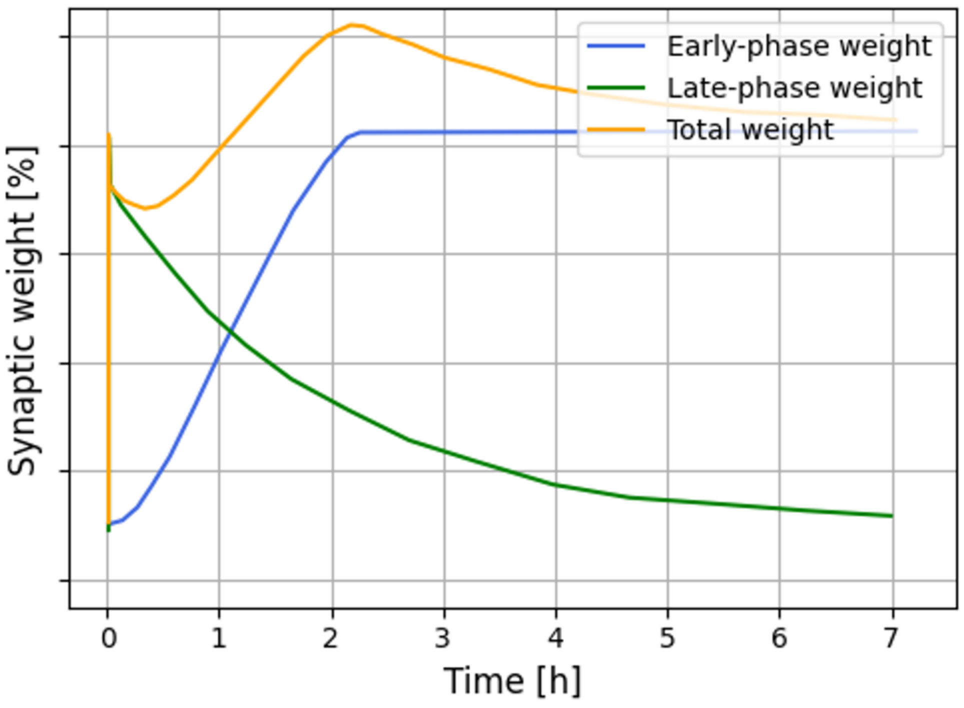


Figure 3: The behaviours of the EPW, LPW and total weight to an intensive period of pre- and postsynaptic spikes from $t = 0\,\mathrm{s}$ to $t = 0.2\,\mathrm{s}$, as determined by STC. The EPW grows during the 0.2 s time interval during which neurons are spiking, and then decays once the spiking stop. During this decay, the LPW grows, consolidating the changes to the EPW. This consolidation only occurs because the change in the EPW met the necessary conditions, which are explained in more detail in Section 2.2.3. Figure adapted from [11].

2.2.1 Calcium concentration

The working principles of STC can be split into three subsections. The first is the response of the calcium concentration at the synapse, which ultimately determines the early-phase and thus late-phase responses. The calcium concentration response is driven by pre- and postsynaptic spikes of the neurons at either end of the synapse, as given by [12]:

$$\frac{d}{dt}c_{ji}(t) = -\frac{c_{ji}(t)}{\tau_c} + c_{\mathrm{pre}}\sum_n \delta(t - t_j^n - t_{\mathrm{c,delay}}) + c_{\mathrm{post}}\sum_m \delta(t - t_i^m), \tag{2}$$

with $c_{ji}(t)$ the calcium concentration at the postsynaptic site; τ_c the calcium time constant; c_{pre} and c_{post} the increase in the calcium concentration due to a pre- and postsynaptic spike, respectively; t_j^n and t_i^m the pre- and postsynaptic spike times, respectively; and $t_{\mathrm{c,delay}}$ the delay in the calcium concentration response after a presynaptic spike.

The first term on the right-hand side of Equation (2) results in the exponential decay of the calcium concentration to zero, whereas the second and third terms give the contributions of pre- and postsynaptic spikes to the calcium concentration, respectively. The behaviour elicited by Equation (2) is depicted in Figure 4.

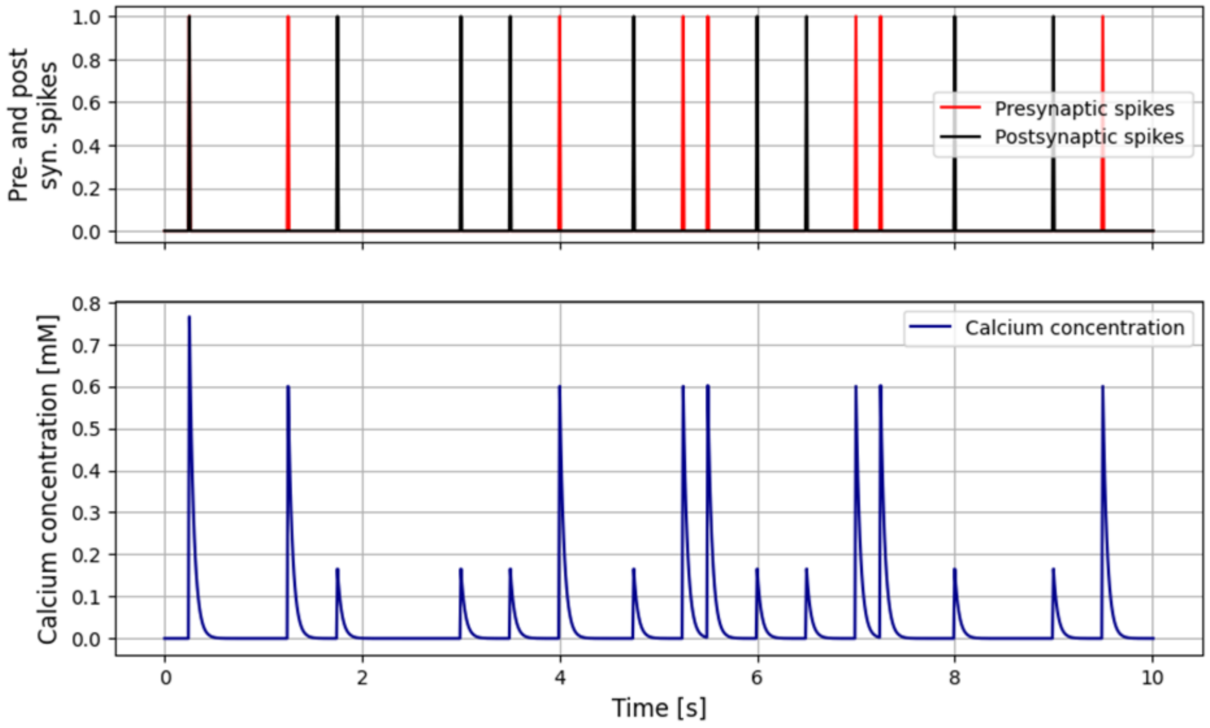


Figure 4: The behaviour of the calcium concentration at the postsynaptic site in response to the presence of pre- and postsynaptic spikes. The behaviour shows a distinct increase in the calcium concentration in response to each type of spike, followed by exponential decay.

2.2.2 Early-phase weight

The calcium concentration then determines the behaviour of the EPW of the synapse. The EPW refers to changes in the synaptic weight that last a few hours [11]. The behaviour of the EPW $h_{ji}(t)$ is described by [12]:

$$\tau_h \frac{dh_{ji}(t)}{dt} = 0.1(h_0 - h_{ji}(t)) + \gamma_p \cdot (h_{max} - h_{ji}(t)) \cdot \Theta(c_{ji}(t) - \theta_p) - \gamma_d \cdot h_{ji}(t) \cdot \Theta(c_{ji}(t) - \theta_d). \quad (3)$$

Here, τ_h is the early-phase time constant; h_{ji} the reference/equilibrium value of the EPW; γ_p and γ_d the potentiation and depression strengths, respectively; h_{max} the maximum value of the EPW; $\Theta(\cdot)$ the Heaviside function; and θ_p and θ_d the potentiation and depression thresholds, respectively.

Equation (3) illustrates the three ways in which the EPW is updated: potentiation, depression and recovery. Potentiation and depression refer to the rapid increase and decrease in the EPW, as a result of the calcium concentration surpassing the potentiation and depression thresholds. The behaviour of these mechanisms is given by the second and third terms on the right-hand side of Equation (3); potentiation and depression respectively. The depression threshold θ_d is lower than the potentiation threshold θ_p, which means that if potentiation is occurring, so is depression. However, γ_p is roughly five times the magnitude of γ_d, making potentiation the dominant mechanism in this scenario. Finally, the recovery term $0.1(h_0 - h_{ji}(t))$ describes the exponential return of the EPW to its reference value. Recovery is always active, but is two orders of magnitude weaker than both potentiation and depression [11, 12]. Figure 5 demonstrates the behaviour of the EPW in response to changes in the calcium concentration.

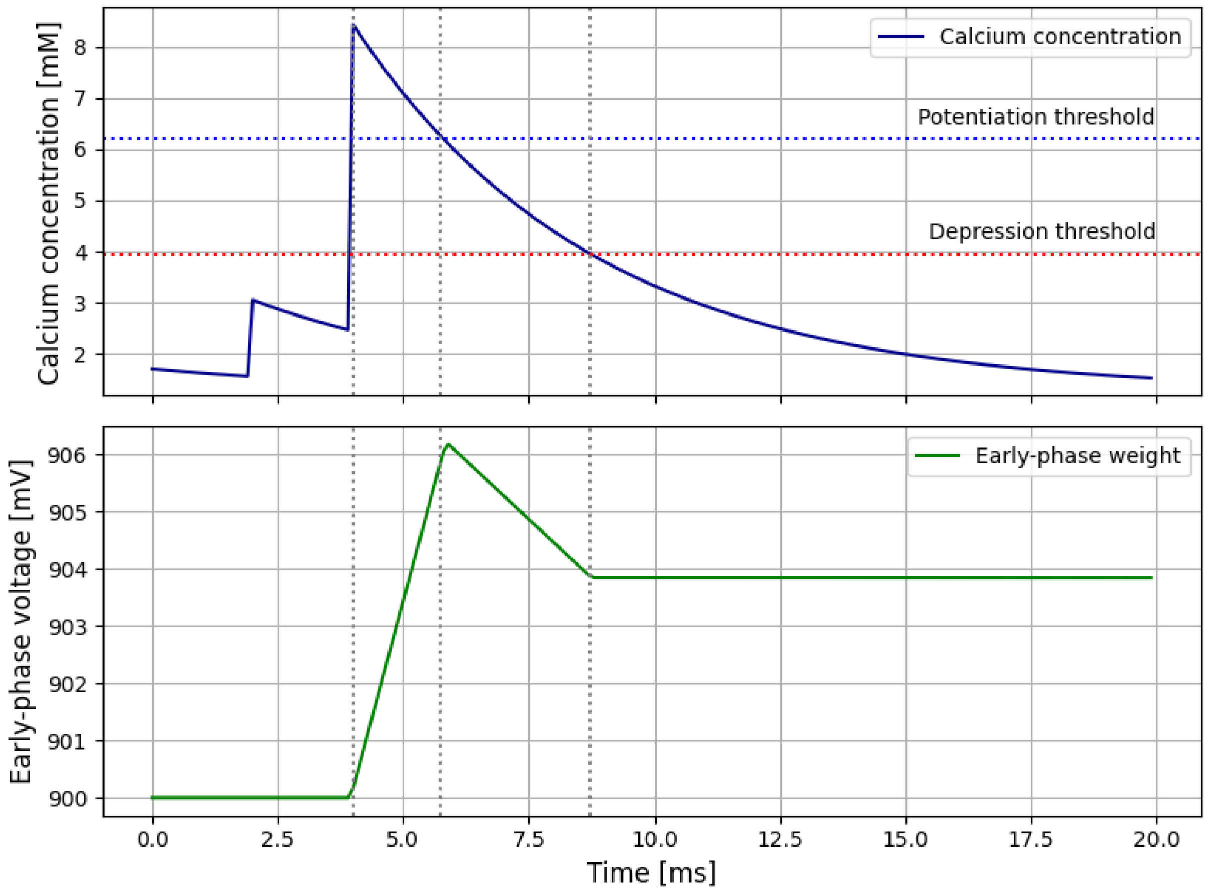


Figure 5: The behaviour of the EPW in responses to changes in the calcium concentration. Potentiation of the EPW is visible during the time interval in which the calcium concentration is above the potentiation threshold (from 4.0 s to 5.9 s), and depression is visible during the time interval the calcium concentration is between the potentiation and depression thresholds (from 5.9 s to 8.4 s). Although always active, recovery is not visible in this figure. This is due to the magnitude of the early-phase time constant (688.4 s), which makes recovery a slow process that cannot be observed on the millisecond timescale.

2.2.3 Late-phase weight

The behaviour of the EPW dictates the response of the synapse's LPW. A sufficient deviation of the EPW from its reference value triggers the formation of a synapse-specific tag, leading to the capture of proteins in the synapse (hence the name synaptic tagging and capture [11]). The concurrence of a synaptic tag and sufficient protein availability results in changes to the LPW. This process is described by [11]:

$$\tau_z \frac{dz_{ji}(t)}{dt} = p_i(t) \cdot (1 - z_{ji}(t)) \cdot \Theta((h_{ji}(t) - h_0) - \theta_{\text{tag}}) - p_i(t) \cdot (z_{ji}(t) + 0.5) \cdot \Theta((h_0 - h_{ji}(t)) - \theta_{\text{tag}}). \tag{4}$$

Here, z_{ji} is the LPW variable; τ_z is the late-phase time constant; and θ_{tag} the tagging threshold that must be surpassed for the formation of a synaptic tag. The first term on the right-hand side of the equation describes late-phase potentiation, and the second term describes late-phase depression. The protein availability $p_i(t)$ at the postsynaptic site is determined based on the sum of the EPW change across all the synapses j connected to this site, as given by [12]:

$$p_i(t) = \begin{cases} \alpha & \text{if } \sum_j \left|h_{ji}(t^*) - h_0\right| > \theta_{pro} \text{ at any } t^* \leq t, \\ 0 & \text{else.} \end{cases} \tag{5}$$

For late-phase potentiation or depression to occur at synapse i, the sum of the absolute value of the change in the EPW of all the synapses j connected to the post-synaptic site of synapse i must be larger than the protein capture threshold θ_{pro}; i.e., $\left|h_{ji}(t^*) - h_0\right| > \theta_{pro}$. Once this condition is met, the protein availability becomes α and stays this way indefinitely. This influence this has on the LPW behaviour is illustrated in Figure 6.

