Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2020

Project Title:	**New Processing and Learning Methods for Event-based Video Cameras**
Student:	**Zhenyi Shen**
CID:	**01201740**
Course:	**EEE4**
Project Supervisor:	**Professor Pier Luigi Dragotti**
Second Marker:	**Dr Wei Dai**

Contents

Abstract

Event-based cameras surpass conventional cameras by power efficiency and temporal resolution. However, many existent algorithms, including classification, associated with the event-based cameras, are not yet mature. This project concerns the classification problem with data generated in the event-based cameras. The state-of-art design is the rate-based SNN (Spiking Neural Networks) [1]. This method, however, has a main disadvantage of efficiency: many spikes are generated throughout the network. Regarding to the power efficiency, Google has recently proposed a temporal-coded SNN achieving high power efficiency – at most one spike generation in each neuron [2]. However, the downside is that it is currently only applicable to the dataset having at most one spike per pixel, which can barely be the case in reality. Thus, the main emphasis of this project is to propose a new architecture that can have a good classification accuracy as well as a high power efficiency. Architectures inspired by the existent networks are designed, implemented, tested, and evaluated.

Acknowledgement

I would like to start by sending my deepest appreciation to Professor Pier Luigi Dragotti who gave me the opportunity of this project, and provided me with valuable feedback when required. Furthermore, I would like to especially thank Vincent Leung who guided me throughout the year. Finally, I am very grateful to my family and friends for their continued and unconditional support.

Chapter 1

Introduction

1.1 Motivation

Biomimicry is a method of finding solutions in the nature. It is used everywhere, including the event-based camera, which is inspired by human retina. The human retina is a very efficient tool compared with a conventional camera: it handles a massive amount of data, but consumes much less power (order of 10)[3] [4]. The difference comes from the fact that a conventional camera records a video by capturing full-images at a specified rate (e.g., 30 frames per second), so if a majority of objects in the video are still, the camera will use up a lot of space and power for redundant information. Neurons in eyes, on the other hand, only fire or respond to locations wherever the brightness changes above a predefined threshold. Here come event cameras. Event cameras like Dynamic Vision Sensors (DVS) [5] mimic this selective response, recording a video asynchronously with events, which encode location, time, and polarity.

Thanks to this efficient mechanism, event cameras surpass traditional cameras in areas of high temporal resolution, low latency, low power, and high dynamic range [10]. Consequently, event cameras show themselves advantageous in many applications such as in classification [8] [1], in object tracking [7] or wearables and robotics [6] where real-time interaction is required, while this project only focuses on classification. Despite their high potential performance in classification, event cameras impose a great challenge to engineers because their power can only be fully released if their unconventional raw data are directly processed. As the challenge is due to the mimic of biology, a natural way to resolve that is also to resort to the nature – Spiking Neural Networks (SNNs).

SNNs are a third generation of Artificial Neural Networks (ANN) [11], different from their predecessors by increased similarity to biological neurons. SNNs are constructed by layers of spiking neurons often modelled either by Leaky Integrate-and-Fire (LIF) neuron [13] or Spike Response Model (SRM) [1]. No matter which model is implemented, the model inherently receives events or spikes – raw data from the event camera. Those spikes will change the neuron's internal state – membrane potential. After the membrane potential exceeds a certain predefined threshold – critical potential, the neuron fires a spike to neurons in the next layer. Besides their nature compatibility with event data, SNNs also have enough complexity to obtain a high processing capacity theoretically. SNNs incorporate not only spatial information but temporal information: a spike affects both the current and the future membrane potential. Though we may know Recurrent Neural Networks (RNN) with conventional neurons like sigmoid neurons also take accounts of temporal information, they bear more complexity. Moreover, SNNs are more efficient in computations as their binary nature does not involve any floating-point multiplications, making them advantageous at low-power applications [13].

Training such a network, however, remains a challenge. Like conventional networks, SNNs' training is concerned with noise and non-stationarity in data, and difficulties in long-range temporal and spatial dependencies [13]. SNNs are further troubled with non-differentiability due to spiking neuron's internal dynamics. This problem is partially solved by a surrogate gradient [13], and achieves a state-of-art performance [1]. This method, however, has a drawback of power efficiency due to its loss function – rate-based loss, which encourages spike generation. More recently, Google has proposed a SNN that encodes information by relative timing of the spikes, which is quite efficient because each neuron at most fires once. This architecture, however, also has a shortcoming: it is currently only applicable to manual data with at most one spike at each pixel. A SNN able to acheive both a high classification accuracy, and a good power efficiency is urged. As there are currently no name to distinguish from the architecture using a surrogate gradient and the architecture using a temporal loss, they will be named as STNNs (Spike Train Neural Networks), and TSNNs(Temporal Spike Neural Networks) respectively in this report.

1.2 Project Objectives

This project aims to propose a SNN architecture that classifies the raw data obtained by event cameras. Two main criteria would evaluate the result of the classification – classification accuracy and power efficiency. The proposed SNN will optimise the classification accuracy under a constraint of low power consumption. Inspired by the good performance of STNN, and the high efficiency of TSNN, the proposed SNN would be a hybrid of the two – applicable to real-world data, and power efficient.

1.3 Report Overview

The following report would be constituted of four parts – Background, Analysis and Design, Results and Evaluation, and Conclusion and Future Work.

Background will firstly introduce the building block of SNNs – spiking neuron models, and the architecture of STNNs and TSNNs. Analysis and Design would take STNNs and TSNNs as baseline, then proposes three architectures with more complexity that aims to achieve the project objectives. Results and Evaluation will present the baseline performance first, then compare and analyse the results of two of proposals (one proposal requires more theoretical preparation). At last, Conclusion and Future Work would outline what is achieved by the project and what should be done in future. Though there would be no specific chapter for implementation, all codes will be available at https://github.com/SHENZHENYI/SNN.

Chapter 2

Background

SNNs are heavily inspired by biological neurons, thus this chapter would introduce the structures and the dynamics of the biological neurons first to build some basis for the derived mathematical models of the spiking neurons, namely LIF Model (Leaky Integrate-and-Fire) and SRM (Spike Response Model). After the introduction of the building blocks of SNNs, the training procedure of SNNs will be discussed. As SNNs, typically non temporal coded SNNs, are fed with a sequence of features, resembling RNNs (Recurrently Neural Networks) to a large extent, RNN training would be introduced first, then the additional training challenges brought by SNNs would be identified and background information about the biological neurons, their structures and dynamics are introduced to form a basis for the mathematical model of the spiking neurons. After the two widely used models are introduced, SNN training would be discussed. The training section split into two parts: RNN training and SNN training. RNN training is reviewed first because RNNs resemble SNNs to a large extent, especially by the learning rule, and is less complicated than SNNs. In SNN training, the additional challenge will be identified and one particular successful algorithm – surrogate gradients will be introduced in detail.

2.1 Scope of Neurons

Biologists have investigated the brain over the past hundreds of years [9], and the most fundamental information processing units in the brain have found to be neurons. Despite of different categories of neurons – glia neurons, non-spiking neurons, and spiking neurons, this project restricts the neurons to be spiking neurons.

2.2 Neuron Structure

A typical neuron can be decomposed into 3 parts: dentrites, soma, and axon shown in Figure 2.1. They are the receiver of the incoming signals, the processing unit, and the transmitter of the signal respectively. To speak of the relation between one neuron to other neurons, axons of one neuron reaches out to the dentrites of other neurons. These contacts or junctions between axons and denstrites are called synapses. Synapse is one important terminology because the neuron transmitting the signal is called presynaptic neuron, and the neurons receiving the signal are called postsynaptic neurons, which are heavily used in literatures. Moreover, the membrane potential response of a postsynaptic neuron to a presynaptic action potential is called PSP (Postsynaptic Potential).

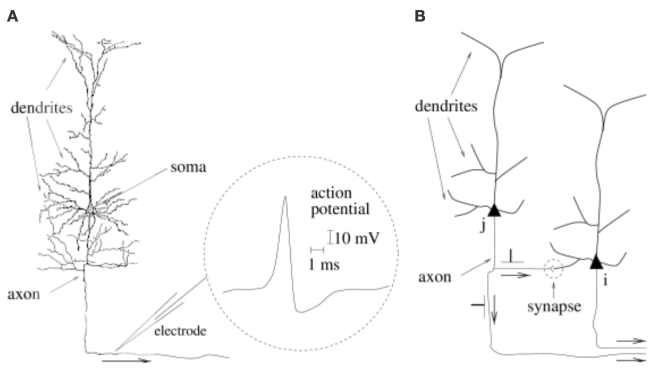


Figure 2.1: Biological neuron structure

2.3 Spikes

Neurons communicate via spikes, or called as pulses or action potentials. One spike is typically of 100mV lasting for 1-2ms [9]. They are of a similar shape among different neurons, thus it can be concluded that information propagation does not embed in spikes' shape but their presence or absence. In other words, a spike train can be characterised only by its firing time when deriving the models.

For the generation of spikes, a neuron must be electrically polarised to a certain extent: its membrane potential is above some critical potential. The membrane potential, $u(t)$, is defined as the potential difference between the interior cell and its surroundings which could be dealt as reference potential [9]. Without any excitation, the membrane potential remains at a resting potential, u_{rest} – normally a negative potential about -65mV [9]. Due to this negative resting potential, an input that reduces the negative polarisation is called 'depolarising', and an input that further increases the negative polarisation is called 'hyperpolarising'.

To fully characterise the dynamics between the spikes and PSP, we need three phases: depolarisation, hyperpolarisation, and undershoot phase (shown in Figure.2.2).

In the depolarisation phase, the resultant membrane potential is approximately the sum of PSPs of all presynaptic neurons. After reaching the critical potential, the linearity breaks: the membrane potential exhibits a large positive pulse-like excursion with an amplitude of 100mV [9] – the spike. After this spike, the neuron starts to be hyperpolarised rapidly, entering the undershoot phase. During the undershoot phase, the membrane potential decreases beneath the resting potential, and subsequently increases back to the resting potential.

2.4 Spiking Neuron Model

This section reviews two models of spiking neuron: the LIF (Leaky Integrate-and-Fire) model and the SRM (Spike Response Model) because both of them are widely used in literature [13] [1]. To compare the two models, SRM is a generalisation of LIF, but in most literature, SRM is formulated in a way that mimics LIF.