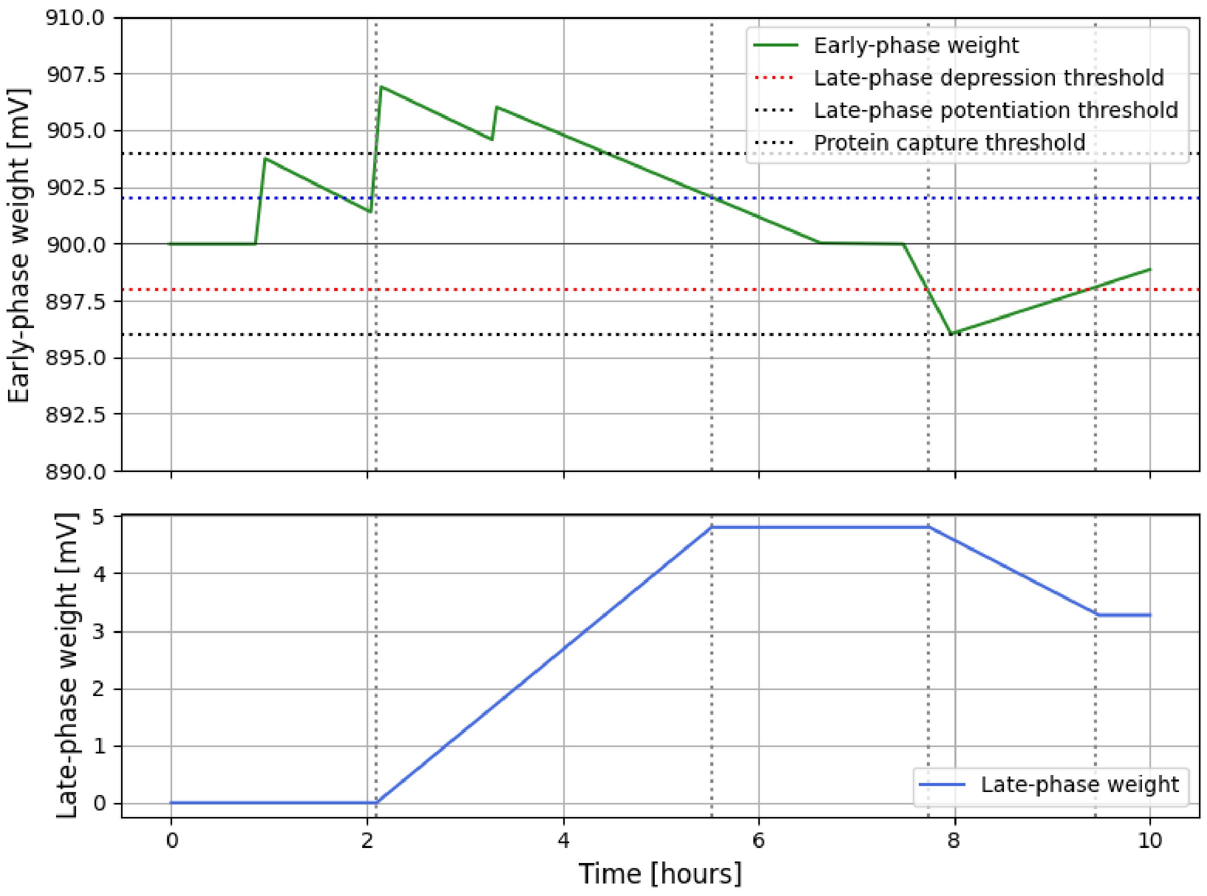


Figure 6: Changes in the LPW in response to changes in the EPW. Before the EPW crosses the protein capture threshold the LPW stays constant, since $p_i(t) = 0$ (from 0.0 s to 2.1 s). Once this threshold is passed (at 2.1 s), a synaptic tag forms and $p_i(t)$ is set to α. The LPW then potentiates when the EPW is above the potentiation threshold (from 2.1 s to 5.5 s), and depresses when the EPW is below the depression threshold (from 7.7 s to 9.4 s). Unlike the EPW, the LPW does not recover.

2.3 Implementation of STC in the CMOS circuits

This work evaluates the implementation of STC in the CMOS circuits proposed in [12] through emulating the behaviour of the circuits in a Python environment. This work then uses the Python code to simulate the behaviour of the circuitry on a network level. This subsection provides an overview of how the circuitry executes the differential equations governing STC presented in the previous subsection, focusing only on the calcium and early-phase portions of the circuits. The explanation of the implementation of the late-phase calculation is excluded, as the code produced in this work uses Equation (4), rather than mimicking the late-phase circuitry. The reasons for this stem from the fact that this portion of the circuit is likely to be evaluated with emerging devices in future iterations of the CMOS circuits, alongside the relative complexity of the late-phase circuitry. This is described in more detail in the *Discussion*.

2.3.1 Calculation of the calcium concentration in the CMOS circuits

A differential pair integrator (DPI) [24] is used to perform the calcium concentration calculations described in Equation (2). The DPI is a current-mode circuit [25], that receives voltage pulses as inputs and outputs a current. The voltage pulses are used to communicate input spikes to the CMOS synapse, and the output current is used as an indicator of the calcium concentration in the synapse [12]. A schematic of the DPI is provided in Figure 7.

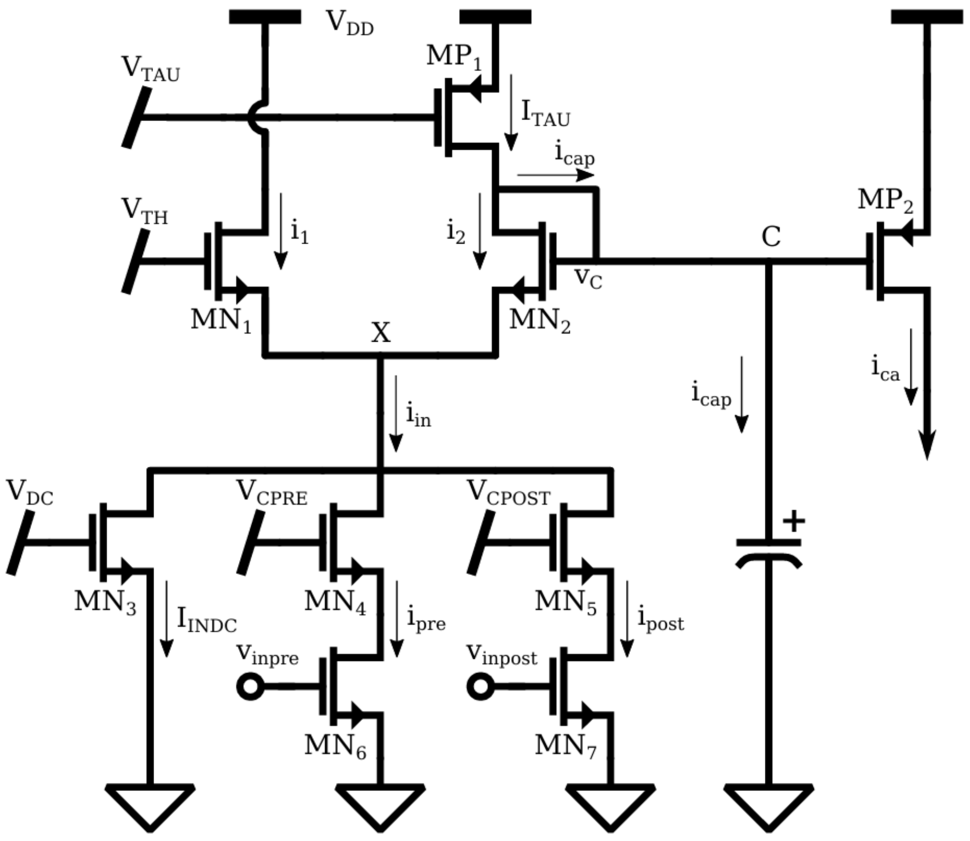


Figure 7: The DPI circuit. The operation of the DPI requires five biasing voltages and three external currents. V_{TH} is used to tune the gain; V_{TAU} is used to tune the time constant of the differential equation the DPI behaviour exhibits; and V_{CPRE}, V_{CPOST}, and V_{DC} are used to weight the inputs i_{pre}, i_{post} and I_{INDC}, respectively [12]. These inputs are explained in the description of Equation 6. Figure extracted from [12].

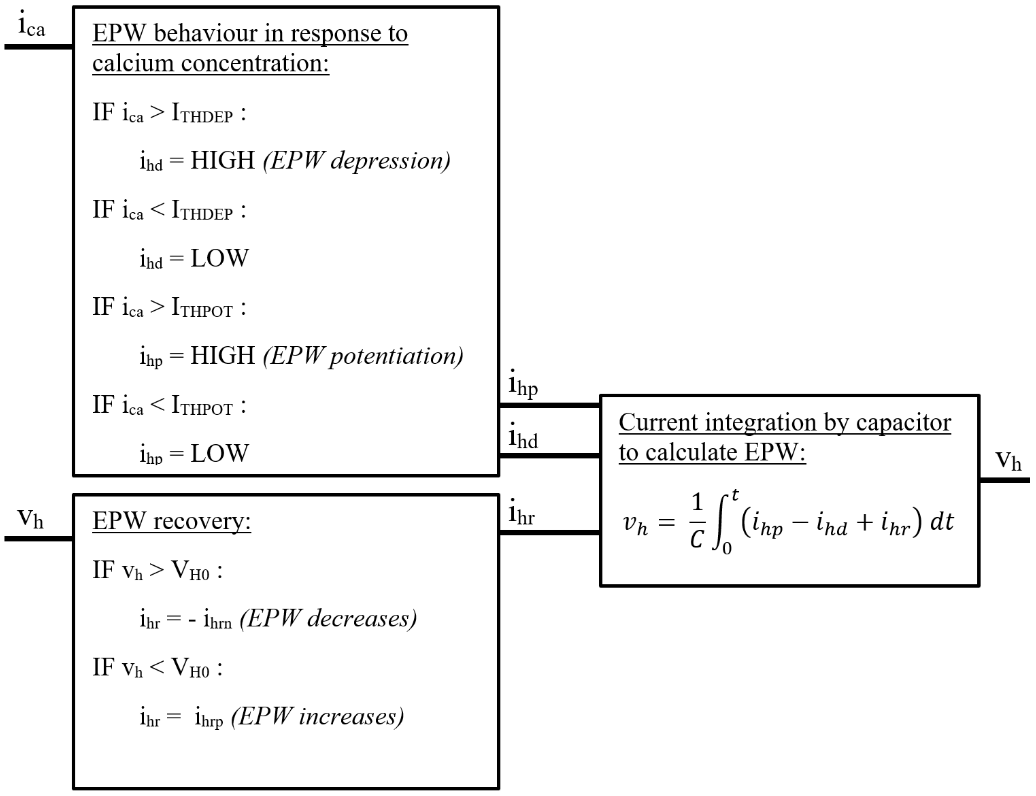
The behaviour of the DPI is given by [12]:

$$\frac{d}{dt}i_{ca} = \frac{1}{\tau_{DPI}}\left(-i_{ca} + \frac{I_{\mathrm{TH}}}{I_{\mathrm{TAU}}}(i_{\mathrm{pre}} + i_{\mathrm{post}} + I_{\mathrm{INDC}})\right). \tag{6}$$

Here, the magnitude of the current i_{ca} indicates the calcium concentration at the postsynaptic site; τ_{DPI} is analogous to the calcium time constant; and i_{pre} and i_{post} are analogous to the contributions to the calcium concentration in response to pre- and postsynaptic spikes, respectively. The only difference between Equation (6), and Equation (2) (the differential equation governing the calcium concentration response in STC) is the addition of the terms I_{TH}, to tune the gain of the circuit; I_{TAU}, to tune the time constant; and I_{INDC}, which is the DC component of the tail current to stimulate the DPI [12]. These additional terms are only there as they are necessary terms in operating the DPI circuit. Their inclusion does not prevent the circuit from correctly performing the calculations of the calcium concentration given by Equation (2).

2.3.2 Calculation of the early-phase weight in the CMOS circuits

The output current i_{ca} of the DPI is fed into the early-phase portion of the CMOS circuits, consisting of a control block [12], two winner-take-all circuits [26, 27], three ultra low leakage blocks [12] and a capacitor. The early-phase circuit takes i_{ca} as an input and calculates the EPW based on Equation (3), outputting the EPW as the voltage v_h. The voltage v_h is then fed back into the input of the early-phase circuit, allowing the circuit to perform the recovery term of Equation (3). The logic extracted from the early-phase circuit is presented by the block diagram in Figure 8.



i_{ca}
EPW behaviour in response to calcium concentration:
IF $i_{ca} > I_{THDEP}$:
i_{hd} = HIGH *(EPW depression)*
IF $i_{ca} < I_{THDEP}$:
i_{hd} = LOW
IF $i_{ca} > I_{THPOT}$:
i_{hp} = HIGH *(EPW potentiation)*
IF $i_{ca} < I_{THPOT}$:
i_{hp} = LOW
v_h
EPW recovery:
IF $v_h > V_{H0}$:
$i_{hr} = - i_{hrn}$ *(EPW decreases)*
IF $v_h < V_{H0}$:
$i_{hr} = i_{hrp}$ *(EPW increases)*
i_{hp}
i_{hd}
i_{hr}
Current integration by capacitor to calculate EPW:
$v_h = \frac{1}{C}\int_0^t (i_{hp} - i_{hd} + i_{hr})\, dt$
v_h

Figure 8: A block diagram illustrating the logic extracted from the early-phase portion of the synapse circuits. i_{ca} is the current indicating the calcium concentration; I_{THPOT} and I_{THDEP} are the potentiation and depression thresholds, respectively; i_{hp} and i_{hd} are the potentiation and depression currents, respectively; i_{hr} is the recovery current; i_{hrn} is the magnitude of the recovery current when the EPW is above its reference value V_{H0}; i_{hrp} is the magnitude of the recovery current when the EPW is below its reference value V_{H0}; and C is the capacitance of the capacitor used to perform the integral of the potentiation, depression and recovery currents.

The block diagram in Figure 8 consists of three blocks. The first (top left) compares the calcium concentration to the early-phase potentiation and depression thresholds, and depending on this comparison, sets the potentiation and depression currents to either high or low. When the current is set to high, it means the corresponding mechanism is active. The second block (bottom left) sets the recovery current i_{hr} to either a positive value i_{hrp} or negative value $-i_{hrn}$ depending on whether the EPW voltage v_h currently lies above or below the reference value v_{H0}, determining the direction and magnitude of EPW recovery. Finally, the integral of the potentiation i_{hp}, depression i_{hd} and recovery currents i_{hr} is performed using a capacitor, accounting for the negative contribution to the EPW that depression should cause. The result of this integral produces v_h.

3 Results

3.1 Summary of methods

This work reproduces the behaviour of a CMOS implementation of a synaptic model, performing the biologically inspired multi-timescale learning rule STC, in a Python environment. To validate the accuracy of this replication, an additional Python script was produced, based on the differential equations governing STC. A comparison of the two scripts was performed, verifying that the circuit-based script performs STC.