2.4.1 Leaky Integral-and-Fire Model

LIF [13] is formulated by three ingredients: the evolution of sub-threshold (below the critical potential) membrane potential w.r.t. synaptic currents, the evolution of synaptic

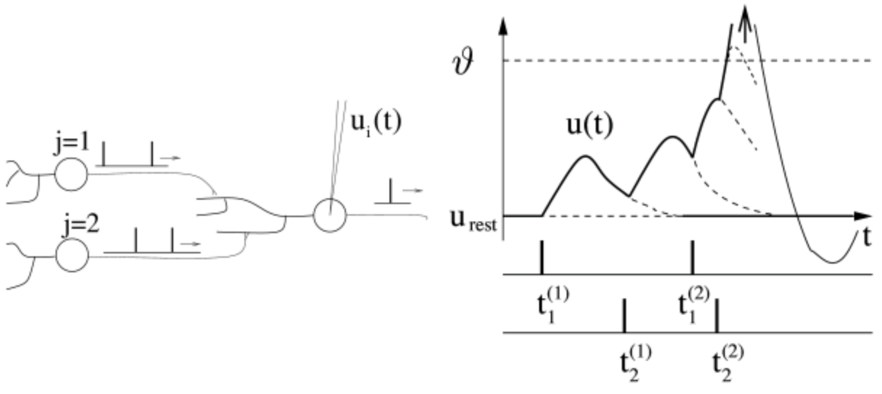


Figure 2.2: The membrane potential of a postsynaptic neuron over time with arrivals of presynaptic spikes [9]. Left: the structure of the sample neural connections. The target postsynaptic neuron is connected with 2 presynaptic neurons labeled as 1 and 2 transmitting spike trains. Right: 3 subplots with the same x-axis are shown – the membrane potential of the postsynaptic neuron, $u(t)$, the spike train of presynaptic neuron 1, and the spike train of presynaptic neuron 2 from top to bottom. Before $t_2^{(2)}$, the neuron is in the depolarisation phase. After that, $u(t)$ reaches the critical potential, ϑ, and results in a spike (the arrow pointing upward). Subsequently, $u(t)$ experiences the hyperpolarisation (a rapid decrease in value), and an undershoot (a decrease beneath the resting potential, u_{rest}, and a gradual increase back to u_{rest}).

currents w.r.t. presynaptic spikes, and the spike generation function according to the membrane potential.

The membrane potential of the neuron i of layer l at time instance t, $U_i^{(l)}$, is defined as the differential equation below.

$$\frac{dU_i^{(l)}}{dt} = -\frac{1}{\tau_m}[U_i^{(l)}(t) - U_{rest} + RI_i^{(l)}(t)] + S_i^{(l)}(t)(U_{rest} - \vartheta) \tag{2.1}$$

where τ_m is the membrane time constant, U_{rest} represents the resting potential, R is the leaky resistance of the membrane, $I_i^{(l)}(t)$ is the momentary synaptic current of neuron i of layer l, S_i^l is the momentary spike train of neuron i of layer l, and ϑ is the critical potential. The paragraph below explains every term in that differential equation from biology.

This differential formulation comes from the fact that the relation between the sub-threshold membrane potential and the synaptic current is like how voltage and current are related in a RC circuit.

Biologically, a neuron is surrounded by a cell of membrane – a rather good insulator. When some ions (can be characterised as synaptic currents, $I(t)$ – the response of the presynaptic spikes) are injected into the neuron, the additional charge would charge membrane – the capacitor, C. Moreover, as the membrane cell is not a perfect capacitor, the leacky current is captured by one additional finite resistance, R. The last term in the equation, $S_i^{(l)}(t)(U_{rest} - \vartheta)$, represents the final phase of the action potential – undershooting. This term, however, instead of having the membrane potential below the resting potential – a biological procedure, brings the membrane potential back to the resting potential immediately after one spike generated. In some formulation, this undershooting phase holds the membrane potential at U_{rest} for a refractory time [20], but we will stick on Equation.2.1 as it only introduces minor differences in practice.

Equation (2.1) only describes the sub-threshold behavior of the membrane potential, the super-threshold or action potential behavior would be described below. The exact form of an action potential carries no information but its firing time as mentioned in Section.2.3. As a result, we have one more variable, $S_i^{(l)}$, to characterise that time.

$$S_i^{(l)} = \Theta(U_i^{(l)} - \vartheta) \tag{2.2}$$

In order to solve the dynamics of the neuron, we need one more equation – the response of the postsynaptic current to a presynaptic spike. One of the most common way to characterise it is a first-order approximation, modeling the current as an exponential decaying current with the presynaptic spikes [13], assumed that the current adds linearly.

$$\frac{dI_i^{(l)}}{dt} = -\frac{dI_i^{(l)}}{\tau_{syn}} + \sum_j W_{ij}^{(l)} S_j^{(l-1)}(t) + \sum_j V_{ij}^{(l)} S_j^{(l)}(t) \tag{2.3}$$

where the summation of j represents all presynaptic neurons, τ_{syn} is the synaptic time constant, W_{ij} is the synaptic weight between the spikes of the presynaptic neuron j of layer l and the synaptic current of the postsynaptic neuron i of layer l, V_{ij} is the recurrent synaptic weight of neurons of the same layer.

For now, there are two differential equations, Equation (2.1) and (2.3), that describe the dynamics of the sub-threshold membrane potential and the synaptic current of the neuron, they, however, are not immediately applicable for implementation, thus come their approximations in discrete time with a simulation step Δt [13]. Without a loss of generality, we adopt that $U_{rest} = 0$, $R = 1$ and $\vartheta = 1$.

Equation (2.1) can be approximated by

$$U_i^{(l)}[n+1] = \beta U_i^{(l)}[n] + I_i^{(l)}[n] - S_i^{(l)}[n] \tag{2.4}$$

where $\beta = \exp(-\frac{\Delta t}{\tau_{mem}})$, and τ_{mem} is positive number such that $0 < \beta < 1$.

Equation (2.3) can be approximated by

$$I_i^{(l)}[n+1] = \alpha I_i^{(l)}[n] + \sum_j W_{ij}^{(l)} S_j^{(l-1)}(t) + \sum_j V_{ij}^{(l)} S_j^{(l)}(t) \tag{2.5}$$

where $\alpha = \exp(-\frac{\Delta t}{\tau_{syn}})$, and τ_{syn} is positive number such that $0 < \alpha < 1$.

These two difference equations, Equation (2.4) and (2.5), together with Equation (2.2), a spiking neuron can be fully described. The corresponding computational graph is shown in Figure 2.3.

2.4.2 Spike Response Model

On the contrary of formulating in differential equations as those in the LIF neuron, SRM is formulated by filter point of view.

The definition of the spikes is the same as that in LIF, generated by a threshold function, with a general form below

$$s_i = \sum_f \delta(t - t_i^f) \tag{2.6}$$

where $t_i^{(f)}$ is the firing time of f^{th} spike in neuron i. Moreover, SRM defines a spike response signal, $a_i(t)$, obtained by convolving the spike train with a spike response kernel, ε, and the corresponding Postsynaptic Potential (PSP) can be defined as $w_i a_i(t)$ – a scaled spike response signal where w_i is the synaptic weight of neuron i.

$$a_i(t) = (\varepsilon * s_i)(t) \tag{2.7}$$

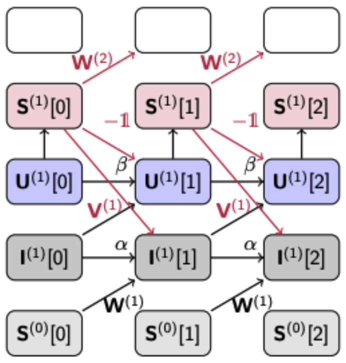


Figure 2.3: Computational graph of a SNN in discrete time [13]. Time evolves from left to right. $S^{(0)}[n]$ are the input at time step n, which are fed to the network from bottom to top. The synaptic current, I, and the membrane potential, U, decays by α, and $beta$ every time step. Spikes are generated when the membrane potential crosses some threshold. The weighted spikes induce the synaptic current, both of them generates the membrane potential????

SRM also models the undershoot phase of the action potential by a refractory kernel, v. The refractory signal, $(v * s)(t)$, restrains the neuron from generating a second spike immediately after the first one.

With PSP and the refractory signal defined, the membrane potential of neuron j, $u_j(t)$, is the sum of them.

$$u_j(t) = \sum_i w_i(\varepsilon * s_i)(t) + (v * s)(t) = w^T a(t) + (v * s)(t) \tag{2.8}$$

If $\varepsilon(t)$ is properly selected such as in a form of $t/\tau_{syn}\exp(1-t/\tau_{syn})$ [1], the membrane potential would be similar to that in a LIF neuron without recurrent connections.

In summary, with Equation (2.2) (2.7) (2.8), a feedforward spiking neural network can be fully describe. Moreover, this formulation is capable of not only implementing fully-connected layers but convolutional and pooling layers [1].

2.5 Training

There is one heavily used SNN architecture making the input to be a sequence of binary features with 1 indicating the presence of the spike, and 0 indicating the absence of the spike [13] [1], which is called STNN (Spike Train Neural Network) in this project. Another kind of SNN is recently proposed by Google, a temporally coded SNN, which receives the input of analog time information [2], and is called TSNN (Temporal Spike Neural Network) in this project. Besides their difference in the data structure for the input, the former usually takes a rate-based loss, while the later employs a temporal loss.

The STNNs resemble RNNs to a large extent because they share some similarities in a sense of architecture, learning through weight adjustment, and especially of temporal dynamics [13]. It is also worth pointing out that STNN can even be viewed as a subclass of RNNs because one spiking neuron's current state depends on the previous one. Thus STNN training section will first introduce RNN training, then additional training challenges brought by the spikes. TSNN training would be subsequently introduced.

2.5.1 STNN

Training of STNN involves the rate-based loss function and a backpropagation algorithm that calculates all the gradients for the weights.

Rate-based Loss

Spike Rate is most commonly used to encode information in SNN. The index of the output neuron that fires the most spikes is the predicted label. The loss is computed as the L2 loss of the difference between the predicted number of spikes, n, and the expected number of spikes of the output neurons, m, as shown in 2.9. For the expected number of spikes, the true class neuron is assigned with 60 spikes, and the wrong class neurons are assigned with 10 spikes each. All of them are found experimentally in [1]. This error assignment method is the state-of-art so far, achieving 99.2% for test accuracy [1]. This method, however, is not so power efficient because this network would generally have many spike generated.

$$loss = \sum_j (n_j - m_j)^2 \tag{2.9}$$

Backpropagation in RNN

The loss function reflects how the current network deviates from the expected performance. And a backpropagation algorithm is required to update the weights of the network in a way that minimizes the loss of the current sample. The most common update method is Gradient Descent (GD). GD modifies the weights in the opposite direction of the gradients. The most general procedure of GD is a spatial credit assignment used in FNN (Feedforward Neural Networks), which firstly obtain the loss by forward-propagation and then use back-propagation to obtain the gradient. Compared with FNN with only spatial credit assignment, GD in RNN is a spatiotemporal credit assignment.

Spatial Credit Assignment

Spatial Credit Assignment propagates error signal only in the sense of architecture of the network – from the output layer towards the input layer. The update rule of a weight, W_{ij}, is defined below [13].

$$W_{ij} \leftarrow W_{ij} - \eta \Delta W_{ij} \tag{2.10}$$

where $\Delta W_{ij} = \frac{\partial \mathcal{L}}{\partial W_{ij}} = \frac{\partial \mathcal{L}}{\partial y_i} \frac{\partial y_i}{\partial a_i} \frac{\partial a_i}{\partial W_{ij}}$, $a_i = \sum_j W_{ij} x_j$ is the input from neurons j to a neuron i, and y_i is the output of a_i after the activation function of the neuron i.

By the chain rule of the derivatives, we can obtain that

$$\Delta W_{ij} = \delta_i^{(l)} y_j^{(l-1)}, \text{where } \delta_i^{(l)} = \sigma'(a_i^{(l)}) \sum_k \sigma_k^{(l+1)} W_{ik}^{T,(l)} \tag{2.11}$$

where σ' is the derivative of the activation function, and T is the transpose.

Spatiotemporal Credit Assignment

There are 2 methodologies for spatiotemporal credit assignment: a backward method and a forward method.

1. The backward method, also known as Backpropagation Through Time (BPTT), unrolls the network in time and do the conventional backpropagation. The only difference between this spatiotemporal credit assignment and the spatial credit assignment is that this method sums up the gradients for W in each time step. The update rule is defined as below [13] and the corresponding computational graph is shown in Figure 2.4.

$$\Delta W_{ij}^{(l)} \propto \frac{\partial \mathcal{L}[n]}{\partial W_{ij}^{(l)}} = \sum_{m=0}^{t} \delta_i^{(l)}[m] y_j^{(l-1)}[m], \text{and } \Delta V_{ij}^{(l)} \propto \frac{\partial \mathcal{L}[n]}{\partial V_{ij}^{(l)}} = \sum_{m=1}^{t} \delta_i^{(l)}[s] y_j^{(l)}[m-1] \tag{2.12}$$

$$\delta_i^{(l)}[n] = \sigma'(a_i^{(l)}[n])(\sum_k \delta_k^{(l+1)}[n] W_{ik}^{T,l} + \sum_k \delta_k^{(l)}[n+1] V_{ik}^{T,l}) \tag{2.13}$$

where the notation follows GD.

2. The forward method keeps the network's recurrent structure, able to run continually in sample time [15]. Below W_{ij}^m is formulated [13], while V_{ij}^m can be derived similarly.

$$\Delta W_{ij}^m \propto \frac{\partial \mathcal{L}[n]}{\partial W_{ij}^m} = \sum_k \frac{\partial \mathcal{L}[n]}{\partial y_k^{(L)}[n]} P_{ijk}^{L,m}[n], \text{with } P_{ijk}^{(l,m)}[n] = \frac{\partial y_k^{(l)}[n]}{\partial W_{ij}^m} \tag{2.14}$$

$$P_{ijk}^{(l,m)}[n] = \sigma'(a_k^{(l)}[n])(\sum_{j'} V_{ij'}^{(l)} P_{ijj'}^{(l,m)}[n-1] + \sum_{j'} W_{ij'}^{(l)} P_{ijj'}^{(l-1,m)}[n-1] + \delta_{lm} y_i^{(l-1)}[n-1]) \tag{2.15}$$

where the notation follows GD.

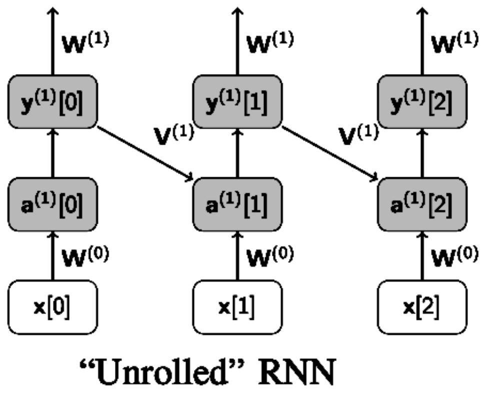


Figure 2.4: Computational graph of a RNN in discrete time [13].

Generally, the backward update method is more efficient than the forward method in terms of computation, but it has to store all inputs and activation outputs for each time step [13]. BPTT's space complexity is $O(NT)$ [13], where N is the number of neurons per layer and T is the number of the time step. For the forward method, it has to store every $P_{ijk}^{l,m}$, which has a space of complexity of $O(N^3)$. This complexity can be huge for a large N, but with some simplification in computational graph [12], it can be reduced to $O(N)$ – much smaller than that of BPTT. This comparison in space complexity, however, is only concerned in implementation: neuromorphic hardware may have constraints on memory, both algorithms generate the same results.

Training of RNN ends here. In the next section, we will look for an exclusive training challenge imposed on SNNs.

Challenges in SNN Training

Backpropagation of a neural network always involves the derivative of the activation function, $\sigma' = \frac{\partial y_i^{(l)}}{\partial a_i^{(l)}}$. This term in SNN, however, is generally the derivative of a Heaviside function whose derivative is zero everywhere except a spike when the membrane potential equals the critical potential. This property prevents the gradient of the weights from flowing: nothing is learnt when the spikes are absent.

To tackle the non-differentialbility, There are most common 4 approaches are devised [13].

1. using biologically inspired local learning rules for the hidden units
2. transferring conventionally trained 'rate-based' NN to SNNs.
3. smoothing the neural model to make it differentiable
4. defining a surrogate gradient that are continuous

While this project focuses on the 4th method – SG (Surrogate Gradients) because this method can train SNNs to benchmarks its performance against the other 3 approaches [13]. Moreover, several papers of this method used promising 'three-factor' plasticity rules [17] which are believed to be the underlying rule of synaptic plasticity in the brain [13].

SG retains the Heaviside function as the threshold function but defines a 'fake' continuous gradient to approximate the derivative of the Heaviside function in a backward propagation. Thus this smooth prevents the gradient from vanishing, if the network is not deep.

Figure 2.5 shows several most commonly used surrogate gradients. Many studies have conducted to construct their own SGs, SGs, however, are always non-linear functions that monotonically increase to the threshold potential. Until now, there is no systematic comparison between different SGs, but the existence of the plethora of SGs may suggest that the form of SGs does not matter much [13]. If there is hardware constraint that makes BPTT impossible, two families of algorithms have been developed about SG to bypass that by making the update rule more local such as Feedback Alignment and Random Error Backpropagation and Supervised Learning with Local Three-factor Learning Rules. The first method successfully achieved little loss in performance but lost the temporal dynamics of the neurons [13]. The second method fixed that problem and even was able to update recurrent synaptic weights locally [13].

There are 2 major challenges remaining. The first one is the bias brought by the approximation of the derivative of the Heaviside function [13]. The second one is that SGs are generally derived from a sigmoid function which will face a gradient vanishing or exploding in a deep network. The later one can be solved by local errors performed with BPTT rules [21].

2.5.2 TSNN

Training TSNN involves the temporal loss function and a backpropagation algorithm which is exact w.r.t. the forward propagation function. The forward propagation is

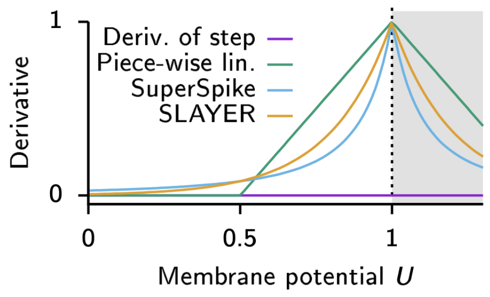


Figure 2.5: Most used Surrogate Gradients [13]. The violet line is the derivative of the Heaviside function which is 0 everywhere except at 1 where it is not defined. The green piece-wise line is just like ReLU [18]. The blue curve is the derivative of a fast sigmoid [17]. The orange curve is an exponential function [1].

one derivative SRM, but is nevertheless listed to facilitate the understanding of the corresponding backpropagation equations.

Temporal Loss

TSNN encodes information in the relative timing of neurons within the same layer. The earliest spike is recognised to carry more critical information, thus the index of the output neuron with the earliest spikes arrived among all output neurons is the prediction of the network. The loss, L, is computed by the cross entropy of the posterior probability, p, of the output neurons, which is computed by a softmax function on the negative value of the output spiking time. The negative value of the spike time is taken to compute the probability because the earlier the spike time, the higher the probability it is the label to be predicted. The two equations are listed below, where σ_i is the spiking time of one output neuron i, and y_i is the one-hot coded label i.

$$p_j = e^{-\sigma_i} / \sum_{i=1}^{n} e^{-\sigma_i}$$

$$L(y_i, p_i) = -\sum_{i=1}^{n} y_i ln(p_i)$$

Forward Propagation

This method has the same forward propagation in SRM models with the spike response kernel defined as $\varepsilon = te^t$ and without a refractory kernel. Refractory kernel is omitted because it is to prevent the immediate generation of a second spike, while TSNN generates only one spike.

The membrane potential is defined as the follows with a set I containing all presynaptic inputs at times $t_i \leq t$.

$$V_{mem}(t) = \sum_{i \in I} w_i(t - t_i)e^{\tau(t_i - t)} \tag{2.16}$$

The neuron spikes when the membrane potential exceeds some predetermined threshold, θ, and the time of that spike is denoted as t_{out}. t_{out} is the earliest spike time of the neuron.

$$V_{mem}(t_{out}) = \sum_{i \in I} w_i (t_{out} - t_i) e^{\tau(t_i - t_{out})} = \theta \tag{2.17}$$

t_{out} could also be expressed as a function of weights and presynaptic spike time. We denote $A_I = \sum_{i \in I} w_i e^{\tau t_i}$ and $B_I = \sum_{i \in I} w_i e^{\tau t_i} t_i$, and W is the Lambert W function. This expression is distinct for the temporal coding, and is the key to do the back propagation.

$$t_{out} = \frac{B_I}{A_I} - \frac{1}{\tau} W(-\tau \frac{\theta}{A_I} e^{\tau \frac{B_I}{A_I}}) \tag{2.18}$$

Backpropagation

The backpropagation involves the definition of the loss function and the gradients of the earliest spike time, t_{out}, with respect to any presynaptic spike time, t_j, and the corresponding weight, w_j.

When the spike time of the output neurons are computed, their negative values are fed into a softmax function, yielding a probability of each neuron: $p_j = e^{-\sigma_i} / \sum_{i=1}^{n} e^{-\sigma_i}$. The negative value of the spike time is taken to compute the posterior probability because the earlier the spike time, the higher the probability it is the label to be predicted. The loss function employs the cross-entropy loss: $L(y_i, p_i) = -\sum_{i=1}^{n} y_i ln(p_i)$, where y_i is the element of the one-hot coded label.

$$\frac{\partial t_{out}}{\partial t_j} = \frac{w_j e^{t_j} (t_j - \frac{B_I}{A_I} + W_I + 1)}{A_I (1 + W_I)} \tag{2.19}$$

$$\frac{\partial t_{out}}{\partial w_j} = \frac{e^{t_j} (t_j) - \frac{B_I}{A_I} + W_I}{A_I (1 + W_I)} \tag{2.20}$$

where $W_I = W(-\frac{\theta}{A_I} e^{\frac{B_I}{A_I}})$.

Synchronisation Pulses

Besides the forward and backward propagation, in order to train the network successfully, some more elements are added. synchronisation pulses works as additional inputs across all layers to provide some bias: spikes brought by neurons along may be not enough to invoke any spike in the next layer. Their spike time and their corresponding weights are trainable, with the gradients defined in Eq. 2.19 and Eq. 2.20. Each of the pulse is initialised uniformly in the interval (0,1), the same interval of normalised inputs.