The output of the circuit-based script was also compared to simulations of the circuitry itself, performed in Cadence Virtuoso, testing whether the similarity was sufficient to use the Python code to analyse the performance of the CMOS circuits at a network level. As these comparisons both yielded satisfactory results, the Python code was then incorporated into a two-layer feedforward SNN, which was trained and tested using the MNIST dataset.

The following subsections present the results of the aforementioned comparisons, displaying the similarity of the code output with regard to the CMOS circuits, which is in supporting the claim that the synapse circuitry functions on a network level.

3.2 Reproduction of the synapse circuits in Python

The code reproducing the CMOS implementation of the synapse was written based on the logic extracted from the circuits, as presented in the *State of the Art*. The calcium concentration updates, in response to pre- and postsynaptic spikes, were calculated using Equation (6), and the early-phase weight calculation was executed using the logic displayed in the block diagram in Figure 8. However, as mentioned in Section 2.3, the late-phase portion of the code deviates from the circuitry, solving Equation (4), rather than performing a direct implementation of the circuit logic. More detail on this can be found in the *Discussion*.

The code was written in Python 3.10, using the Google JAX framework and the Equinox library. The complete code is displayed in Appendix A.

3.2.1 Testing of the synapse circuit-based script

To test the code reproducing the synapse circuits, another script was written based purely on the differential equations governing STC (Equations 2, 3 and 4), solving them using the forward Euler method. This script served as a control: it exhibits the behaviour of the differential equations that the circuitry is based on. Comparisons between the two Python scripts were then performed to verify that the circuit reproduction does perform STC as intended. One such comparison is depicted in Figure 9, illustrating the response of the two scripts to a random array of pre- and postsynaptic spikes. The difference in the time scale is a result of the acceleration factor of ten implemented in the CMOS model, which was applied to relax circuit design constraints [12].

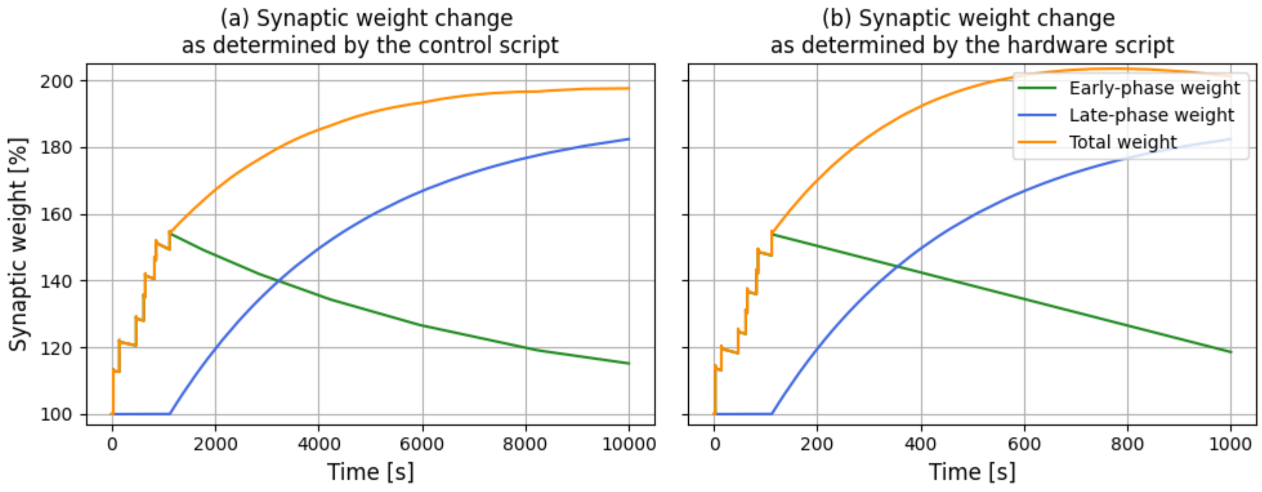


Figure 9: The output of the control script (a) and the circuit-based script (b) in response to an identical test input consisting of randomly distributed pre- and postsynaptic spikes in the first tenth of the time interval (from 0 to 1000 s in (a), and from 0 to 100 s in (b)), followed by an absence of spikes. The circuit-based code can be found in Appendix A, the differential equation-based code in Appendix B and the constants used to produce these figures can be found in Appendix C. The code used to produce this figure, alongside all other relevant figures can be accessed via the link provided in Appendix E.

The comparison of the two scripts illustrated in Figure 9 was used to substantiate the correct functionality of the circuit replication, as both codes exhibit the defining characteristics of STC at the appropriate times. Potentiaton - the steep increase in the EPW - is observed during the interval in which pre- and postsynaptic spikes are present, as these spikes cause the calcium concentration to surpass the potentiation threshold. The drop following each of the potentiations is due to both depression and EPW decay. The two mechanisms are distinguishable by the rate of the decrease they elicit. Figure 10 presents an enlarged depiction of one of the potentiation spikes in Figure 9, clearly illustrating the two different gradients in the decline of the EPW. The steeper of the two is depression, caused by the brief period during which the tail of the calcium concentration has decayed to below the potentiation threshold, but is still above the depression threshold. The sudden leveling out of the gradient occurs once the calcium concentration decays beyond the depression threshold, and the only contribution to the EPW is its recovery towards its steady state.

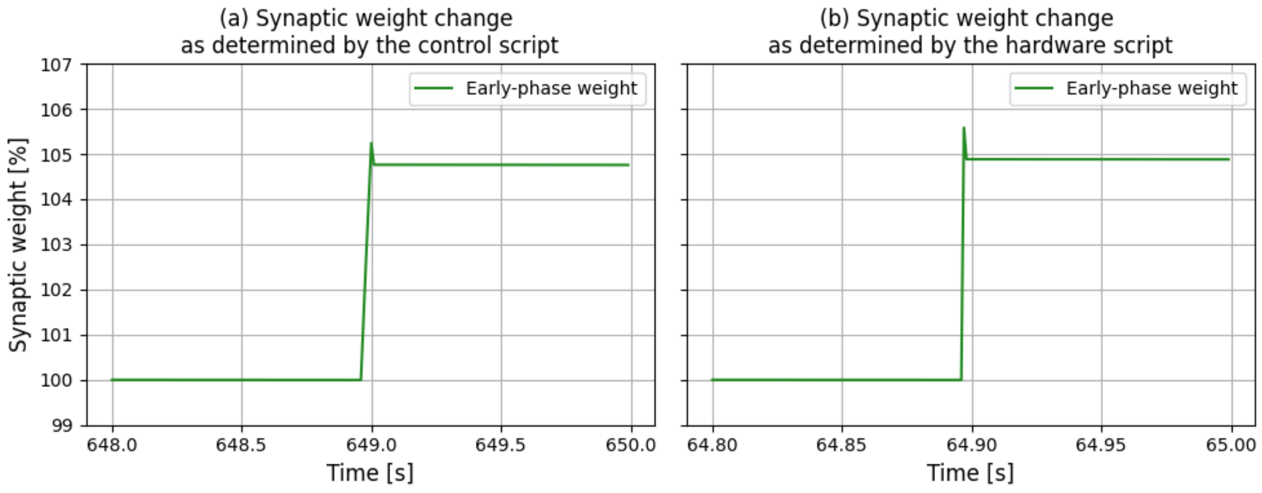


Figure 10: An enlarged version of the EPW, focusing on its behaviour directly after potentiation occurs. Although the effects of depression are clearly visible in this figure (the steep drop in the EPW), the effects of EPW decay/recovery are not. This is merely due to the narrow time window used, relative to the time constant determining this decay (688.4 s for (a) and 68.84 s for (b)).

Aside from the timescale, the only major difference in the behaviours of the two codes is that in the script based on the circuits, recovery is linear, whereas in the script based on the equations it is exponential. This difference arises from the circuitry itself, rather than from an oversight in one of the two codes. The circuits calculate the rate of recovery by integrating the recovery current over time. This current is either one of two values: a fixed negative value or a fixed positive value, depending on whether the EPW is above or below the EPW reference value. Hence, the integration of the recovery current over time produces a linear response, rather than the exponential decay observed in the behaviour of the differential equation the CMOS circuits are based on. The justification for this approximation is the magnitude of the early-phase time constant: it is large enough that a linear EPW recovery is not a problematic difference.

In addition to potentiation, depression and recovery, consolidation is also observed in Figure 9, with the initiation of changes to the LPW demonstrating the presence of a synaptic tag. This means the EPW change has exceeded the tagging threshold. In the example presented in Figure 9, this threshold is reached just as the synaptic spikes stop. Hence, as the EPW decays, it is consolidated by means of long-term potentiation of the LPW. It is worth noting that if the test input had resulted in the EPW being depressed beyond the tagging threshold, this would be consolidated by long-term depression of the LPW.

3.2.2 Comparison of the circuit-based script to literature

To further investigate the accuracy of the circuit-based script in simulating the CMOS implementation of the STC-based synapse, the code was compared to both previously conducted simulations of the CMOS circuits and the results of the STC simulations published in [11]. Previous simulations of the circuits were performed Cadence Virtuoso [12], whereas the results in [11] were determined using a C++ script written to solve the differential equations governing STC.

Figure 11 depicts the response of an STC-based synapse to a short period of intense potentiation. Figures 11 (a) and 11 (c) are the outputs of the circuit-based script produced in this work, with the

EPW recovery accelerated by a factor of 32 and 7.2, respectively. The acceleration factor of 32 was chosen to match the Cadence simulation, presented in Figure 11 (b). Figure 11 (c) was included to demonstrate that reducing the acceleration factor used in the circuit-based script exaggerates the curve in the total synaptic weight, which is observed in both Figures 11 (b) and 11 (d), with 11 (d) presenting the results of the simulations performed in [11].

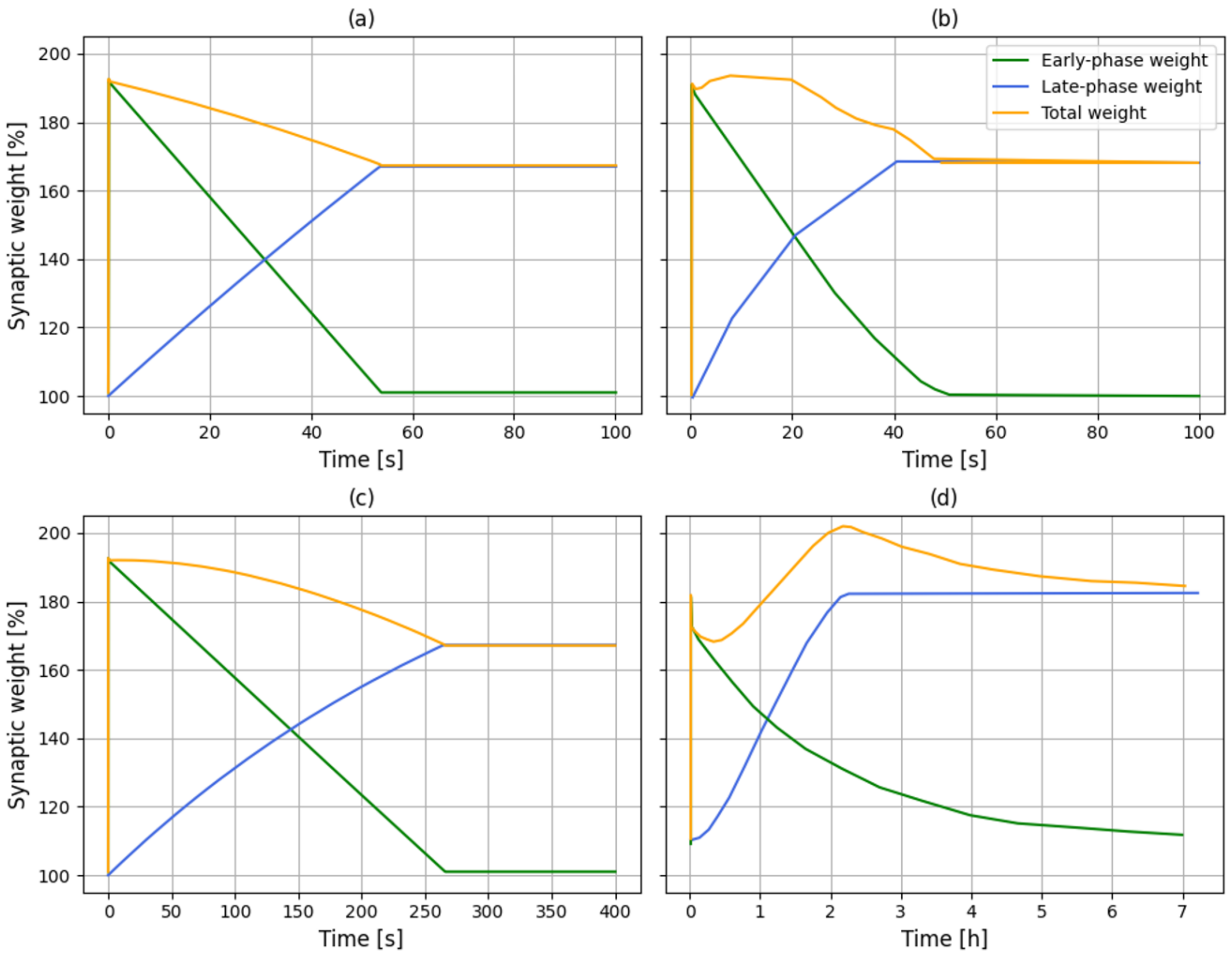


Figure 11: Four simulations of the response of the synaptic weight to a strong potentiation 0.2 s-long at t = 0. (a) The output of the circuit-based script produced in this work, using an acceleration factor of 32 for the EPW decay. (b) The results produced by a simulation of the synapse circuits in Cadence Virtuoso, using an acceleration factor of 32 for the EPW decay. The data presented is extracted from [12]. (c) The output of the circuit-based script produced in this work, using an acceleration factor of 7.2 for the EPW decay. (d) An adaptation of the figure presenting the output of the simulation performed in [11].

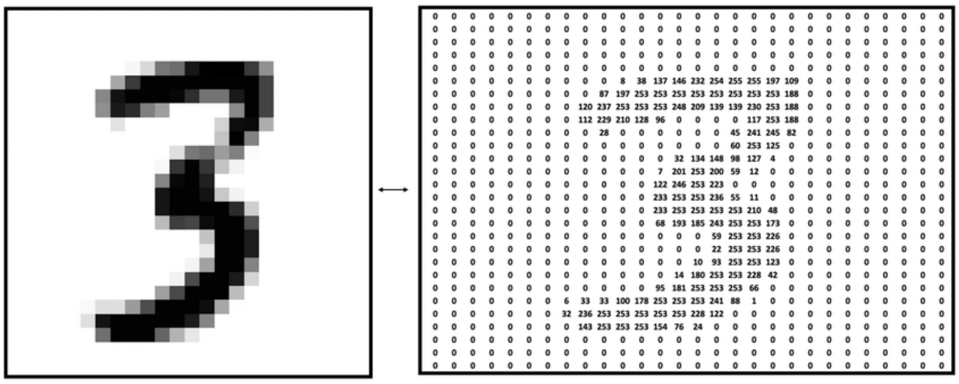
Although there are observable differences in the behaviours illustrated in the four sub-figures, they are comparable. If the acceleration factor used in (c) is accounted for, then the responses in (a), (b) and (c) all exhibit activation of the same STC mechanisms at the appropriate times: potentiation during the 0.2 s-long spiking interval at the start of the simulation; recovery of the EPW once the spiking input ends; and consolidation of the EPW by means of potentiation of the LPW, which indicates the development of a synaptic tag during the EPW potentiation, leading to protein capture at the synapse.