2.6 Additional Comments on TSNN

TSNN achieves high performance on manual data, XOR and non-convolutional N-MNIST. Both of them, however, only have one spike per pixel, the real-world data can have many spikes per pixel within a time interval. Thus, the current algorithm is hard to generalise to real-word data because the neurons can only operate on one spike time.

The advantage of TSNN is that it is using the exact gradient, while STNN generally uses fake gradients. This fake gradients may work for rate-based loss, but less applicable to a temporal loss, which has a higher demand in precision.

Chapter 3

Analysis and Design

3.1 Dataset

This project does classification on two datasets: DVS128 Gesture Dataset, and N-MNIST (Neuromorphic-MNIST). The ultimate goal is to classify the gesture data from DVS128 Gesture, though the architecture decisions will be made with N-MNIST because the two datasets share certain similarities, and N-MNIST enables a quicker feedback. Moreover, the same number of training and test data of N-MNIST are used, 1000 and 100 respectively, in order to mimic the situation in DVS128 Gesture. Although the two datasets have a different extension, they will be transformed into the same data structure – an event class, which would be introduced below.

3.1.1 DVS128 Gesture Dataset

The data was recorded with a DVS128 event-based camera. It contains 11 gestures from 29 different people under 3 different illumination conditions. The l1 gestures are arm roll, hand clamp, left hand clockwise, left hand counter clockwise, left hand wave, right hand wave, right hand clockwise, right hand counter clockwise, air drums, air guitar, and other gestures as shown in Figure 3.2.

Each gesture frame is constructed by 128x128 pixels. Each pixel has 3 values -1, 0, and 1, representing a positive change, no change, and a negative change in the brightness respectively. Moreover, each video has a certain length, may or may not be the same, but typically around 6 seconds.

3.1.2 N-MNIST Dataset

The N-MNIST is a spiking version of the original MNIST hand-written digit data. It consists 60,000 training data, and 10,000 test data. The samples are recorded by a moving event-based camera (ATIS sensor) on the MNIST data on a LCD monitor. Each frame is constructed by 28x28 pixels. Each video has a different length, but typically around 0.3 second. One snapshot is shown in Figure 3.2.

3.2 Preprocessing

Preprocessing is to process the original data before feeding them into the network. There are generally two goals to do preprocessing. One goal to make the original data compatible to a specific network. Another goal is to make the information easier to be extracted via the network. Before introducing preprocessing, the data structure of the

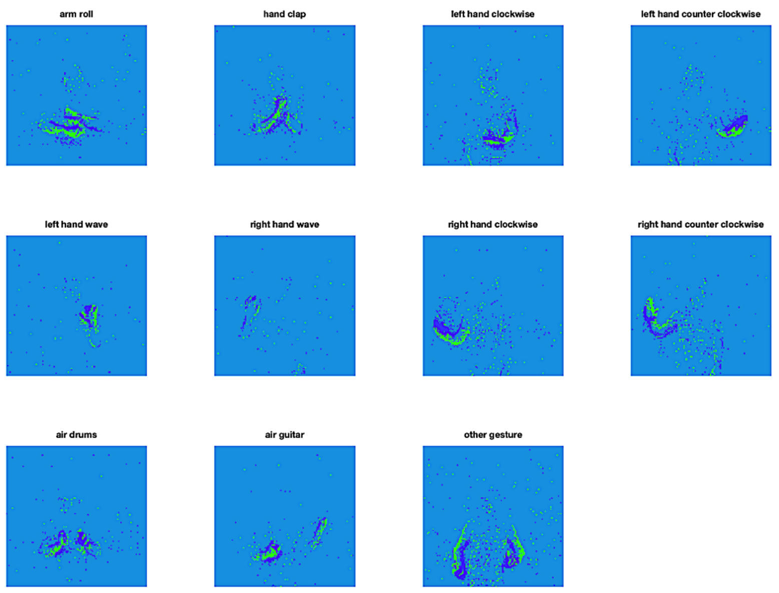


Figure 3.1: 11 typical labels in DVS128 Gesture clipped at 0.1s [23]

raw data is firstly introduced. Moreover, because there are two baseline models in this project, the preprocessing procedure of both of them would be discussed.

3.2.1 Data Structure of Raw Data

The data type of the raw data is a class, denoted as 'event', with 4 attributes: x-axis, y-axis, polarity, and time. For example, if the pixel at a position (2,5) has a negative change in the brightness (polarity is -1) at a time instance of 33.456ms, then we would have event.x = 2, event.y =3, event.p =-1, event.t =33.456. This is a sparse representation because this class only contains information whenever there is a spike. This is a valid representation because spiking or not is a binary event: if we know when and where there is a spike, whatever left do not contain a spike. In this way, a massive amount of memory can be saved because most pixels contain no spikes.

3.2.2 STNN

Preprocessing of STNN converts the original data, a sparse data representation, into a dense representation – containing information of both spiking time and locations, and non-spiking time and locations. The dense representation is required because it enables one pixel to be represented by one input neuron, which is necessary for the training to happen. Thus, the input layer will have a size of $34*34*2 = 2312$. Moreover, the features are the flattened like any other fully-connected neural networks. At this stage, we will not extend our work to CNNs (Convolutional Neural Networks), which can receive a data with its original shape (34x34x2) because CNNs would be too complex to begin with.

Besides that STNN requires an input layer of size 2312, the input needs to be a sequence of frames of spikes – a spike train. It has to be a sequence because otherwise the sparse and dense representation can never be equivalent. Also note that the length of the sequence depends on the sampling period and the window of the sampling process. Following SLAYER's convention [1], we will truncate the video at 300ms, and sample the data at a sampling period of 1ms. As a result, one sample would have a size

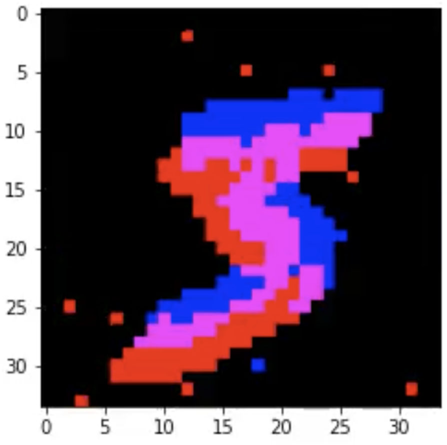


Figure 3.2: Digit 5 in 'N-MNIST' clipped at 0.1s

of (300x2312), and STNN would be fed with a 2312 binary vector for 300 iterations. The vector is binary because spiking is a binary event and the spike's shape is of no significance

3.2.3 TSNN

TSNNs can not receive a sequence, thus it is fed with a feature vector of size 2312. This single feature vector, however, would never be enough to fully represent the original data. Given this limitation, it is necessary to extract the most discriminative information from the original data. If we still recall the logistic of the original paper, we will construct a filter, extracting the earliest spikes of each pixel in the original data [2]. Given the earliest spikes required to be extracted, we still have the freedom to define the scope of 'early'. For example, we can either take earliest spikes from different windows or even loosen the constraint of t́he earliestt́o the second earliest or to the third earliest spikes. These different interpretations of earliest spikes will yield more data, and we can choose how to process these data – one of the most significant thoughts for the derivatives of TSNN. Besides processing the data structure, preprocessing can also filter the input to make the classification easier as shown in the following sections.

Normalisation

TSNNs are fed with the earliest spikes. All of these temporal information will be restricted in an interval between 0 and 1. There are two reasons for this normalisation. The first one is that it brings the timing of all data to the same scale. The second argument is that when TSNN was firstly proposed, the data were restricted in that interval [2]. Thus, if this project uses the same scale, we could use their parameters as a starting point because N-MNIST and the dataset they applied on share certain similarities – both of them originating from MNIST.

Compression

Each frame in the N-MNIST dataset has a size of 34x34x2, while each frame in DVS128 Gesture dataset has a size of 128x128x2 – almost 14 times larger than N-MNIST data. As a result, fitting a network with 100 hidden units with the gesture data of a batch size of 10 makes the CUDA out of memory. Thus a compression process is required to overcome this memory problem. The filter will compress four neighbouring pixels into one. Speak of the function of the filter, it can either take the average spiking time or the minimum spiking time. Average filter aims to incorporate information equally from each pixel, while a minimum filter regards the earliest spike among the four as the most significant.

3.3 Baseline Architecture

3.3.1 STNN

STNN is employed in both SLAYER and Surrogate Gradients [1] [13]. The input data is a sequence of binary vector as described in Section 3.2.2. STNN operates one frame a time like a RNN because both of them receive sequential data, though STNN does have explicit recurrent connections but an implicit one. The implicit recurrency comes from the fact that the current states of the neuron depends on the previous states which can be seen from Equation (2.4) and (2.5), where the states are synaptic current and membrane potential of the neurons. Speak of the propagation of information, spikes are the media to communicate between neurons in different layers. They result in internal dynamics of the neurons, which will or will not transmit a spike to neurons in the next layer depending on whether the membrane potential has crossed the threshold.

Every operation in the dynamics is well-defined except a spike generation function – a Heaviside function. The reason is that the derivative of a Heaviside function is a Dirac function, which will prevent the gradient from flowing in backpropagation. To overcome this problem,a surrogate gradient, *grad*, is defined for the spike generation function.

$$grad = \frac{1}{(100 * |x - \vartheta| + 1)}$$

where x is the membrane potential, and ϑ is the threshold.

This architecture achieves the state-of-art performance among SNNs (99.20% test accuracy for N-MNIST), but is intrinsically inefficient: STNN is usually associated with the rate-based loss, which encourages spike generation throughout the network. STNN will not be fine-tuned as we have already know its power from other's work, and our main focus will be on TSNN.

3.3.2 TSNN

TSNN is different from STNN fundamentally in the data representation as described in Section 3.2.3 [2]. TSNN's input is not a sequence but a single vector with each element storing an analog value – the spiking time of that particular pixel. Thus, layers will be communicated by an analog-valued timing of the spikes, which is very similar to a conventional neural network with its activation function defined in a SNN way. The forward propagation of TSNN is the same as that of STNN, while for the backpropagation, unlike STNN using a surrogate gradient, TSNN is able to derive an exact form of gradient, which is very promising. For power efficiency, this network cannot be more efficient because the number of spikes generated in one classification is at most the sum of neurons involved: every neuron at most spikes once. The limitation of TSNN is that one pixel can only be represented by one spike, resulting a loss from the original data. Moreover, because of this reason, currently no one is applying TSNN onto a real-world data such as 'N-MNIST' and 'DVS128 Gesture', but we will translate this network onto those datasets with the preprocessing methods mentioned in Section 3.2.3. At last, TSNN will be fine-tuned as no one has applied it onto the datasets we are interested in, and its performance would be our baseline performance.

3.4 Design of the Network

STNN and TSNN would become two baseline architectures for this project, while each of them has certain advantages – good accuracy and high power efficiency respectively. Thus, our aim is to construct a spiking network that combines the strength of STNN and TSNN. In order to do so, I proposed the following architectures.

3.4.1 Concatenation of Features

This method is TSNN based. As the problem of the baseline TSNN is the truncation of the original data, in this version network, we will feed it with more spikes as additional features. There would be two ways to define the additional spikes. One way is to split the spikes according to the order of earliest spikes such as the first earliest spikes, the second earliest spikes, and so on. Another way is to partition the original data into multiple windows, and take the earliest spikes in each individual window.

If we split data according to orders, and only uses the first two earliest spikes, the weights between the first earliest spikes and the hidden units will be as w1, and the weights between the second earliest spikes and the hidden units will be denoted as w2. Because the first earliest spikes and the second earliest spikes are originated from the same pixels, w1 and w2 must have, more or less, some correlation. This correlation can be configured as a regularisation term in TSNN. One way to introduce this regularisation is to use a shared parameter (w1 = w2) – similar to the idea of parameter sharing in CNN (Convolutional Neural Networks).

Another way is to introduce a regularisation loss, which can either be defined as a L1 regularisation loss or a L2 regularisation loss as listed below. With this additional loss, w1 and w2 would be optimised to have a similar modulus.

$$L1\_regLoss = \sum_{ij} |w1_{ij} - w2_{ij}|$$

$$L2\_regLoss = \sum_{ij} (w1_{ij} - w2_{ij})^2$$

This method, however, has a drawback of overfitting because with the number of training data unchanged, more features will be trained. Nevertheless, this overfitting problem can be theoretically alleviated by the regularisation methods introduced in the previous paragraph.

3.4.2 Concatenation of the TSNN and STNN

As discussed in Section (3.3.1) and (3.3.2), it is known that STNN has a strength of its receiving part, but a drawback of inefficiency due to the loss function. TSNN, on the other hand, could only receive partial original data, but be able to have a temporal loss which is power efficient. Thus, an idea has been come up to concatenate STNN and TSNN, allowing STNN to receive the data, and TSNN to produce a temporal loss.

This concatenation of two networks, however, has a problem in the junction of the two networks because the data are not compatible. Let's refresh ourselves of the data type of the two networks. STNN's input is a sequence of binary features, while TSNN's input is one temporal feature vector. Assumed that the input has 3 features and a maximum time of 5ms, STNN's input (S1) and the corresponding normalised TSNN's input (S2) would be like the expressions below.

$$S1 = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

$$S2 = \begin{bmatrix} 0.25 \\ 1 \\ 0 \end{bmatrix}$$

To convert S1 to S2, we can think of a mask matrix (M), having the same size of S1, and every row of M is the flipped normalised time.

$$M = \begin{bmatrix} 1 & 0.75 & 0.5 & 0.25 & 0 \\ 1 & 0.75 & 0.5 & 0.25 & 0 \\ 1 & 0.75 & 0.5 & 0.25 & 0 \end{bmatrix}$$

Thus, S2 can be obtained by

$$S2 = 1 - max(M * S1) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - max \begin{bmatrix} 0 & 0.75 & 0 & 0.25 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0.5 & 0.25 & 0 \end{bmatrix} = \begin{bmatrix} 0.25 \\ 1 \\ 0 \end{bmatrix}$$

where the max function is only applied on the rows of the matrix.
Note that the reason that S2 is obtained by a max function instead of a min function combined with a reversely ordered mask function is that the minimum value can be 0, which may indicate that there is no spike in S1.

In this way, S2 contains the temporal information of earliest spikes in S1. The data type conversion from STNN to TSNN, however, is completed only for the forward propagation. For backpropagation, the gradient calculated by 'autograd' for this specific combination of $S1$ and $S2$ is

$$\frac{dS2}{dS1} = \begin{bmatrix} 0 & -0.75 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

This gradient is hardly right, which can be reasoned as follows. Firstly, although it is unknown that if a gradient of a spike at 0.25 should have a higher magnitude that that of a spike at 0.0, the second row of $\frac{S2}{S1}$ contains only zero elements, making the gradient of the time with spike the same as the gradient of the time without a spike. Secondly, the minus sign should not appear here because it originates from a mathematical manipulation that facilitates the extraction of the first earliest spike, which does not have any physical meaning for the conversion.

Thus, we need to define a surrogate gradient for this conversion function. Provided that each element of S2 is the earliest spike of each row in S1, the gradient of S2 then should only contribute to the entry that contains the earliest spikes in S1. This suggests that the gradient of conversion will only be non-zero when the entry of S1 is the earliest spike. The magnitude of the non-zero gradient is still undecided for now. As a baseline, those non-zero values will be forced to be the same, which is arbitrarily taken as 1. Taking this convention to the previous example, the surrogate gradient will be

$$\frac{dS2}{dS1} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

3.4.3 Sequential TSNN

Sequential TSNN aims to receive more information from the original data in a sequential way without concatenating a STNN in the front. In this way, we can avoid defining the conversion function in the previous section. Each time, Sequential TSNN receives a feature, and the feature together with the hidden state of each neuron, produces spikes, if there is any, and updates the hidden states. The hidden state is defined as the previous membrane potential.

The forward propagation functions (Equation (2.16) and (3.3)) of TSNN can be modified as

$$V_{mem}(n+1,t) = \sum_{i \in I} w_i(t - t_i(n+1))e^{t_i(n+1)-t} + V'_{mem}(n) \tag{3.1}$$

where n indicates the frame number in the sequence of inputs, and $V'_{mem}(n)$ is the decayed membrane potential of n^{th} frame.

$$\sum_{i \in I} w_i(t_{out} - t_i(n+1))e^{\tau(t_i(n+1)-t_{out}(n+1))} = \theta - V'_{mem}(n) \tag{3.2}$$

From Equation 3.2, we can express $t_{out}(n+1)$ by using a dummy variable.

$$t_{out}(n+1) = \frac{B_I}{A_I} - \frac{1}{\tau}W(-\tau\frac{\theta - V'_{mem}(n)}{A_I}e^{\tau\frac{B_I}{A_I}}) \tag{3.3}$$

Moreover, it is worth noting again that $V'_{mem}(n)$ is a decayed version of $V_{mem}(n)$. More precisely, it is decayed for an extra time equal the difference between the first spike to the second spike of the neuron. To illustrate this point, let's review one example an input sequence with two spikes – one spiking at 0.3s and another at 1s. The two spikes induces two spike responses of the neuron – PSP1 and PSP2 respectively as shown in Figure 3.3. At 0.4s, the neuron spikes, yielding a postsynaptic spiking time of 0.4 and a membrane potential, $V_{mem}(n, 0.4)$. The spiking time is propagated to the next layer, while the membrane potential is saved as a hidden state. $V_{mem}(n, 0.4)$, however, cannot be directly used because when the neuron spikes for a second time at 1.1s, it should be $V_{mem}(n, 1.1)$ involved in calculation for the second spike rather than $V_{mem}(n, 0.4)$.

Decaying the membrane potential until the generation of a next spike is hard because the membrane potential of one neuron is the consequence of all presynaptic spikes (2312 for the membranes in the second layer). Each of their PSPs must be tracked individually as those PSPs starts at different time. Thus, it is almost impossible to construct an exact decaying method but a proxy due to the complexity. The proxy is constructed to be the membrane potential decaying to the starting time of the next sampling time or the end time of the current window as shown in Equation (3.4), which ignores the decay from the starting time of the next sampling time to the generation of the second spike, and the membrane potential in the previous state, $V'_{mem}(n-1)$.

$$V'_{mem}(n) = \sum_{i \in I} w_i(t_s - t_i(n))e^{t_i(n)-t_s} \tag{3.4}$$

where t_s is the end time of the current window. For example, if the window is from 20ms to 30ms. t_s will be 30ms.
Thus, this decaying method overestimates the membrane potential, $V'_{mem}(n, 1.1)$, which will introduce an error to Sequential TSNN.

Here ends the forward propagation, and starts the definition of backpropagation. Since there is one more input and one more output in forward propagation, there would also be additional gradients to calculate in backpropagation. All of them can be derived from forward propagation functions above.

From Equation (3.4), we can obtain

$$\frac{\partial V_{out}}{\partial w_i} = (t_s - t_i)e^{t_i - t_s} \tag{3.5}$$

$$\frac{\partial V_{out}}{\partial t_i} = -(1 - t_i)e^{t_i - t_s} \tag{3.6}$$

$$\frac{\partial V_{out}}{\partial V_{in}} = 0 \tag{3.7}$$

where V_{out} is the output membrane potential, and V_{in} is the input membrane potential.

From Equation (3.3), we can obtain

$$\frac{\partial t_{out}}{\partial V_{in}} = -\frac{1}{A_I} e^{\tau \frac{B_I}{A_I}} W'(-\tau \frac{\theta - V_{in}}{A_I} e^{\tau \frac{B_I}{A_I}}) \tag{3.8}$$

where W' is the derivative of Lambert W function. Luckily, it is a function of Lambert W function, which makes the computation easy.

$$W'(x) = \frac{1}{x + \frac{x}{W(x)}}$$

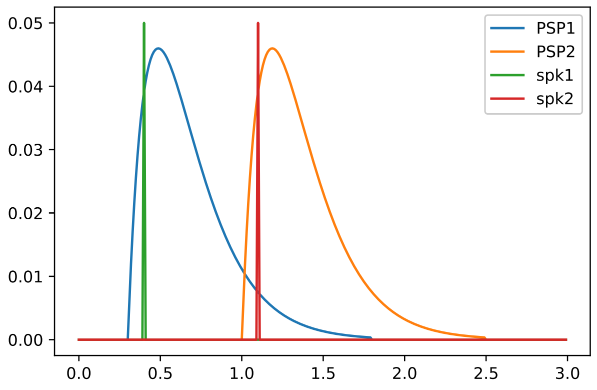


Figure 3.3: Spike responses and Postsynaptic spikes of a neuron fed with two presynaptic spikes at 0.3s and 1s respectively. PSP1 is the spike response of sample spike 1. PSP2 is the spike response of the next sample. spk1 is a Dirac function showing when the neuron firstly spikes, and spk2 shows when the neuron spikes for sample 2.

For the last comment on Sequential TSNN, the windowing would also be a hard part because a too short window would make both the training process and the prediction process too long, while a too long window would make the previous membrane potential decay to 0 – losing information in hidden states, and omit too many spikes (only earliest spikes are used in a window). In summary, this design is still immature, having problems in windowing and the membrane potential decay, thus it is of the lowest priority for implementation.

3.5 Loss Functions

Loss function calculates how the predicted output is deviated from the actual label. The definition of the loss function controls in which way weights are updated, and it is the loss function makes the efficiency difference between STNN and TSNN.