Sub-figure (d) is somewhat of an outlier, but this is logical, as this work aims to replicate the CMOS circuits exhibiting the behaviour in Figure 11 (b), not the exact behaviour illustrated in 11 (d). The simulations performed to obtain the results in (d) used slightly different differential equations to the ones implemented in the circuitry, with the latter omitting two terms in the equation determining the EPW, specifically a term dictating calcium-dependent fluctuations and a term enforcing two 'special plasticity conditions' [12, 11]. The simulations performed in [11] have only been included to serve as a reference point for the expected circuit behaviour, as they exhibit the 'targeted behaviour' [12] of the CMOS circuits replicated in Python by this work.

The comparability of the behaviours depicted in Figures 11 (a), (b) and (c) indicates that the extent to which the circuit-based Python script accurately replicates the behaviour of the CMOS circuits is sufficienct. It meets the requirements to which this work will be subjected, as this work serves as a proof-of-concept, rather than an exact simulation of the circuits. The justification for not needing an exact simulation as of yet is detailed in the *Discussion*.

3.3 Creation of an SNN to investigate the behaviour of the synapse circuitry at a network level

Once it had been verified that the circuit-based Python script was a sufficiently accurate replica of the CMOS circuits, the code was used to simulate the circuitry on a network level. This was achieved by creating an SNN using the circuit-based Python script and then training and testing the network using the MNIST dataset. The MNIST dataset is a database of handwritten digits, rasterised onto a 28×28 pixel grid, with each pixel being assigned an intensity value, ranging from values of 0 to 255 [28]. A visual explanation of this encoding process is depicted in Figure 12.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	8	38	137	146	232	254	255	255	197	109	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	87	197	253	253	253	253	253	253	253	253	188	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	120	237	253	253	253	248	209	139	139	230	253	188	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	112	229	210	128	96	0	0	0	0	117	253	188	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	28	0	0	0	0	0	0	45	241	245	82	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	60	253	125	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	32	134	148	98	127	4	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	7	201	253	200	59	12	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	122	246	253	223	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	233	253	253	236	55	11	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	233	253	253	253	253	210	48	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	68	193	185	243	253	253	173	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	59	253	253	226	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	22	253	253	226	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	93	253	253	123	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	14	180	253	253	228	42	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	95	181	253	253	253	66	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	6	33	33	100	178	253	253	253	241	88	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	32	236	253	253	253	253	253	228	122	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	143	253	253	253	154	76	24	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 12: The encoding of a handwritten '3' as an MNIST sample. Figure extracted from [29].

3.3.1 Network architecture

The SNN created comprises a two-layer, feedforward structure, as depicted in Figure 13. The input layer contains 784 neurons, one for each pixel of the 28×28 grid. Each neuron in the first layer is connected by an STC synapse to each output neuron in the second layer. The number of output neurons varied, matching the number of digits the network was being trained to identify (e.g. if the network was only being trained to identify ones and zeros, there were only two output neurons). Each output neuron has two sources of input: the input spikes delivered via the synapses from the first layer neurons and secondly, a stimulus input in the form of a stimulation current, active only during the training of the network. The network uses LIF neurons, which were implemented in the code by solving Equation 1 in discrete time using the forward Euler method.

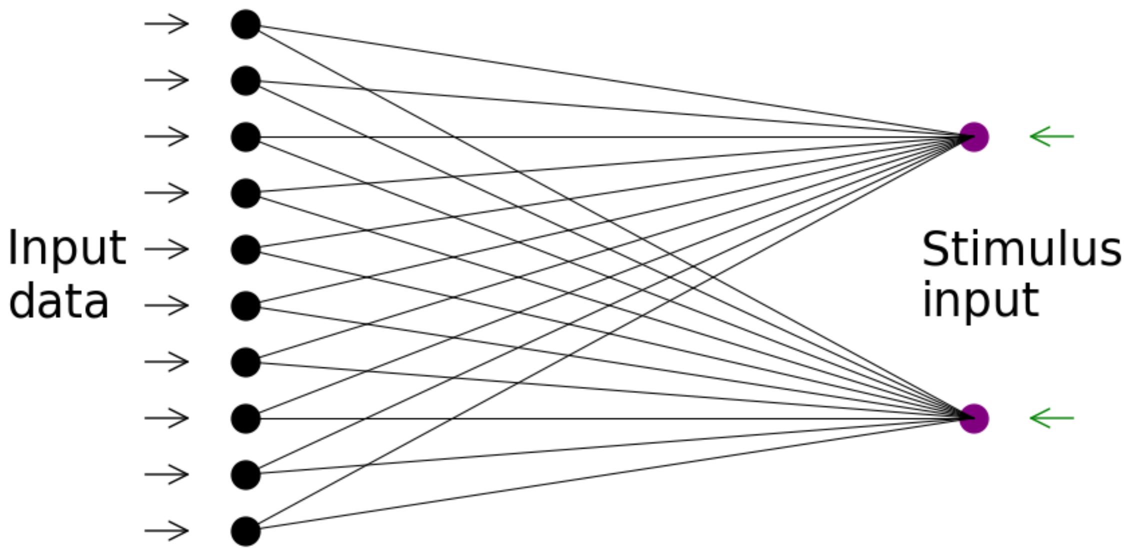


Figure 13: A two-layer, feedforward neural network with input neurons on the left and output neurons on the right. This figure only illustrates ten of the 784 neurons in the first layer for improved clarity. Arrows on the left represent the MNIST input data, whereas arrows on the right represent the stimulus input.

3.3.2 Training and testing

To train the network, the input data was first rate-encoded, meaning information was communicated by neurons through making use of different spiking frequencies. A linear translation was used, projecting the pixel intensities onto a range of firing rates starting at 0 Hz (corresponding to a pixel intensity of zero) and ending at 40 Hz (for the maximum pixel intensity value of 255). The 40 Hz limit was chosen to keep firing rates within a biologically plausible range [30].

During training, each sample was presented to the network for 150 ms, followed by a 200 ms phase absent of any spikes, allowing all neuron membranes to decay to their resting values. Each 150 ms phase was accompanied by a 1 nA stimulation current pulse sent to the second layer neuron matching the label of the sample being presented, and a -1 nA inhibition current sent to all other second layer neurons. The purpose of the stimulatory/inhibitory current pulses is to ensure only the correct second layer neuron fires for each sample presented, i.e., the second layer neuron matching the label of the sample being presented. This winner-take-all mechanism is a computationally efficient way to implement the behaviour of inhibitory neurons. It ensures that for each sample labelled 'X' presented to the network, only the synapses connecting the firing input neurons and the output neuron labelled 'X' experience both pre- and postsynaptic spikes. All other synapses experience either no spiking, or *only* either pre- or postsynaptic spikes.

Minor adjustments were made to the currents and thresholds defining the behaviour of the synapse (the updated variables can be found in Appendix D), such that the presence of both a pre- and postsynaptic spike at a synapse would result in potentiation, and the presence of only one of the two would result in depression. As a result, the synapses connecting the neurons in the first layer (that tend to spike for input 'X') and neuron 'X' in the second layer strengthen, whereas all other connections weaken. Hence, the second layer neuron 'X' becomes far more likely to spike when a sample 'X' is presented to the network in future, allowing the network to distinguish between different inputs. This process is visualised in Figure 14.

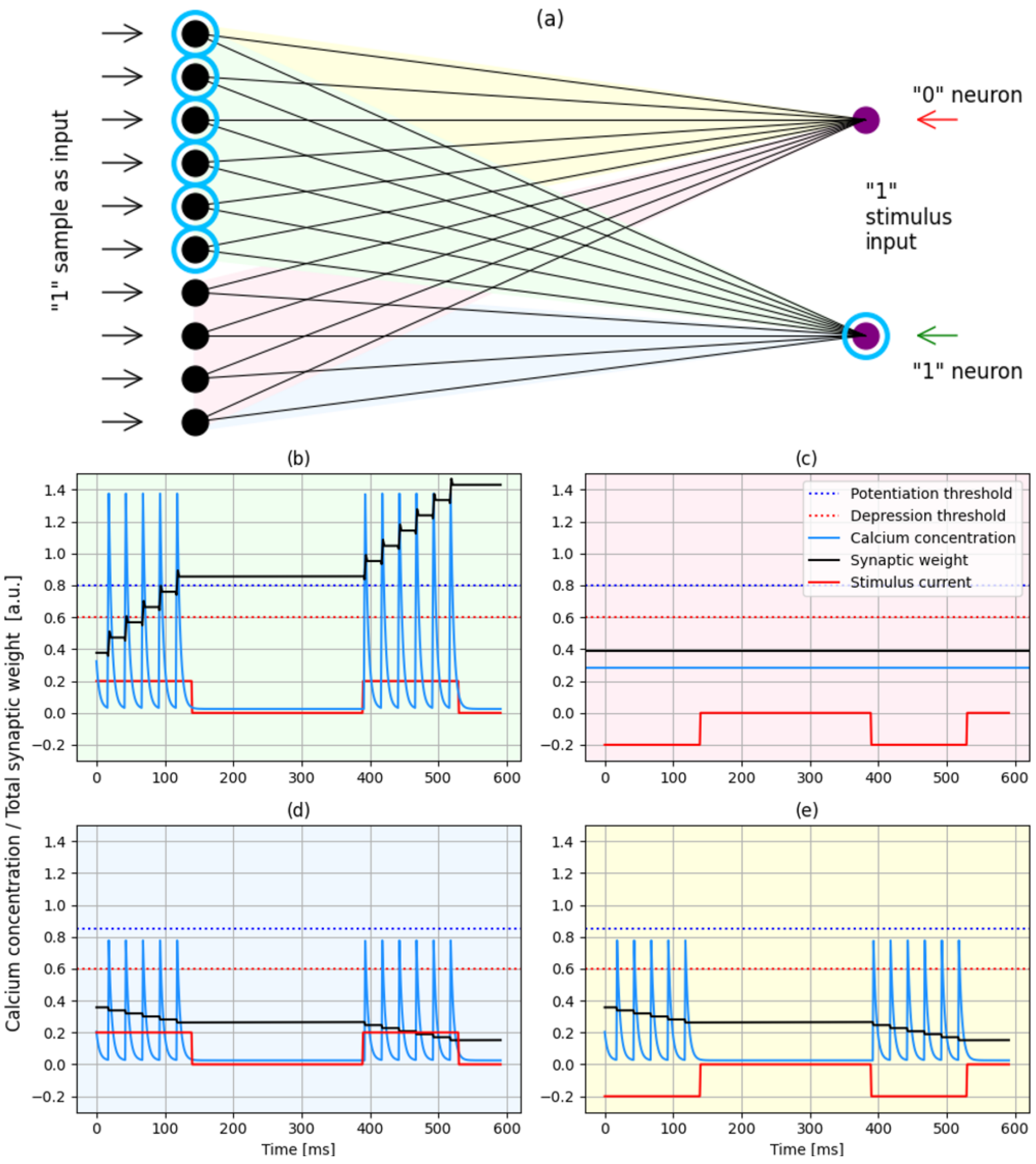


Figure 14: (a) The network architecture, highlighting the four possible groups of synapses. The network is being presented with an example input sample labelled '1'. Blue rings indicate firing neurons, the green arrow indicates a stimulatory current and the red arrow an inhibitory current. (b) The presence of both pre- and postsynaptic spikes at a synapse causes the calcium concentration to surpass the potentiation threshold, leading to an increase in synaptic weight. The solid red line indicates the stimulus input, which can be either positive (stimulatory) or negative (inhibitory). (c) There is no change in synaptic weight if neither pre- nor postsynaptic neurons fire. (d) The presence of *only* postsynaptic spikes causes the calcium threshold to surpass the depression threshold, but not the potentiation threshold, leading to a decrease in synaptic weight. (e) The presence of *only* presynaptic spikes causes the calcium threshold to surpass the depression threshold, but not the potentiation threshold, leading to a decrease in synaptic weight.

To test the performance of the neural network, it was trained on a subset of the MNIST dataset containing 170 samples of zeroes and ones and then tested on a separate subset of 100 samples. During testing, synaptic plasticity was disabled, meaning that the synaptic weights could not change. Testing was performed 200 ms after the last training sample was presented. The accuracy of the network was tested in two scenarios: using the standard early- and late-phase time constants (68.84 s and 360.0 s, respectively); and then using an acceleration factor of 32 for both constants. An accuracy of 96% was achieved in the former case, in which the network was not given time for consolidation: i.e. the EPW and LPW were not given sufficient time to undergo any major weight changes between the end of training and testing. In the latter case, the accelerated time constants allowed the network to consolidate the synaptic weight changes that occurred during training by accelerating the decay of the EPW and the potentiation/depression of the LPW. This produced an accuracy of 93%. The differing levels of accuracy demonstrate the multi-timescale nature of STC, illustrating the behaviour of the network on both the short (standard time constants) and long time-scales (accelerated time constants). Heat maps of the synaptic weights for the standard scenario are plotted in Figure 15.