3.5.1 Rate-based Loss for STNN

The rate-based loss calculates how the predicted number of spikes in each output neuron is deviated from the expected number. It can be designed as the sum of squared difference between the actual spikes, and the expected spikes. The expected number of spikes in labels are found experimentally in SLAYER [1]. The expected true label has 60 spikes, and each of the expected wrong labels has 10 spikes. For the predicted number of spikes, as the spike train is a binary vector, 1 indicating the presence of a spike, and 0 indicating the absence of a spike. The predicted number of spikes will then be the sum of values in the spike train. It is worth pointing out that in this definition, the gradient from the loss side would propagate to every element in the spike train equally

3.5.2 Cross-entropy Loss for TSNN

For TSNN, the output will be the timing of spikes arriving the output neurons. The posterior probability, the probability of a label given the sample, is estimated by the negative value of the output fed into a softmax function with the equation listed below.

$$p_j = e^{-\sigma_i} / \sum_{i=1}^{n} e^{-\sigma_i}$$

The negative value of the spiking time is to make the probability of the earlier spike more significant than the latter ones. This decision, however, is arbitrary. Taking the reciprocal value of the spiking time also does the work.

Cross-entropy loss for one prediction, $L(y_i, p_i)$, is the log of the reciprocal posterior probability of the groundtruth label:

$$L(y_i, p_i) = -\sum_{i=1}^{n} y_i ln(p_i)$$

where y_i is the one-hot label.
If the actual label has a very low predicted posterior probability like 0.1, the loss will be 2.3, while if the actual label has a predicted posterior probability of 1.0, the loss will be 0.

3.5.3 Temporal loss for STNN

STNN can be assigned a temporal loss by a conversion function defined in Section 3.4.2 after the output layer and a cross-entropy loss.

The conversion function convert the output spike train into a single vector with each element containing the timing of the earliest spikes at that particular pixel. Then the negative value of that timing is fed into a softmax to estimate the posterior probability of each label of the sample. At last, the cross-entropy loss is calculated from those posterior probabilities.

3.6 Hyperparameters

3.6.1 Optimiser

The optimiser starts with the Adam optimisation algorithm because Adam is found to be both robust and suitable to a wide range of non-convex optimisation [24]. As it combines the advantage of both AdaGrad and RMSProp, which are good at dealing with sparse gradients, and non-stationary objectives respectively. However, Adam is also found to have a worse generalisation property than SDG (Stochastic Gradient Decent) with momentum [25]. Nevertheless, Adam converges quickly and generally achieves a decent result. Thus, we will experiment some candidates of other optimisers such as SGD, AdaGrad, and RMSProp only if time allows.

3.6.2 Batch Size

The baseline batch size for N-MNIST is set at 100. Heuristically, a large batch size results in a worse generalisation and a higher requirement for CUDA memory, while a small batch size is slower to be trained. A bad generalisation makes the difference between designs obscure, while slow training time makes all experiments proportionally longer, which may not be affordable. The baseline batch size of 100 is the trade-off of the dilemma, though various batch size would be experimented.

For DVS128 Gesture, however, CUDA runs out of memory easily with a batch size of 100. Thus, the batch size of gesture data is restricted to 10 or 20 because of the memory.

Chapter 4

Results and Evaluation

This section following Analysis and Design presents the results of different designs. The aim of this section is to improve accuracy of a temporally coded SNN from the baseline model. Firstly, the baseline model would be fine-tuned to achieve its best performance. All the derivative models would be subsequently tuned and analysed.

4.1 Baseline STNN

The baseline model of STNN achieves a test accuracy of 96% on N-MNIST, a training accuracy of 99.9% and generates 2,800 spikes throughout the network for one classification on average. Its learning curve is shown in Figure 4.1, and all the hyperparameters selected for this baseline are shown in Table 4.1.

Parameter	Default Value
batch size	100
Time constant for synaptic current	0.01
Time constant for membrane potential	0.005
Threshold of the spike generation	1
Learning rate	0.0002
Hidden units	500

Table 4.1: Hyperparameters of baseline STNN

The accuracy of this baseline model (96.0%) is not that far to that achieved by the state-of-art (99.2%). The difference can be due to the following reasons. Firstly, the baseline STNN is train on only a fraction of N-MNIST (1,000 rather than 60,000 samples), but it is surprising that the baseline works so well on such a small fraction of the data. Secondly, the state-of-art blends a CNN architecture, which brings more complexity to fit data. Besides the state-of-art with CNN, SLAYER also trained a model with architecture of 2312-500-500-10, where the number indicates the number of units in a layer and '-' separates different layers. Thus, this SLAYER model has one additional layer with 500 units compared with our baseline model, and it achieves an accuracy of 98.89%. This higher accuracy compared with our baseline model can also be attributed to the additional complexity.

Apart from the accuracy, the STNN baseline model generates 2,700 spikes for each classification. Since the model would have 510 neurons other than input neurons, each neuron contributes 5.3 spikes on average. This is not a huge number because the input feature is relatively small – 32x32x2, and the number of hidden units used is also not large – 500. If a larger feature vector (e.g. 128x128x2 in DVS128 Gesture) is applied on the network, the number of hidden units has to increase, and the membrane potential of the hidden units would receive more spikes from the input layer. As a result, the number of spikes generated per neuron would be easily scaled up.

As it is already known that the state-of-art accuracy of STNN on N-MNIST is 99.20%, the STNN would not be further tuned. Moreover, 99.20% would be the benchmark for the proposed model against for NMNIST data, and 93.64% for DVS128 Gesture data.

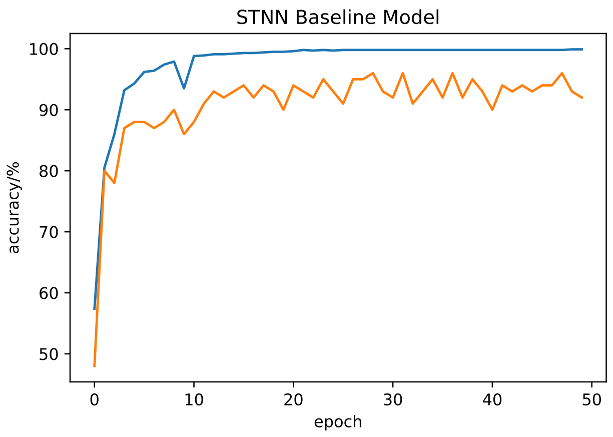


Figure 4.1: Learning curve of baseline STNN.

4.2 Baseline TSNN

The baseline model of TSNN achieves a test accuracy of 87% on N-MNIST, a training accuracy of 95.6%, and generates 510 spikes throughout the network for one classification on average. Its learning curve is shown in Figure 4.2, and all the hyperparameters selected for the baseline are shown in Table 4.2.

Parameter	Default Value
batch size	100
Precision in kernel function	100.0
Maximum time in kernel function	4.0
Time constant of the kernel	2
Threshold of the spike generation	0.5
Learning rate	0.002
Learning rate for pulses	0.06
Hidden units	340
Pulses per layer	10
Penalty for non-spiking neurons	1

Table 4.2: Hyperparameters of baseline STNN. Precision in kernel function and Maximum time in kernel function are parameters controlling the error in the computation of the kernel function and the speed of processing.

From the baseline performance, it can be observed that TSNN finds it harder to fit the training data. This is probably because the complexity of the model is not enough, more hidden units should be added. The training and test loss against epoch is shown in Figure 4.3. The test loss tracks the training loss well, showing no obvious overfitting. Moreover, this baseline model is also applied on the full N-MNIST dataset, achieving a 90% test accuracy, confirming the validity of the 87% reflected only by 100 test data.

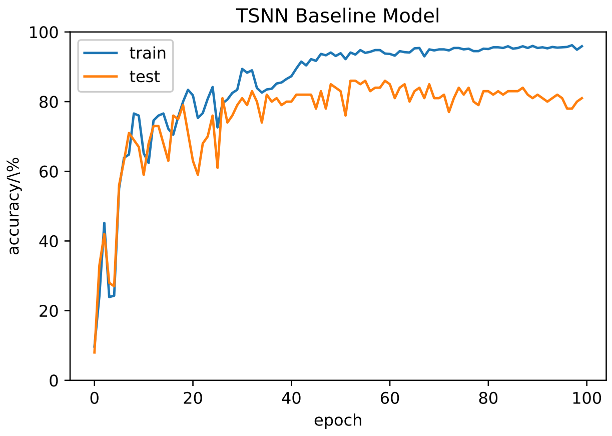


Figure 4.2: Learning curve of baseline TSNN

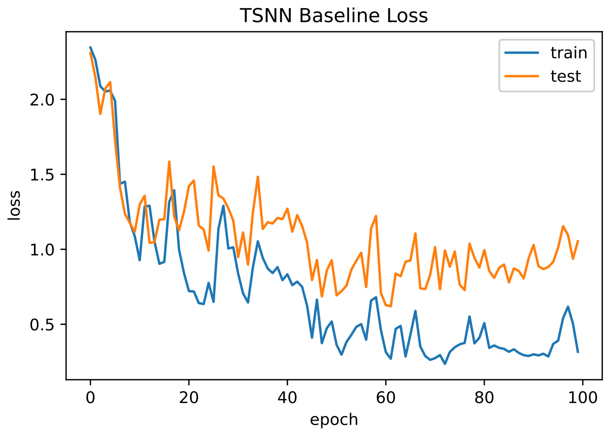


Figure 4.3: Loss of baseline TSNN

Unlike STNN, which has been fine-tuned by SLAYER, TSNN has never been implemented on N-MNIST or DVS128 Gesture to our knowledge. Thus, we will fine-tune it, and find the relationship between the result and those parameters.

4.2.1 Hidden Units

Even with the increase of hidden units, the training accuracy of the model could not achieve 100% accuracy as shown in Table 4.3. There would be two reasons for this to happen. Firstly, as approximation methods are used in the forward propagation of TSNN, the accumulated error could prevent the model to fit the data perfectly. This problem, however, would be in the schedule of future work. Secondly, the information provided solely by the earliest spikes restrains any model to do a classification task:

some features can only be found stochastic – no pattern. This problem could be solved by incorporating more spikes in the network.

Hidden Units	Training Accuracy/%	Test Accuracy/%
100	96.3	85.5
200	94.5	85.7
300	96.6	87.2
400	95.6	83.0
500	97.6	84.4
700	98.3	86.3
1000	97.6	87.0

Table 4.3: Hidden Units v.s. Accuracy.

4.2.2 Kernel Function

The kernel function has a few parameters: precision, time constant, and the threshold.

The precision reflects the quantisation error in the membrane potential calculation. It is set at 100 as a default value, which means that there are 100 time step in 1 time unit. A rough search on the precision ranging from 10 to 200 has found that there is no significant improvement from default precision as shown in Figure 4.4. This is a good news in a way that the quantisation error introduced by the forward propagation algorithm does not affect the performance to a large extent.

Precision	Training Accuracy/%	Test Accuracy/%
10	95.3	86.5
25	95.9	87.3
50	94.0	85.6
100	96.6	87.0
200	93.6	84.4

Table 4.4: Precision v.s. Accuracy

The time constant of the kernel changes both the peak amplitude and the decaying speed of the kernel. It is worth noting that we can link the effect of the time constant only to the change of decaying speed by multiplying a factor to the threshold. Table 4.5 shows that a time constant of 2 is the best choice for TSNN on N-MNIST data. As the input data is constrained to an interval of 0 to 1, this time constant of 2 may be general to other data applied with the same normalisation method.