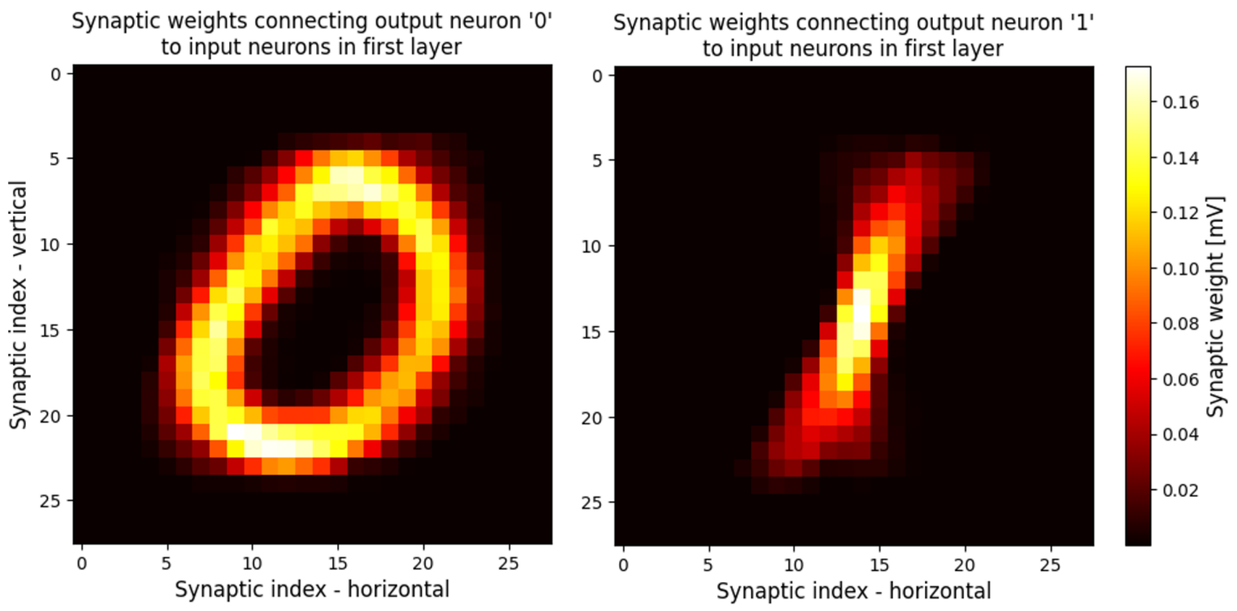


Figure 15: Plot depicting the synaptic weights connecting pre- and postsynaptic neurons after training, but before memory consolidation.

The lowered accuracy for the accelerated time constants can be explained by the drop in total synaptic weight during consolidation, as displayed in Figure 11. Owing to this weight drop, non-zero synaptic weights were on average 9.4% higher in the non-accelerated case. This increased disparity in the weights of connections between a second layer neuron and the first layer neurons helps the network distinguish between the two types of input. Thus, the drop in the total synaptic weight as the network consolidates the changes that occurred during training reduces accuracy, explaining the drop from 96% to 93%.

It is interesting to note that this is the opposite to the behaviour observed in [23], where accuracy improved after consolidation. The explanation for this difference is that synaptic plasticity was disabled during testing in this work, whereas in [23] it was not. When synaptic plasticity is left on, the testing/recall input causes an increase in the EPW. The relative increase in the EPW due to this recall is much larger once consolidation has taken place, as the EPW has decayed to a much lower level than in the case when the recall stimulus is sent directly after training. If recall is performed directly after training, the EPW has not had a chance to decay, and hence, the total synaptic weight increases by a lower percentage than when testing is done after consolidation. This larger increase in total synaptic weight during recall leads to improved accuracy, explaining why the network becomes more accurate after consolidation. Again, it ought to be emphasized that this only happens when synaptic plasticity is left turned on during testing, which in this work it was not, as the evaluation of the influence of synaptic plasticity during testing is beyond the scope of this paper.

4 Discussion

This study demonstrates the compatibility of the CMOS implementation of a synapse, performing the STC learning rule, on a network level. It does this by producing a Python script mimicking the CMOS circuit implementation, and testing the script in a neural network. The validity of the Python script produced was corroborated through comparison of the script to both previous simulations of the synapse circuitry and the differential equations governing STC. The two-layer, feedforward SNN the code was used in achieved an accuracy of 96%, proving the functionality of the synapse circuits on a network level.

It should be noted, however, that the Python script used is not an exact simulation of the CMOS circuits. While the portions of the code responsible for the updates to the calcium concentration and the EPW are derived entirely from the synapse circuitry, the late-phase portion of the code is not. Rather than simulating the circuits, the late-phase portion of the code merely uses the forward Euler method to solve Equation (4), which the late-phase portion of the circuitry is based on. This choice arose from the relative complexity of the late-phase circuitry, which implements late-phase plasticity using a mixed signal approach (both analog and digital), comprising a bump-anti bump circuit [31], two winner-take-all circuits [26, 27], and a control block with a digital counter-oscillator integrator system [12].

To avoid the increase in computational cost the simulation of these components would incur, the code was first written using only Equation (4). The output of the code was then compared to simulations of the CMOS circuits performed in Cadence Virtuoso (Figure 11). Based on these comparisons, it was decided that the behaviour of the code and the circuit simulation matched closely enough, omitting the need to properly replicate the late-phase circuitry in the code. This can be justified further through closer examination of the Cadence Virtuoso simulation producing Figure 11 (b). To reduce simulation times, an 8-bit counter was used in place of the 16-bit counter proposed in the CMOS model, resulting in distinct notches in the slope of the LPW. Had a 16-bit counter been used, the curve would have been smoother, leading to less overestimation of the LPW. This would diminish the prominence of the initial upward slope in the LPW, increasing the similarity of Figure 11 (b) to Figures 11 (a) and (c). Overestimation of the LPW also helps explain why the EPW reaches its maximum value earlier in the Cadence simulation than the Python output, as illustrated in Figure 16.

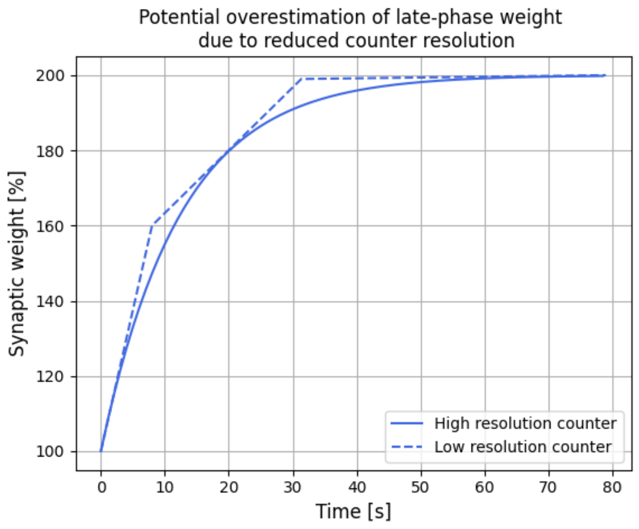


Figure 16: An exaggerated illustration of the potential cause for the larger late-phase contribution in the 8-bit Cadence Virtuoso simulation results in Figure 11(b) relative to the results obtained using the Python script, depicted in Figures 11(a) and (c).

The approach to the late-phase updates is not the only dissimilarity between the circuitry and the Python script aiming to replicate it. Although both the calcium and early-phase portions of the script are based on the CMOS implementation of the synapse, the circuit-based Python code does not simulate the components within the circuits. Instead, the calcium portion of the script implements Equation (6), the differential equation defining the behaviour of the DPI the CMOS synapse uses to calculate the calcium concentration; the code does not replicate each individual transistor and wire the DPI contains, but merely uses the extracted logic to replicate the collective behaviour of all of these components. Likewise, the early-phase portion of the script implements the behaviour of the early-phase circuit, rather than a direct simulation of the components that make up the circuit. Thus, the Python script omits the influence of parasitics owing to the physical characteristics of the circuit layout, such as parasitic capacitances in the various transistors of the circuit [32]. An example of the resulting difference in the Python script and circuit behaviour is illustrated in Figure 17, which depicts the calcium concentration and early-phase response to a post-synaptic spike at 2 ms and a presynaptic spike at 4 ms.

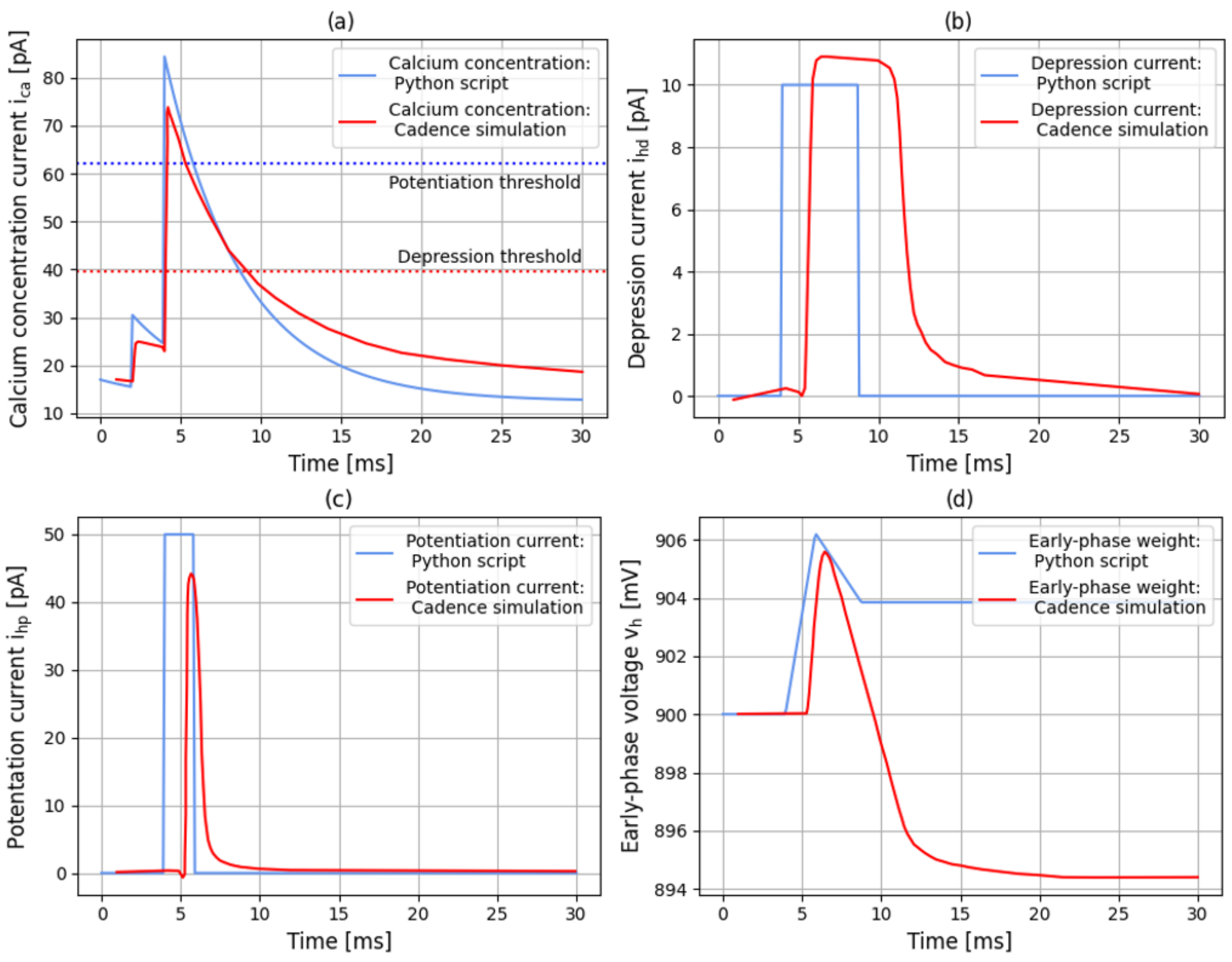


Figure 17: The influence of non-idealities on the behaviour of the synapse circuits. In red is the response of the synapse circuits simulated in Cadence Virtuoso, and in blue is the response of the Python script, using the same parameters and inputs as the Cadence simulation. (a) The response of the calcium concentration to a postsynaptic spike and a presynaptic spike, separated by 2 ms. (b) The depression current resulting from the change in the calcium concentration. (c) The potentiation current resulting from the change in the calcium concentration. (d) The early-phase response to the input, determined by the integration of the depression and potentiation currents. The results of the Cadence Virtuoso simulations are adapted from [12].

Although the general behaviours of both simulations, Python and Cadence, are similar, Figure 17 (d) shows different outcomes in the final EPW. This can be explained by the parasitic effects on the depression and potentiation currents. The calcium concentrations in both the Python script and Cadence simulation spend roughly equal amounts of time above the two thresholds, but despite this, the resulting depression and potentiation currents are not the same. The initiation of each of the two currents in the Cadence simulation exhibits a delay with regards to the calcium concentration crossing their respective thresholds, delaying the changes to the EPW with respect to the Python model. The currents also tail off gradually - owing to the characteristics of the ultra-low leakage blocks [12] used in the early-phase circuitry - rather than the instant drop the circuit logic suggests, extending the time that potentiation and depression are active. However, neither of these observations are sufficient in explaining the far steeper slopes, nor the dramatic depression, of the EPW observed in Figure 17 (d). This hints at the presence of other non-idealities within the early-phase circuitry or variability within the Cadence simulation.

A comprehensive fitting of parameters, combined with updates to the Python script implementing certain parastics such as the delays or current pulse tails, would be possible. It would improve the mapping of the Python script behaviour onto the actual behaviour of the CMOS synapse. However, a full fit achieving this was not performed. The reasons for this are twofold: the parameters used were adequate in producing a functional neural network, and secondly, although the circuits have been taped out and the circuits are being fabricated, a physical version does not yet exist. The calibration of the circuit-based Python code has been scheduled for when the CMOS circuits are available, making it possible to base this calibration on the experimental characterisation of the circuits. This way, the simulations produced by the Python code will account for the influence of all the potential non-idealities the circuits may be subjected to.

Finally, the functionality of the CMOS synapse on a network level is demonstrated using a limited number of samples in the training and testing datasets. This limit was imposed by computing power, with larger datasets exceeding the RAM available. Access to a GPU would be required for a larger scale test, which would substantiate the accuracy results obtained. Larger scale tests are also needed in performing tests on the CMOS synapse's behaviour in biologically plausible networks (which the two-layer feedforward network used was not, owing to its lack of key features such as true inhibitory neurons, background noise, axonal time delay and plausible probabilities for a connection existing between two neurons [11, 30, 33]). However this is outside of the scope of this work, and hence has been left for future research; the small sample sizes used were sufficient in demonstrating this works objective, proving the functionality of the STC circuitry.

5 Conclusion

This work produced a Python code replicating the behaviour of a CMOS implementation of a synapse performing the biologically inspired multi-timescale learning rule STC. The accuracy of this Python code as an emulation of the circuit behaviour was tested through comparisons of the Python code to circuit simulations, performed in Cadence Virtuoso, and to an additional Python script created based on the differential equations governing STC. Once sufficient accuracy was achieved, the circuit-based Python code was used to test the CMOS circuitry on a network level. A two-layer, feedforward neural network was constructed to perform this testing, and the results revealed functionality of the network in distinguishing between MNIST sample ones and zeroes on both the early- and late-phase timescales. This meets the objective of this work, which set out to demonstrate the operation of the CMOS synapse on a network level.

This work is a milestone in a much larger project, and therefore there is a broad scope for future research. A comprehensive fitting of the Python code onto the behaviour of the physical circuits will have to be performed once they are taped out, and then the code will be used to test the operation of the CMOS synapse in a wide variety of applications. Such research could involve: the testing of the CMOS circutis in more biologically plausible neural networks, such as recursive networks; using the networks to perform tasks more suited to the multi-timescale nature of STC, such as in keyword spotting; and ultimately, trying to achieve the long-term goal of implementing the proposed CMOS synapse into real-life edge computing applications.

References

[1] Scott McKinney et al. “International evaluation of an AI system for breast cancer screening”. In: *Nature* 577 (Jan. 2020), pp. 89–94. DOI: 10.1038/s41586-019-1799-6.

[2] Awni Hannun et al. “Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network”. In: *Nature Medicine* 25 (Jan. 2019). DOI: 10.1038/s41591-018-0268-3.

[3] Yikai Yang et al. “A Multimodal AI System for Out-of-Distribution Generalization of Seizure Identification”. In: *IEEE Journal of Biomedical and Health Informatics* PP (Mar. 2022), pp. 1–1. DOI: 10.1109/JBHI.2022.3157877.

[4] Yikai Yang et al. “Weak self-supervised learning for seizure forecasting: a feasibility study”. In: *Royal Society Open Science* 9 (Aug. 2022). DOI: 10.1098/rsos.220374.

[5] Oriol Vinyals et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* 575 (Nov. 2019). DOI: 10.1038/s41586-019-1724-z.

[6] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[7] Jason Eshraghian et al. “Training Spiking Neural Networks Using Lessons From Deep Learning”. In: *Proceedings of the IEEE* PP (Sept. 2023), pp. 1–39. DOI: 10.1109/JPROC.2023.3308088.

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

[9] Yao Man et al. “Spike-based dynamic computing with asynchronous sensing-computing neuromorphic chip”. In: *Nature Communications* 15 (May 2024). DOI: 10.1038/s41467-024-47811-6.

[10] Mahyar Shahsavari et al. “Advancements in spiking neural network communication and synchronization techniques for event-driven neuromorphic systems”. In: *Array* 20 (Oct. 2023), p. 100323. DOI: 10.1016/j.array.2023.100323.

[11] Jannik Luboeinski and Christian Tetzlaff. “Organization and Priming of Long-term Memory Representations with Two-phase Plasticity”. In: *Cognitive Computation* 15 (June 2022), pp. 1–20. DOI: 10.1007/s12559-022-10021-7.

[12] Jorge Quijada Navarro. “CMOS circuit design of a multi-timescale learning rule for spiking neural networks”. MA thesis. Technische Universität Dresden, 2024.

[13] Kai Malcom and Josue Casco-Rodriguez. *A Comprehensive Review of Spiking Neural Networks: Interpretation, Optimization, Efficiency, and Best Practices*. Mar. 2023.

[14] Jim Hsu, Carol Nilsson, and Fernanda Laezza. “Role of the Axonal Initial Segment in Psychiatric Disorders: Function, Dysfunction, and Intervention”. In: *Frontiers in Psychiatry* 5 (Aug. 2014), p. 109. DOI: 10.3389/fpsyt.2014.00109.

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

[16] Atrayee Chatterjee and Souvik Paul. “An approach of curing Depression by Artificial Neural Networks(ANN) Approach with 5 Input Node Reviewing 3 Input Node”. In: *IJCSET* 6 (Dec. 2015), pp. 655–663.

[17] Image courtesy of Tim Taylor.

[18] Jia-Qin Yang et al. “Neuromorphic Engineering: From Biological to Spike-Based Hardware Nervous Systems”. In: *Advanced Materials* 32 (Nov. 2020), p. 2003610. DOI: 10.1002/adma.202003610.

[19] Wulfram Gerstner and Kistler M. Werner. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Aug. 2002. DOI: 10.1017/CBO9780511815706.

[20] Federico Paredes-Vallés, Kirk Scheper, and Guido Croon. *Unsupervised Learning of a Hierarchical Spiking Neural Network for Optical Flow Estimation: From Events to Global Motion Perception*. Mar. 2019.

[21] Alex Prodan and Richard Morris. “Synaptic Tagging and Capture: Functional Implications and Molecular Mechanisms”. In: Apr. 2024, pp. 1–41. ISBN: 978-3-031-54863-5. DOI: 10.1007/978-3-031-54864-2_1.

[22] Raheel Khan, Don Kulasiri, and Sandhya Samarasinghe. “A multifarious exploration of Synaptic Tagging and Capture hypothesis in synaptic plasticity: development of an integrated mathematical model and computational experiments”. In: *Journal of Theoretical Biology* 556 (Oct. 2022), p. 111326. DOI: 10.1016/j.jtbi.2022.111326.

[23] Jannik Luboeinski and Christian Tetzlaff. “Memory consolidation and improvement by synaptic tagging and capture in recurrent neural networks”. In: *Communications Biology* 4 (Mar. 2021). DOI: 10.1038/s42003-021-01778-y.

[24] Chiara Bartolozzi, Srinjoy Mitra, and Giacomo Indiveri. “An ultra low power current-mode filter for neuromorphic systems and biomedical signal processing”. In: Nov. 2006, pp. 130–133. ISBN: 978-1-4244-0436-0. DOI: 10.1109/BIOCAS.2006.4600325.

[25] R. Edwards and Gert Cauwenberghs. “Synthesis of Log-Domain Filters from First-Order Building Blocks”. In: *Analog Integrated Circuits and Signal Processing* 22 (Mar. 2000), pp. 177–186. DOI: 10.1023/A:1008373826094.

[26] John Lazzaro et al. “Winner-Take-All Networks of O(N) Complexity”. In: *Neural Information Processing Systems*. 1988. URL: https://api.semanticscholar.org/CorpusID:1248336.

[27] Giacomo Indiveri. “A Current-Mode Hysteretic Winner-take-all Network, with Excitatory and Inhibitory Coupling”. In: *Analog Integr. Circuits Signal Process* 28 (Sept. 2001). DOI: 10.1023/A:1011208127849.

[28] Shiva Subbulakshmi Radhakrishnan et al. “A biomimetic neural encoder for spiking neural network”. In: *Nature Communications* 12 (Apr. 2021). DOI: 10.1038/s41467-021-22332-8.

[29] Soha Boroojerdi and George Rudolph. “Handwritten Multi-Digit Recognition With Machine Learning”. In: *2022 Intermountain Engineering, Technology and Computing (IETC)*. 2022, pp. 1–6. DOI: 10.1109/IETC54973.2022.9796722.

[30] Peter Diehl and Matthew Cook. “Unsupervised learning of digit recognition using spike-timing-dependent plasticity”. In: *Frontiers in Computational Neuroscience* 9 (2015). ISSN: 1662-5188. DOI: 10.3389/fncom.2015.00099. URL: https://www.frontiersin.org/articles/10.3389/fncom.2015.00099.

[31] Tobi Delbruck. “’Bump’ circuits for computing similarity and dissimilarity of analog voltages”. In: *IJCNN-91-Seattle International Joint Conference on Neural Networks* i (1991), 475–479 vol.1. URL: https://api.semanticscholar.org/CorpusID:16870681.

[32] Sherin A. Thomas et al. “Analysis of Parasitics on CMOS based Memristor Crossbar Array for Neuromorphic Systems”. In: *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2021, pp. 309–312. DOI: `10.1109/MWSCAS47672.2021.9531867`.

[33] Yoshua Bengio et al. “Towards Biologically Plausible Deep Learning”. In: *CoRR* abs/1502.04156 (2015). arXiv: `1502.04156`. URL: `http://arxiv.org/abs/1502.04156`.

Appendices

Appendix A

The complete code used to train and test the SNN implementing the CMOS synapse. A link to the digital version of the code is provided here:

https://github.com/MANicholson/Modelling-and-Simulation-of-a-CMOS-Synapse-Implementing-Two-phase-Plasticity.git