Time constant	Training Accuracy/%	Test Accuracy/%
0.5	80.7	75.0
1	82.5	78.4
2	96.6	87.0
3	96.5	85.6
4	96.9	83.3

Table 4.5: Time Constant v.s. Accuracy

At last, the threshold is tuned, which indirectly changes the amplitude of the kernel. It can be seen from Table 4.6 that a higher or lower threshold does not perform as well as a threshold of 0.5 and 0.75. Theoretically, however, the threshold should not affect the performance as it only controls the scale of weights. This may imply that those experiments performing worse than the baseline have not converged yet. If the initialisation makes the weights far from the modulus required by the threshold, it would take too long for the weights converge to the right values.

Threshold	Training Accuracy/%	Test Accuracy/%
0.25	83.6	68.2
0.5	96.6	87.5
0.75	94.1	87.3
1	94.4	84.0
1.5	95.2	85.0
2	92.9	80.1

Table 4.6: Threshold v.s. Accuracy

4.2.3 Learning

The batch size is changed in a range of 10 to 200. Moreover, the learning rate is changed accordingly, following a rule that if the batch size is increased by a factor, the learning rate decreases by the same factor. The reason is that a larger batch size makes the gradient calculated in that batch more accurate, thus we could put more faith on that gradient – a larger learning rate. Although we would expect a higher accuracy obtained by models trained by a smaller batch size, they do not as shown in Table 4.7. It may be caused by the fact that although the learning rate should increase while the batch size decreasing, the ratio can be not optimal. For example, when the learning rate is 0.0002 with a batch size of 100, the accuracy is 87%. However, if the learning rate is decreased by a factor of 2 with the same number of samples in each batch, the accuracy decreases to 82%. In summary, TSNN is very sensitive to the learning rate.

Batch size	Learning rate	Training Accuracy/%	Test Accuracy/%
10	0.0002	97.7	83.2
25	0.0005	92.2	82.1
50	0.001	98.1	81.1
100	0.002	96.6	87.5
200	0.004	97.5	83.2

Table 4.7: Batch Size v.s. Accuracy

4.3 Concatenation of data

Concatenation of data aims to incorporate more temporal information to TSNN by adding more input features. There are several ways to incorporate more spikes into the feature vector.

4.3.1 Order of Spikes or Windowed Spikes?

First of all, we need to decide which feature can be passed into the input vector. If we only take one more feature in each pixel, there would be two methods to do so. On one hand, the additional feature can be taken as the second earliest spikes of the input – following the order of spikes. On the other hand, the original data is split into two equal intervals, and the features taken in each interval are the earliest spikes.

Table 4.8 shows the results obtained by the two methods with the TSNN baseline model. The hidden unit number has not been changed because the additional features are viewed as additional information in the same pixel. The pixel number determines the input neuron number, which has not changed, thus nor would the hidden unit number change. The hidden units would, nevertheless, be doubled in the section to verify this argument. It can be seen from Table 4.8 that the two methods perform equally well, though still have not surpassed the baseline result. The reason can be that the regularisation brought by the shared parameter is too strong to optimise the performance.

Method	Training Accuracy/%	Test Accuracy/%
Order	95.6	84.2
Windowing	96.6	84.1

Table 4.8: Different Features v.s. Accuracy

4.3.2 More Hidden Units

The hidden unit number is doubled from the last two designs, and their results are shown in Table 4.9. Both of the results decrease. As the increase of hidden units makes the model more complex, which often brings the training accuracy up rather than down, we may conclude that the two models have not converged. Thus 200 more epochs have been run on those two models, but similar results are obtained. This consequence could come from the fact that the learning rate has not been tuned to the best configuration, which would normally make the model converge to a sub-optimal point. Or the regularisation power brought by more shared weights over-weigh the degree of freedom brought by the additional weights.

Method	Training Accuracy/%	Test Accuracy/%
Order	92.4	78.3
Windowing	92.5	76.0

Table 4.9: Different Features v.s. Accuracy. Models with 600 Hidden Units.

4.3.3 Regularisation

As the poor performance of the model can be attributed to the strong regularisation power of the shared parameter, a softer regularisation method is employed to relate to weights of the original feature and the additional feature. The regulariastion loss is defined as the sum of the squared difference between the two weights. Table 4.10, however, shows no obvious improvement from a shared parameter.

Lambda	Training Accuracy/%	Test Accuracy/%
0.01	84.6	74.0
0.005	87.3	75.4
0.001	96.7	81.5
0.0005	97.9	84.4
0.0001	94.6	83.6

Table 4.10: Regularisation Parameter v.s. Accuracy

4.3.4 More features

More spikes can be added as the additional features, but it is not always a good idea. Firstly, more features can bring an overfitting problem, which may not be compensated by the regularisation method employed. The second reason would concern whether it is the windowing method or the order of spikes method employed. If we are using the windowing method, with more window is split, the information in each window decreases because each window contains fewer spikes. The product of the windows and the number of spikes is about constant. If the new features follow the method of ordering of spikes, then as the order increases, the number of spike will decrease as shown in Table 4.11. Thus each additional feature would bring less information than the previous feature: information converges. As a result, we may infer that the performance may decrease with the number of features increasing because features contain different level of information but have a bounded weight.

As a result, Table 4.12 shows a decreasing trend of accuracy with the increase of features for a model adopting number of window split. The trend is similar to a model with ordering of spikes split, thus the table not presented.

Index of features	1	2	3	4	5	6
Number of spikes	751.85	634.29	579.02	517.36	438.6	327.82

Table 4.11: Number of Spikes in the First Six Earliest Spike Feature

Number of features	Training Accuracy/%	Test Accuracy/%
1	96.6	87.1
2	96.6	84.1
3	90.5	80.4
4	91.5	78.4
5	92.5	78.6

Table 4.12: Number of Features v.s. Accuracy

4.3.5 Limitation of Concatenation of Data

Concatenation of data fails to improve the baseline model. Here are the possible reasons. For new feature constructed by the second earliest spikes, the weight of the first earliest and the second earliest spikes is the same, they are penalised equally. Their significance, however, could be different because it is claimed that the first earliest spikes are the most prominent [2]. For this problem, a higher learning rate could be adopted for the weights associated with] the first earliest spikes to signify the impact of the first earliest spikes in future work.

For the feature constructed by earliest spikes in different windows, the window now is selected in a way that just considering the length of windows but not the periodicity of the data. Although the periodicity in N-MNIST data can be the same, that in various gestures performed by different people can be quite different in DVS128 Gesture. As a result, more statistics should be done before selecting the length of the window in future work.

4.4 Concatenation of STNN and TSNN

This network is constructed by STNN and TSNN joint with a conversion function to receive full data, and have a temporal loss. Before examine this network, we could also directly apply the conversion function at the end of STNN, and gives STNN a temporal loss. This design is firstly reviewed because this may shed a light on how solely the surrogate gradient of the conversion function affects the result.

4.4.1 Temporal Loss for STNN

As analysed in Section 3.4.2 , the gradient of the conversion function calculated by 'autograd' may not be valid. Thus, various surrogate gradients are tested. Provided that the gradient is only non-zero for those entries containing earliest spikes, the expression below only concerns those non-zero gradients.

Firstly, all of the non-gradients are assigned with a constant. Table 4.13 shows that the temporal loss function could do classification with STNN to some extent, though not achieving a good performance. Moreover, the sign of the gradient does not have a significant impact on the performance, and setting the constant to -10 currently achieves the best accuracy – 54%. This poor result may be caused by the even gradients for every spikes, thus the following experiment makes the gradient dependent on spiking time of the earliest spikes.

Constant	Training Accuracy/%	Test Accuracy/%
1	45.5	43.5
10	40.3	39.3
100	29.2	27.4
-1	35	32.9
-10	50	54.8
-100	37.9	35.1

Table 4.13: Different Constants in Surrogate Gradient v.s. Accuracy with a 200-hidden-unit Model

Table 4.14 tests various spiking time dependent surrogate gradients, and finds $-t$ does the best work, though its performance is still bad – 60%. Except assuming that any hidden function not tested will do a better work, we may infer it is not only the surrogate gradient of the conversion function prohibits the network from having a decent performance, but something else which is tolerable for a rate-based loss but not for a temporal loss (e.g. the surrogate gradient for the spike function, and the coarse sampling rate).

Expression	Training Accuracy/%	Test Accuracy/%
t	51.5	51.9
$-t$	66.7	60.6
t^2	15	14.3
$-t^2$	20	22.4
$1/t$	48.2	51.7
$-1/t$	42.5	43.5

Table 4.14: Number of Features v.s. Accuracy with a 200-hidden-unit Model. Exponential functions are also tested, but the classifier barely predicts.

Moreover, the number of spikes generated throughout the network has decreased from typically 2,700, to typically 1,800. It implies that the temporal loss does discourage the generation of spikes, making the classifier more efficient.

4.4.2 Incorporating TSNN

Instead of converting the loss function to temporal by purely, in this section, the STNN read-out layer is replaced by a TSNN layer. As a result, the performance boosts from 60 to 82 from Table 4.15. It implies that TSNN is more suitable to play the role of real-out layer than STNN. Also, other surrogate gradients no shown in Table 4.15 but applied in the previous section all obtained an accuracy near 10%, showing that this configuration may be more sensitive to the gradient.

Definition of SG	Training Accuracy/%	Test Accuracy/%
-1	93	80.6
-10	92.1	79.2
-100	90	76.7
$-t$	92.1	82.6

Table 4.15: Definition of Surrogate Gradient v.s. Accuracy with a 200-hidden-unit Model.

Besides the tuning of surrogate gradients. Table 4.16 shows that there is no significant change with different number of hidden units.

Hidden units	Training Accuracy/%	Test Accuracy/%
100	92.5	78.1
200	92.1	82.8
300	92.7	82.2
400	91.2	80.4
500	90.7	79.5

Table 4.16: Number of Hidden Units v.s. Accuracy

Moreover, the number of spikes generated throughout the network has decreased typically 800, becoming the most efficient model involved with STNN.

4.4.3 Limitation of concatenating STNN and TSNN

This method incorporates most all original inputs in the network and have a temporal loss. The coordination of STNN and TSNN does not yet perform well. The gradient in the TSNN may not mean the same thing in the context of STNN, and the gradient of the conversion function should play a role to compensate that difference, which might not be achieved yet. Moreover, another concern about this concatenation is that STNN performs in a discrete domain as it samples input time, while TSNN performs in a continuous domain, which demands a higher precision.

4.5 Model applied on DVS data

Considering the large size of the features (128x128x2) in DVS128 Gesture, the feature is firstly compressed to a size (64x64x2) before feeding into the network. The filter defined as a minimum filter. None of the baseline nor the proposed network works well for this complex data.