Complete code:

```
# Installs -------------------------------------------------------------------

!pip install snntorch
!pip install equinox

# Imports --------------------------------------------------------------------

import jax
import random
import jax.numpy as jnp
import numpy as np
import matplotlib.pyplot as plt
import equinox as eqx
import snntorch as snn
import torch
import numpy as np
from snntorch import spikegen
import pandas as pd

# Importing data -------------------------------------------------------------
# Mounting drive

from google.colab import drive
drive.mount('/content/drive')

# Importing the data from drive

arr_full_path = '/content/drive/ ... '
df = pd.read_csv(arr_full_path)

arr_test_path = '/content/drive/ ... '
df_test = pd.read_csv(arr_test_path)


# Rate encoding of data ------------------------------------------------------

# Parameters required to produce spike input

spike_length = 150 # Length of time (in ms) each sample is presented to the network
gap_length = 200 # Length of time (in ms) between training samples
Number_synapses = 28*28
dt = 0.001 # Time step
max_spiking_frequency = 40 # Maximum spiking frequency in Hz

# Code to transform datasets into input spike arrays that will be fed to the
# neural network.

def generate_input_spikes(df, spike_length, dt, gap_length, number_synapses):
    def generate_spike_train(value, length, dt):
        frequency = (value / 255) * max_spiking_frequency
        num_spikes = int(frequency * length * dt)
        spike_train = np.zeros(length)

        if num_spikes > 0:
            if num_spikes >= length:
                spike_train[:] = 1
            else:
                interval = length // num_spikes
                for i in range(num_spikes):
                    spike_train[i * interval] = 1

        return spike_train

    def column_to_spike_array(column):
        spike_array = []
        for value in column:
            spike_train = generate_spike_train(value, spike_length, dt)
            spike_array.append(spike_train)
            spike_array.append(np.zeros(gap_length))
        return np.concatenate(spike_array)

    input_spikes = []
    for column in df.columns[1:]:
        spike_array = jnp.array(column_to_spike_array(df[column]))
        input_spikes.append(spike_array)

    input_spikes = jnp.array(input_spikes)
    input_spikes = input_spikes.reshape(number_synapses, 1, input_spikes.shape[1])

    print("Output array shape:", input_spikes.shape)
    return input_spikes

# Producing the training input

INPUT_SPIKES = generate_input_spikes(df, spike_length, dt, gap_length, Number_synapses)

# Producing the test input

INPUT_SPIKES_TEST = generate_input_spikes(df_test, spike_length, dt, gap_length, Number_synapses)

# Creating the stimulus input ---------------------------------------------------

# Parameters required to produce spike input

stimulus_magnitude = 1e-9 # Magnitude of stimulus current (in A)
number_secondary_neurons =   2 # Number of second layer neurons

def create_input_stimuli(df, spike_length, gap_length, stimulus_magnitude,
                         number_secondary_neurons):
    def create_input_stimulus(df, desired_value, stimulus_magnitude):
        result_array = []
        for value in df.iloc[:, 0]:  # Iterate through the first column
            if value == desired_value:
                result_array.extend([stimulus_magnitude] * spike_length)
                result_array.extend([0] * gap_length)
            else:
                result_array.extend([-stimulus_magnitude] * spike_length)
                result_array.extend([0] * gap_length)
        return np.array(result_array)

    stimuli = []
    for i in range(number_secondary_neurons):
        stim = jnp.array([create_input_stimulus(df, i, stimulus_magnitude)])
        stimuli.append(stim)

    return stimuli

# Producing the stimulus input
stimuli = create_input_stimuli(df, spike_length, gap_length, stimulus_magnitude,
                               number_secondary_neurons)

# Base code for the neural network ----------------------------------------------

#

class many_synapse(eqx.Module):
    tau_DPI: float
    dt: float
    delta_capre: float
    delta_capost: float
    I_TH: float
    I_INDC: float
    I_TAU: float
    ica_0: float
    C: float
    v_H0: float
    I_THPOT: float
    I_THDEP: float
    I_TAILP: float
    I_TAILP_low: float
    I_TAILD: float
    I_TAILD_low: float
    i_hrn: float
    i_hrp: float
    v_h0: float
    V_DD: float
    tau_z_c: float
    z_max: float
    z_min: float
    theta_tag_c: float
    alpha: float
    theta_pro_circuit: float
    z_0: float
    beta: float
    reset_voltage: float
    V_rev: float
    V_spike_threshold: float
    tau_m: float
    R_m: float
    refractory_period: float
    h_0: float = 0.0

    def __call__(self, t_spike_pre_array, I_stim):
        num_input_neurons = t_spike_pre_array.shape[0]

        # Initial states
        i_ca = jnp.zeros((num_input_neurons, 1)) + self.ica_0
        v_h = jnp.zeros((num_input_neurons, 1)) + self.v_h0
        z_ji = jnp.zeros((num_input_neurons, 1)) + self.z_0
        w_ji = jnp.zeros((num_input_neurons, 1)) + self.beta * self.v_h0 + self.beta * self.v_h0 * self.z_0
        V_i = jnp.zeros((1,)) + self.V_rev
        p_i = jnp.zeros((num_input_neurons, 1))
        t_spike_post = 0.0  # Initialize t_spike_post to 0
        refractory_timer = jnp.zeros((1,))

        # Lists to store the results
        V_i_list = []
        t_spike_post_list = []

        def step(carry, input_):
            t_spike_post, i_ca, v_h, z_ji, p_i, w_ji, V_i, refractory_timer = carry
            t_spike_pre, I_stim = input_

            # Code for the neurons in the second layer:..............................

            in_refractory = jnp.squeeze(refractory_timer > 0)
            V_i = jax.lax.cond(in_refractory, lambda _: jnp.array([self.reset_voltage]), lambda _: V_i, operand=None)

            inhibition_condition = jnp.squeeze(jnp.sum(I_stim) < 0)


            reversal_term = (1 / self.tau_m) * (-V_i + self.V_rev)
            pre_spike_contribution = (1 / self.tau_m) * jnp.sum(t_spike_pre * w_ji)
            current_term = (1 / self.tau_m) * jnp.sum((self.R_m * I_stim))

            spike_condition = jnp.squeeze(V_i > self.V_spike_threshold)

            V_i_new = jax.lax.cond(spike_condition, lambda _:
                                   jnp.array([self.reset_voltage]),
                                   lambda _: V_i + (reversal_term + current_term)
                                   * self.dt + pre_spike_contribution, operand=None)
            # For inhibition
            V_i_new = jax.lax.cond(inhibition_condition, lambda _:
                                   jnp.array([self.reset_voltage]),
                                   lambda _: V_i + (reversal_term + current_term)
                                    * self.dt + pre_spike_contribution, operand=None)

            # Update t_spike_post based on spike condition
            t_spike_post_new = jax.lax.cond(spike_condition, lambda _: 1.0,
                                            lambda _: 0.0, operand=None)
            t_spike_post_list.append(t_spike_post_new)

            # To enforce the refractory period
            refractory_timer_new = jax.lax.cond(spike_condition, lambda _:
                                                jnp.array([self.refractory_period]),
                                                lambda _:
                                                jax.numpy.maximum(refractory_timer - self.dt, 0),
                                                operand=None)

            # Calcium part -------------------------------------------------------
            dica_dt = (1 / self.tau_DPI) * (-i_ca + self.I_TH * self.I_INDC / self.I_TAU)
            ica_pre_increase = t_spike_pre * self.delta_capre
            ica_post_increase = t_spike_post_new * self.delta_capost
            i_ca_new = i_ca + dica_dt * self.dt + ica_pre_increase.T + ica_post_increase


            # Early-phase part ---------------------------------------------------
            recovery_condition = jnp.squeeze(v_h > self.v_H0)
            potentiation_condition = jnp.squeeze(i_ca_new > self.I_THPOT)
            depression_condition = jnp.squeeze(i_ca_new > self.I_THDEP)

            i_recovery = jnp.array([jnp.where(recovery_condition, - self.i_hrn, self.i_hrp)])

            i_potentiation = jnp.array([jnp.where(potentiation_condition, self.I_TAILP, self.I_TAILP_low)])

            i_depression = jnp.array([jnp.where(depression_condition, - self.I_TAILD, self.I_TAILD_low)])

            v_h_new = v_h + (1 / self.C) * (i_recovery.T + i_potentiation.T + i_depression.T) * self.dt

            # Ensure v_h_new is not less than 0
            v_h_new = jnp.maximum(v_h_new, 0)

            # Late-phase part ----------------------------------------------------
            epsilon_hi = jnp.abs(v_h_new - self.v_h0)
            condition = jnp.array([jnp.squeeze(epsilon_hi > self.theta_pro_circuit)])

            p_i = jnp.where(condition.T, self.alpha, p_i)
            pot_term = p_i * (self.z_max - z_ji) * ((v_h_new - self.v_h0 - self.theta_tag_c) > 0)
            dep_term = p_i * (z_ji - self.z_min) * ((self.h_0 - v_h_new - self.theta_tag_c) > 0)
            dz_dt = (1 / self.tau_z_c) * (pot_term - dep_term)
            z_ji_new = z_ji + dz_dt * self.dt

            # Ensure z_ji_new is not less than 0
            z_ji_new = jnp.maximum(z_ji_new, 0)

            # Total synaptic weight ----------------------------------------------
            w_ji_new = self.beta * v_h_new + self.beta * self.v_h0 * z_ji_new

            V_i_list.append(V_i_new)

            carry = (t_spike_post_new, i_ca_new, v_h_new, z_ji_new, p_i,
                     w_ji_new, V_i_new, refractory_timer_new)
            return carry, (i_ca_new, v_h_new, z_ji_new, w_ji_new,
                           V_i_new, t_spike_post_new)

        carry = (t_spike_post, i_ca, v_h, z_ji, p_i, w_ji, V_i, refractory_timer)
        _, (i_ca, v_h, z_ji, w_ji, V_i, t_spike_post) = jax.lax.scan(step, carry, (t_spike_pre_array.T, I_stim.T))
        time_array = jnp.arange(len((i_ca.T)[0][0])) * self.dt

        return i_ca.T, v_h.T, z_ji.T, w_ji.T, jnp.concatenate(V_i), t_spike_post, time_array

# Constants for differential code

# tc_delay = 0.0188  # Delay of postsynaptic calcium influx after presynaptic spike [s]
c_pre = 0.6  # Presynaptic calcium contribution [in vivo adjusted]
c_post = 0.1655  # Postsynaptic calcium contribution [in vivo adjusted]
tau_c = 0.0488  # Calcium time constant [s]
tau_h = 688.4 # ALTERED FROM 688.4  # Early-phase time constant [s]
tau_p = 60*60  # Protein time constant [s]
tau_z = 60*60  # Late-phase time constant [s]
# nu_th = 40  # Firing rate threshold for LTP induction [Hz]
gamma_p = 1645.6 # ALTERED FROM 1645.6  # Potentiation rate
gamma_d = 313.1 # ALTERED FROM 313.1  # Depression rate
theta_p = 0.003 # ALTERED FROM 3  # Calcium threshold for potentiation
theta_d = 0.002 # ALTERED FROM 1.2  # Calcium threshold for depression
# sigma_pl = 2.90436 * 10**(-3)  # Standard deviation for plasticity fluctuations [V]
alpha = 1.0  # Protein synthesis rate
theta_pro = 0.003 #2.10037  * 10**(-3)  # Protein synthesis threshold [V]
theta_tag = 0.840149  * 10**(-3)  # Tagging threshold [V]
h_0 = 4.20075  * 10**(-3) # Median initial excitatory→excitatory coupling strength [V]
h_init = h_0
z_0 = 0.01
# Combining [12] and [11]

h_max = 10 * 10**(-3) # The maximum value of the early-phase variable [V]
z_max = 1 # The minimum value of the late-phase variable
z_min = - 0.5 # The maximum value of the late-phase variable

beta = 4.6675*10**(-3)

# Circuit-based code constants

# Calcium:
I_INDC = 25 * 10**(-12)
i_pre = 800 * 10**(-12)
i_post = 200 * 10**(-12)
tau_DPI = 0.00488
I_TH = 10 * 10**(-12)
I_TAU = 20 * 10**(-12)
ica_0 = 17 * 10**(-12)
delta_capre = 15 * 10**(-12)
delta_capost = 15 * 10**(-12)


# Early-phase:
tau_h_c = 68.84 # [s]
C = 1.2215 * 10**(-12)
v_H0 = 0.9
I_THPOT = 30 * 10**(-12)
I_THDEP = 25* 10**(-12)
I_TAILP = 50 * 10**(-12)
I_TAILP_low = 1.2 * 10**(-15)
I_TAILD = 10 * 10**(-12)
I_TAILD_low = 1.2 * 10**(-15)
i_hrp = 2.5 * 10**(-15)
i_hrn = 80 * 10**(-15)
v_h0 = 900 * 10**(-3)
V_DD = 1.8
z_0 = 0.0
# Late phase:
theta_tag_c = 0.0151226
theta_pro_circuit = 0.01
tau_z_c = 11.25
tau_p_c = 360

# Neuron constants

tau_m = 0.01
```

```
V_rev = -0.065 #
R_m = 10e6
reset_voltage = -0.07
V_spike_threshold = 70
refractory_period = 0.0001
dt = 0.001

# Training the network ------------------------------------------------------------

manySynapse = many_synapse(
    tau_DPI, dt, delta_capre, delta_capost, I_TH, I_INDC, I_TAU, ica_0,
    C, v_H0, I_THPOT, I_THDEP, I_TAILP, I_TAILP_low, I_TAILD, I_TAILD_low, i_hrn, i_hrp,
    v_h0, V_DD, tau_z_c, z_max, z_min, theta_tag_c, alpha, theta_pro_circuit,
    z_0, beta, reset_voltage, V_rev, V_spike_threshold, tau_m, R_m, refractory_period
)

results = []

# Creating arrays with the final weights and plotting synaptic weights

for i, stim in enumerate(stimuli):
    results.append(manySynapse(INPUT_SPIKES, stim))

final_weights = []
for i in range(number_secondary_neurons):
    w_ji = results[i][3]  # Assuming w_ji is the 4th returned value
    final_values = w_ji[-1, :, -1]
    final_weights.append(final_values)

print(len(final_weights))

# Plotting heatmaps
fig, axs = plt.subplots(1, number_secondary_neurons, figsize=(15, 5))

for i, weights in enumerate(final_weights):
    square_grid = weights.reshape((28, 28))

    im = axs[i].imshow(square_grid, cmap='hot', interpolation='nearest')
    axs[i].set_title(f'Synaptic weights connecting output neuron \'{i}\'\nto input
                                                    neurons in first layer')
    plt.colorbar(im)
    axs[1].set_ylabel('Synaptic weight [mV]', fontsize = 12)
    axs[i].set_xlabel('Synaptic index - horizontal', fontsize = 12)
    axs[0].set_ylabel('Synaptic index - vertical', fontsize = 12)



plt.show()
plt.tight_layout()

# Testing the network -------------------------------------------------------------

tau_m = 0.01
V_rev = -0.065 #
R_m = 10e6
reset_voltage = -0.07
V_spike_threshold = 70
refractory_period = 0.0001
dt = 0.001

class test_Network(eqx.Module):
    tau_DPI: float
    dt: float
    delta_capre: float
    delta_capost: float
    I_TH: float
    I_INDC: float
    I_TAU: float
    ica_0: float
    C: float
    v_H0: float
    I_THPOT: float
    I_THDEP: float
    I_TAILP: float
    I_TAILP_low: float
    I_TAILD: float
    I_TAILD_low: float
    i_hrn: float
    i_hrp: float
    v_h0: float
    V_DD: float
    tau_z_c: float
    z_max: float
    z_min: float
    theta_tag_c: float
    alpha: float
    theta_pro_circuit: float
    z_0: float
    beta: float
    reset_voltage: float
    V_rev: float
    V_spike_threshold: float
    tau_m: float
    R_m: float
    refractory_period: float
    h_0: float = 0.0

    def __call__(self, t_spike_pre_array, w_ji):
        num_input_neurons = t_spike_pre_array.shape[0]

        # Initial states
        V_i = jnp.zeros((1,)) + self.V_rev
        t_spike_post = 0.0  # Initialize t_spike_post to 0
        refractory_timer = jnp.zeros((1,))
        W_ji = w_ji.T

        # Lists to store the results
        V_i_list = []
        t_spike_post_list = []

        def step(carry, input_):
            t_spike_post, V_i, refractory_timer, w_ji = carry
            t_spike_pre = input_

            # Code for the neurons in the second layer:.............................

            in_refractory = jnp.squeeze(refractory_timer > 0)
            V_i = jax.lax.cond(in_refractory, lambda _: jnp.array([self.reset_voltage]),
                               lambda _: V_i, operand=None)

            reversal_term = (1 / self.tau_m) * (-V_i + self.V_rev)
            pre_spike_contribution = (1 / self.tau_m) * jnp.sum(t_spike_pre * W_ji)
            # print(t_spike_pre)
            # print(w_ji)
            # print(jnp.sum(t_spike_pre * W_ji))

            spike_condition = jnp.squeeze(V_i > self.V_spike_threshold)

            V_i_new = jax.lax.cond(spike_condition, lambda _: jnp.array([self.reset_voltage]),
                                   lambda _:
                                   V_i + (reversal_term) * self.dt + pre_spike_contribution,
                                   operand=None)

            # Update t_spike_post based on spike condition
            t_spike_post_new = jax.lax.cond(spike_condition, lambda _: 1.0, lambda _: 0.0,
                                            operand=None)
            t_spike_post_list.append(t_spike_post_new)

            refractory_timer_new = jax.lax.cond(spike_condition, lambda _:
                                                jnp.array([self.refractory_period]),
                                                lambda _:
                                                jax.numpy.maximum(refractory_timer - self.dt, 0),
                                                         operand=None)

            V_i_list.append(V_i_new)

            carry = (t_spike_post_new, V_i_new, refractory_timer_new, w_ji)

            return carry, (V_i_new, t_spike_post_new)

        carry = (t_spike_post, V_i, refractory_timer, w_ji)
        _, (V_i, t_spike_post) = jax.lax.scan(step, carry, (t_spike_pre_array.T))

        time_array = jnp.arange(len((t_spike_post))) * self.dt

        return V_i, t_spike_post, time_array

# Running the test

network_test = test_Network(
    tau_DPI, dt, delta_capre, delta_capost, I_TH, I_INDC, I_TAU, ica_0,
    C, v_H0, I_THPOT, I_THDEP, I_TAILP, I_TAILP_low, I_TAILD, I_TAILD_low, i_hrn, i_hrp,
    v_h0, V_DD, tau_z_c, z_max, z_min, theta_tag_c, alpha, theta_pro_circuit, z_0, beta,
    reset_voltage, V_rev, V_spike_threshold, tau_m, R_m, refractory_period
)

number_secondary_neurons = len(final_weights)
number_test_samples = len(df_test)
print("number test samples:", number_test_samples)

# Initialize results_matrix with appropriate dimensions
results_matrix = np.zeros((number_secondary_neurons, number_test_samples))

# Run network_test for each final weight
for neuron_idx, final_weight in enumerate(final_weights):
    V_i, t_spike_post, _ = network_test(INPUT_SPIKES_TEST, final_weight/4)

    # Divide t_spike_post into subarrays
    subarray_length = len(t_spike_post) // number_test_samples
    t_spike_post_subarrays = np.array_split(t_spike_post, number_test_samples)

    for row_idx, subarray in enumerate(t_spike_post_subarrays):
        results_matrix[neuron_idx, row_idx] = jnp.sum(subarray)

# Find the secondary neuron that fired the most for each row
most_firing_neurons = np.argmax(results_matrix, axis=0)

print(len(most_firing_neurons)) # Verify length of most_firing_neurons

# Compare with the actual labels
correct_predictions = 0
for row_idx, row in df_test.iterrows():
    true_label = row[0]
    predicted_label = most_firing_neurons[row_idx]
    if predicted_label == true_label:
        correct_predictions += 1

print(most_firing_neurons) # Print most_firing_neurons for debugging
print(correct_predictions) # Print correct_predictions for debugging

# Calculate and print accuracy
accuracy = correct_predictions / number_test_samples
print(f"Accuracy: {accuracy * 100:.2f}%")


```

Appendix B

The differential equation-based code used to train test whether the circuit-based code performed STC. A link to the digital version of the code is provided here:

https://github.com/MANicholson/Modelling-and-Simulation-of-a-CMOS-Synapse-Implementing-Two-phase-Plasticity.git

The code:

```
# Installs --------------------------------------------------------------------
!pip install equinox

# Imports ---------------------------------------------------------------------
import jax
import random
import jax.numpy as jnp
import matplotlib.pyplot as plt
import equinox as eqx

# Differential equations solved using the forward Euler method -----------------

class full_synapse_diff(eqx.Module):
    # Calcium constants -------------------------------------------------------
    c_pre: float
    c_post: float
    tau_c: float
    dt: float
    c_0: float
    # Early-phase constants ---------------------------------------------------
    tau_h: float
    h_max: float
    gamma_p: float
    gamma_d: float
    theta_p: float
    theta_d: float
    h_0: float
    h_init: float
    # Late-phase constants ----------------------------------------------------
    tau_z: float
    z_max: float
    z_min: float
    theta_tag: float
    alpha: float
    theta_pro: float
    z_0: float

    def __init__(self, c_pre, c_post, tau_c, dt, c_0, tau_h, h_max, gamma_p,
                 gamma_d, theta_p, theta_d, h_0, h_init, tau_z, z_max, z_min,
                 theta_tag, alpha, theta_pro, z_0):
        # Calcium constants ---------------------------------------------------
        self.c_pre = c_pre
        self.c_post = c_post
        self.tau_c = tau_c
        self.dt = dt
        self.c_0 = c_0
        # Early-phase constants -----------------------------------------------
        self.tau_h = tau_h
        self.h_max = h_max
        self.gamma_p = gamma_p
        self.gamma_d = gamma_d
        self.theta_p = theta_p
        self.theta_d = theta_d
        self.h_0 = h_0
        self.h_init = h_init
        # Late-phase constants ------------------------------------------------
        self.tau_z = tau_z
        self.z_max = z_max
        self.z_min = z_min
        self.theta_tag = theta_tag
        self.alpha = alpha
        self.theta_pro = theta_pro
        self.z_0 = z_0

    def __call__(self, input_):

        # Calcium part --------------------------------------------------------
        c_ji_0 = jnp.zeros((1,)) + self.c_0  # Initial state

        def differential_eq_calcium(c_ji, input_):
            t_spike_pre, t_spike_post = input_

            dc_dt = -c_ji / self.tau_c
            c_pre_increase = t_spike_pre * self.c_pre
            c_post_increase = t_spike_post * self.c_post

            c_ji_new = c_ji + dc_dt * self.dt + c_pre_increase + c_post_increase

            return c_ji_new, c_ji_new

        _, c_ji = jax.lax.scan(differential_eq_calcium, c_ji_0, input_)

        time_array = jnp.arange(len(c_ji)) * self.dt

        # Early-phase part ----------------------------------------------------
        if self.h_init is None:
            self.h_init = self.h_0

        h_ji_0 = jnp.zeros((1,)) + self.h_init

        def differential_eq_early_phase(h_ji, c_ji):
            # Early-phase weight decay:
            decay_term = (1 / self.tau_h) * 0.1 * (self.h_0 - h_ji)

            # Early-phase LTP (potentiation occurs when the calcium concentration
            # surpasses the specified threshold)
            ltp_term = (1 / self.tau_h) * self.gamma_p * (self.h_max - h_ji) *
                                                        (c_ji > self.theta_p)

            # Early-phase LTD (depression occurs when the calcium concentration
            # surpasses the specified threshold)
            ltd_term = (1 / self.tau_h) * self.gamma_d * h_ji * (c_ji > self.theta_d)

            # Combine terms
            dh_dt = decay_term + ltp_term - ltd_term

            # Update the early-phase variable using Euler's method
            h_ji_new = h_ji + dh_dt * self.dt

            return h_ji_new, h_ji_new

        _, h_ji = jax.lax.scan(differential_eq_early_phase, h_ji_0, c_ji)

        # Late-phase part -----------------------------------------------------
        z_ji_0 = jnp.zeros((1,)) + self.z_0

        # Set the protein availability to 0:
        p_i = 0.0

        def differential_eq_late_phase(carry, input_):
            z_ji, p_i = carry
            h_ji, c_ji = input_

            # Early-phase change of neuron i
            epsilon_hi = jnp.abs(h_ji - self.h_0)

            # Check if the early-phase change is sufficient to trigger protein synthesis:
            condition = jnp.squeeze(epsilon_hi > self.theta_pro)
            p_i = jax.lax.cond(condition, lambda _: self.alpha, lambda _: p_i, operand=None)

            # Calculate the change in late-phase variable based on the differential equation.

            # Potentiation contribution
            pot_term = p_i * (self.z_max - z_ji) * ((h_ji - self.h_0 - self.theta_tag) > 0)

            # Depression contribution
            dep_term = p_i * (z_ji - self.z_min) * ((self.h_0 - h_ji - self.theta_tag) > 0)

            # Combine terms to get the total change
            dz_dt = (1 / self.tau_z) * (pot_term - dep_term)

            # Update late-phase variable using Euler's method
            z_ji_new = z_ji + dz_dt * self.dt

            return (z_ji_new, p_i), z_ji_new

        carry = (z_ji_0, p_i)
        inputs = (h_ji, c_ji)
        _, z_ji = jax.lax.scan(differential_eq_late_phase, carry, inputs)

        # Calculate the total synaptic weight ----------------------------------

        w_ji_0 = jnp.zeros((1,)) + h_ji_0 + h_ji_0 * z_ji_0


        def total_synaptic_weight(w_ji, Inputs_):
            h_ji, z_ji = Inputs_

            w_ji_new = h_ji + h_ji_0 * z_ji

            return w_ji_new, w_ji_new


        Inputs_ = (h_ji, z_ji)
        _, w_ji = jax.lax.scan(total_synaptic_weight, w_ji_0, Inputs_)

        return c_ji, h_ji, z_ji, w_ji, time_array

# Testing ------------------------------------------------------------------

# Defining constants:

# tc_delay = 0.0188  # Delay of postsynaptic calcium influx after presynaptic spike [s]
c_pre = 0.6  # Presynaptic calcium contribution [in vivo adjusted]
c_post = 0.1655  # Postsynaptic calcium contribution [in vivo adjusted]
tau_c = 0.0488  # Calcium time constant [s]
tau_h = 688.4 # ALTERED FROM 688.4  # Early-phase time constant [s]
tau_p = 60*60  # Protein time constant [s]
tau_z = 60*60  # Late-phase time constant [s]
# nu_th = 40  # Firing rate threshold for LTP induction [Hz]
gamma_p = 1645.6 # ALTERED FROM 1645.6  # Potentiation rate
gamma_d = 313.1 # ALTERED FROM 313.1  # Depression rate
theta_p = 0.3 # ALTERED FROM 3  # Calcium threshold for potentiation
theta_d = 0.2 # ALTERED FROM 1.2  # Calcium threshold for depression
# sigma_pl = 2.90436 * 10**(-3)  # Standard deviation for plasticity fluctuations [V]
alpha = 1.0  # Protein synthesis rate
theta_pro = 0.0023 #2.10037  * 10**(-3)  # Protein synthesis threshold [V]
theta_tag = 0.640149  * 10**(-4)  # Tagging threshold [V]
h_0 = 4.20075  * 10**(-3) # Median initial excitatory→excitatory coupling strength [V]
h_init = h_0
z_0 = 0.1

# Constants required to combine results from following the logic of the circuits in
# "CMOS circuit design of a multi-timescale learning rule for spiking neural
# networks" by Jorge Navarro Quijada
# and the results obtained using the differential equations in
# https://doi.org/10.1007/s12559-022-10021-7

h_max = 10 * 10**(-3) # The maximum value of the early-phase variable [V]
z_max = 1 # The minimum value of the late-phase variable
z_min = - 0.5 # The maximum value of the late-phase variable

# beta = 4.6675*10**(-3)

# Creating test input:

import jax
import jax.numpy as jnp
```

```python
import jax.random as random

def generate_binary_array(length, ratio, key):
    """
    Generate a binary array of 0s and 1s with a specific ratio of 1s to 0s.

    Args:
        length (int): Length of the array.
        ratio (float): Desired ratio of 1s to 0s.
        key (jax.random.PRNGKey): Random key for JAX's random number generator.

    Returns:
        jax.numpy.ndarray: Array of 0s and 1s with the specified ratio.
    """
    num_ones = int(length * ratio)
    num_zeros = length - num_ones

    # Create an array with the specified number of 1s and 0s
    binary_array = jnp.array([1] * num_ones + [0] * num_zeros)

    # Shuffle the array to randomize the distribution of 1s and 0s
    shuffled_array = random.permutation(key, binary_array)

    return shuffled_array

# Example usage
key = random.PRNGKey(0)
length = 100  # Length of the array
ratio = 0.1   # Desired ratio of 1s to 0s

pre_spikes =  [0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,

post_spikes = [0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,

# Add 300 zeros to the end of the array
post_spikes.extend([0] )
pre_spikes.extend([0])

print(post_spikes)
print(pre_spikes)
print(f"Length of post_spikes: {len(post_spikes)}")

t_array = jnp.linspace(0, 10, len(pre_spikes))
dt = 0.01

t_array_c = jnp.linspace(0, 50, len(pre_spikes))
dt_c = 0.001

# Initialize time array and concentration array.
t_max = t_array[-1]
t = jnp.arange(0, t_max + dt, dt)

# Map presynaptic spike times onto new t array to ensure the spikes on a multiple of dt
t_spike_pre_new = jnp.zeros_like(t)
for i, time_point in enumerate(t_array):
    # Find the index of the closest time point in 't'
    closest_idx = jnp.argmin(jnp.abs(t - time_point))
    # Assign the value from 'i_in' to the corresponding location in 'i_new'
    t_spike_pre_new = t_spike_pre_new.at[closest_idx].set(pre_spikes[i])


# Map postsynaptic spike times onto new t array to ensure the spikes on a multiple of dt
t_spike_post_new = jnp.zeros_like(t)
for i, time_point in enumerate(t_array):
    # Find the index of the closest time point in 't'
    closest_idx = jnp.argmin(jnp.abs(t - time_point))
    # Assign the value from 'i_in' to the corresponding location in 'i_new'
    t_spike_post_new = t_spike_post_new.at[closest_idx].set(post_spikes[i])

# Running the code ------------------------------------------------------------

```

```
synapseModel = full_synapse_diff(c_pre, c_post, tau_c, dt, 0, tau_h, h_max,
                                 gamma_p, gamma_d, theta_p, theta_d, h_0, h_init,
                                 tau_z, z_max, z_min, theta_tag, alpha, theta_pro, z_0)

c_ji_eqx, h_ji_eqx, z_ji_eqx, w_ji_eqx, t_eqx = synapseModel((jnp.array(t_spike_pre_new),
                                                             jnp.array(t_spike_post_new)))
```

Appendix C

The constants used in the creation of Figure 9. The first table presents the constants used in the circuit-based code, and the second table presents the constants used in the differential equation-based code.

Variable	**Value**	**Unit**	**Description**
I_INDC	25×10^{-12}	A	The DC component of the tail current to stimulate the DPI.
τ_{DPI}	0.00488	s	The time constant of the DPI.
I_TH	10×10^{-12}	A	The current to tune the gain of the DPI.
I_TAU	20×10^{-12}	A	The current to tune the time constant of the DPI.
delta_capre	60×10^{-12}	A	The contribution of a presynaptic spike to the calcium concentration current.
delta_capost	15×10^{-12}	A	The contribution of a postsynaptic spike to the calcium concentration current.
τ_{h_c}	68.84	s	The early-phase time constant.
C	1.2215×10^{-12}	C	The capacitance of the capacitor.
v_H0	0.9	V	The reference value of the EPW voltage.
I_THPOT	62×10^{-12}	A	The early-phase potentiation threshold for the calcium concentration.
I_THDEP	55×10^{-12}	A	The early-phase depression threshold for the calcium concentration.
I_TAILP	90×10^{-12}	A	The magnitude of the HIGH potentiation current.
I_TAILP_low	1.2×10^{-15}	A	The magnitude of the LOW potentiation current.
I_TAILD	10×10^{-12}	A	The magnitude of the HIGH depression current.
I_TAILD_low	0.8×10^{-15}	A	The magnitude of the LOW depression current.
i_hrp	2.5×10^{-15}	A	The magnitude of the potentiating-recovery current.
i_hrn	2.5×10^{-15}	A	The magnitude of the depressing-recovery current.
V_DD	1.8	V	The maximum value of the EPW voltage.
theta_tag_c	0.0151226	A	The tagging threshold for a synaptic tag to be formed.
theta_pro_circuit	0.45	A	The protein capture threshold, for protein capture at a synapse to occur.
τ_{z_c}	360	s	The late-phase time constant.
beta	4.6675×10^{-3}	n.a.	The factor used to match the outputs from portion of the code based on the circuits to the outputs based on the late-phase differential equation.

Table 1: Circuit-based Python script constants used in Figure 9. These are adapted from [12].

Variable	**Value**	**Unit**	**Description**
c_pre	0.6	mM	Presynaptic calcium contribution.
c_post	0.1655	mM	Postsynaptic calcium contribution.
τ_c	0.0488	s	Calcium time constant.
τ_h	688.4	s	Early-phase time constant.
τ_p	3600	s	Protein time constant.
τ_z	3600	s	Late-phase time constant.
gamma_p	1645.6	n.a.	Potentiation rate.
gamma_d	313.1	n.a.	Depression rate.
theta_p	0.3	mM	Calcium threshold for potentiation.
theta_d	0.2	mM	Calcium threshold for depression.
alpha	1.0	n.a.	Protein synthesis rate.
theta_pro	0.0023	V	Protein synthesis threshold.
theta_tag	0.640149×10^{-4}	V	Tagging threshold.
h_0	4.20075×10^{-3}	V	Median initial excitatory-excitatory coupling strength.
h_max	10×10^{-3}	V	The maximum value of the early-phase variable.
z_max	1	V	The maximum value of the late-phase variable.
z_min	-0.5	V	The minimum value of the late-phase variable.

Table 2: Differential equation-based Python script constants used in Figure 9. These are adapted from [11].

Appendix D

The constants used in the training and testing of the neural network:

Variable	**Value**	**Unit**	**Description**
I_INDC	25×10^{-12}	A	The DC component of the tail current to stimulate the DPI.
τ_{DPI}	0.00488	s	The time constant of the DPI.
I_TH	10×10^{-12}	A	The current to tune the gain of the DPI.
I_TAU	20×10^{-12}	A	The current to tune the time constant of the DPI.
ica_0	17×10^{-12}	A	The initial calcium concentration current.
delta_capre	15×10^{-12}	A	The contribution of a presynaptic spike to the calcium concentration current.
delta_capost	15×10^{-12}	A	The contribution of a postsynaptic spike to the calcium concentration current.
τ_{h_c}	68.84	s	The early-phase time constant.
C	1.2215×10^{-12}	C	The capacitance of the capacitor.
v_H0	0.9	V	The reference value of the EPW voltage.
I_THPOT	30×10^{-12}	A	The early-phase potentiation threshold for the calcium concentration.
I_THDEP	25×10^{-12}	A	The early-phase depression threshold for the calcium concentration.
I_TAILP	50×10^{-12}	A	The magnitude of the HIGH potentiation current.
I_TAILP_low	1.2×10^{-15}	A	The magnitude of the LOW potentiation current.
I_TAILD	10×10^{-12}	A	The magnitude of the HIGH depression current.
I_TAILD_low	1.2×10^{-15}	A	The magnitude of the LOW depression current.
i_hrp	2.5×10^{-15}	A	The magnitude of the potentiating-recovery current.
i_hrn	80×10^{-15}	A	The magnitude of the depressing-recovery current.
v_h0	900×10^{-3}	V	The initial value of the EPW voltage.
V_DD	1.8	V	The maximum value of the EPW voltage.
theta_tag_c	0.0151226	A	The tagging threshold for a synaptic tag to be formed.
theta_pro_circuit	0.02	A	The protein capture threshold, for protein capture at a synapse to occur.
τ_{z_c}	360	s	The late-phase time constant.
z_0	0.0	V	The initial late-phase value.
τ_m	0.01	s	The neuron membrane time constant.
V_rev	-0.065	V	The reversal voltage.
R_m	10×10^{6}	Ω	The resistance used in the LIF neuron.
reset_voltage	-0.07	V	The neuron reset voltage.
V_spike_threshold	70	V	The spiking threshold for the LIF neuron.
refractory_period	0.0001	s	The refractory period following a spike.

Table 3: List of constants used in the circuit-based Python code, adapted from [11] and [12].

Appendix E

The following is a link to the code and input data used to produce all the relevant figures in the Results section:

https://github.com/MANicholson/Figures-Modelling-and-Simulation-of-a-CMOS-Synapse-Implementing-Two-phase-Plasticity.git