4.5.1 Baseline Model

The baseline model achieves an accuracy of 45.4%. One typical learning curve of the baseline model is shown in Figure 4.4. The training process of the baseline has shown the problem of overfitting because it can be observed that the training accuracy keeps increasing until 80%, while the the test accuracy stuck at 40%. This phenomenon can be caused by overfitting. When the model is applied on the mini N-MNIST data, there are 1,000 training data, and 100 test data – similar amount as DVS128 Gesture. The features in DVS128 Gesture, however, are much more, 4 times more after compression, than those in N-MNIST. Thus, the model has to deal with 4 times more input features with the same amount of data, which could most possibly cause the overfitting. To alleviate this problem, we could augment more data to the training data. The augmentation data will come from the truncated original data: only first 1.5 second of the original data is used to generate the first and second earliest spikes, while the rest 4.5 second data remain unused. As long as we could prove that those currently unused data are periodic to the data used, they could augment to the training dataset.

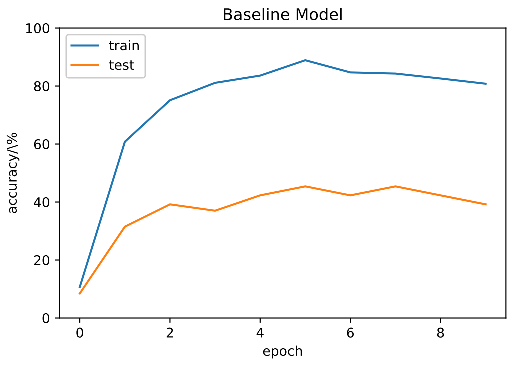
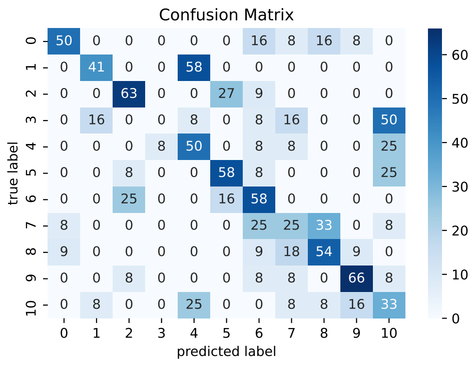


Figure 4.4: Learning Curve of Baseline TSNN

Moreover, the confusion matrix of the baseline is shown in Figure 4.5.



true \ predicted	0	1	2	3	4	5	6	7	8	9	10
0	50	0	0	0	0	0	16	8	16	8	0
1	0	41	0	0	58	0	0	0	0	0	0
2	0	0	63	0	0	27	9	0	0	0	0
3	0	16	0	0	8	0	8	16	0	0	50
4	0	0	0	8	50	0	8	8	0	0	25
5	0	0	8	0	0	58	8	0	0	0	25
6	0	0	25	0	0	16	58	0	0	0	0
7	8	0	0	0	0	0	25	25	33	0	8
8	9	0	0	0	0	0	9	18	54	9	0
9	0	0	8	0	0	0	8	8	0	66	8
10	0	8	0	0	25	0	0	8	8	16	33

Figure 4.5: Confusion Matrix of Baseline TSNN

4.5.2 Concatenation of Data

This model incorporates the second earliest spikes to the feature vector, and the additional feature shares the same weight as the first earliest spikes. Unlike the situation in N-MNIST data, this model performs similar to the baseline, and slightly more, achieving an accuracy of 46.9%. One typical learning curve of this model shown in Figure 4.6.

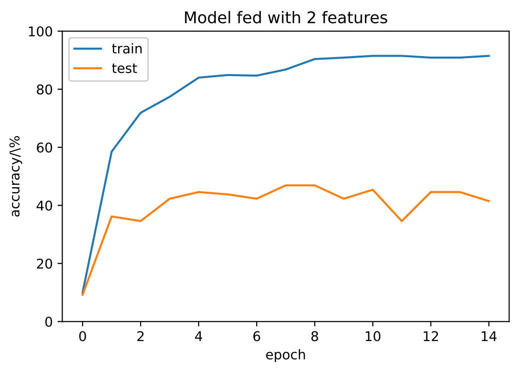


Figure 4.6: Learning Curve of Model Using Additional Features

Moreover, the confusion matrices of this model are shown in Figure **??**.

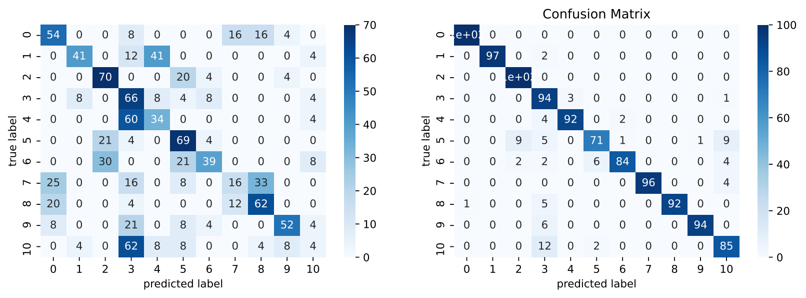


Figure 4.7: Confusion Matrix of the Model Using Additional Features. Left: on test data. Right: on training data.

4.5.3 Evaluation of Models applied on DVS128 Gesture

The performances of all models applied on the gesture data are generally poor, far below the state-of-art performance (93.64%). One of the reason is overfitting as mentioned in Section 4.5.2. Another potential problem lies in the normalisation of input data. Previously, the N-MNIST data would be normalised to a range between 0 and 1 with a maximum time of 0.3 second, while the truncated gesture data are normalised the same interval with a maximum time of 1.5 second. Thus, the temporal information would be more likely to be lost with the same level of quantisation. As a result, the model may be able to fit the training data, while hard to generalise to test data. Moreover, the model adopting the strategy of concatenation of STNN and TSNN has not been applied onto DVS128 Gesture data because of the limit of the CUDA memory.

Chapter 5

Conclusion and Future Work

The project aims to propose, implement, and evaluate a temporal coded SNN that classifies real-world data. TSNN was taken as a baseline and applied onto real-world data (N-MNIST), and it achieved a decent performance – a test accuracy of 87%, showing its potential to classify real-world data. TSNN, however, is limited from its reception of input data: each pixel can only contain one temporal information, resulting in much information truncated. Thus, in order to further improve its performance on real-world data, incorporating more temporal information to the network is necessary. There are generally two ways to incorporate more spikes – either adding more features or having a sequence of features. One of the proposal of the project, concatenation of features, is to feed more features to TSNN. The project analysed and tested various definitions of the additional features and the number of features needed to be added in Section 4.3. This proposal, however, failed to improve from the baseline TSNN with the limitation explained in Section 4.3.5. Besides adding more features to alleviate the loss of spikes in original data, one more ambitious way to incorporate additional data is to make the TSNN sequential. Thus comes the proposal of concatenating STNN and TSNN, and Sequential TSNN. The former takes advantage of the receiving part of STNN, and let one STNN layer as the receiving layer for TSNN: STNN is able to receive almost full information of original data in a sequential way. The biggest challenge of this method is recognised as definition of the conversion function, which allows data from STNN communicate with those in TSNN. The project proposed the definition of both the forward propagation function and the backward propagation of the conversion function in Section 3.4.2 and tested in Section 4.4. This proposal, though makes the efficiency of STNN higher, performs worse than the baseline TSNN. The limitation of the current design is mainly the gradient propagation through the conversion function, and the precision issue in STNN as discussed in Section 4.4.3. Moreover, a sequential TSNN was proposed to make TSNN itself able to receive full information from the original data. In essence, the project designed a hidden state in each temporal spiking neuron, and derived all the forward and backward propagation functions in Section 3.4.3. This proposal, however, remains a problem of defining the input sequence, and an error in calculating the decayed membrane potential in the same Section. Thus, it would not be implemented without those theoretical preparations.

The future work would be suggested to follow the direction of the project has proposed – having a larger input feature vector or make TSNN able to receive sequential data. For the former direction, it is suggested to have attention on the relationship between the different features from the same pixels. This project currently focused on implementing a shared parameter, and a L1 or L2 regularisation loss to force the associated weights similar. Adding a correlation to the two features, however, could still be designed differently. For example, we can train multiple independent SNN fed with multiple features in parallel, and combine their probability distribution in the output layer just like a committee machine. For making TSNN receive sequential, it is encouraged to explore different conversion functions that link STNN and TSNN, or to make the Sequential TSNN proposed in this project realisable. Moreover, for the models applied onto the DVS128 Gesture data, the outcome is discouraging, the best performance achieved so far is about 50%. This, however, could also be viewed as an opportunity to make TSNN more general to the domain of real-world data.

In conclusion, this project has introduced TSNN to the domain of real-world data, and has shown its potential. The direction of how to improve from the baseline have been recognised and analysed, and specific designs tested. Although the outcomes achieved by those designs are mediocre, the way to further improvement has been paved. Moreover, it is shown that TSNN finds it harder to adapt to DVS128 Gesture (46.9% accuracy) data than N-MNIST (87%) data. More work should be done to classify DVS128 Gesture in the future.

Bibliography

[1] S. B. Shrestha, G. Orchard, *"SLAYER: Spike Layer Error Reassignment in Time, "* arXiv:1810.08646, 2018.

[2] I. M. Comsa, K. Potempa, L. Versari, T. Fischbacher, A. Gesmundo, J. Alakuijala, *"Temporal coding in spiking neural networks with alpha synaptic function, "* arXiv:1907.13223.

[3] O. Skorka, and D. Joseph. (2011). *"Toward a digital camera to rival the human eye."* Journal of Electronic Imaging - J ELECTRON IMAGING. 20. 10.1117/1.3611015.

[4] A. Rutkin. *"A Camera That Sees like the Human Eye."* Available from https://www.technologyreview.com/s/518586/a-camera-that-sees-like-the-human-eye

[5] P. Lichtsteiner, C. Posch, and T. Delbruck *"A 128x128 120 dB 15 μs latency asynchronous temporal contrast vision sensor,"* IEEE J. Solid-State Circuits, vol. 43, no. 2, pp. 566–576, 2008.

[6] T. Delbruck, *"Neuromorphic vision sensing and processing,"* in Eur. Solid-State Device Research Conf. (ESSDERC), 2016, pp. 7–14.

[7] T. Delbruck and P. Lichtsteiner, *"Fast sensory motor control based on event-based hybrid neuromorphic-procedural system,"* in IEEE Int. Symp. Circuits Syst. (ISCAS), 2007, pp. 845–848.

[8] I.-A. Lungu, F. Corradi, and T. Delbruck, *"Live demonstration: Convolutional neural network driven by dynamic vision sensor playing RoShamBo,"* in IEEE Int. Symp. Circuits Syst. (ISCAS), 2017.

[9] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal dynamics: From single neurons to networks and models of cognition.* Cambridge University Press, 2014.

[10] G. Gallego, T. Delbruck, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. Davison, J. Conradt, K. Daniilidis and D. Scaramuzza *"Event-based Vision: A Survey",* arXiv:1904.08405, 2019.

[11] W. Maass. *"Network of spiking neuron: the third generation of neural network models Neural Network, 10 (9) (1997), pp. 1659-1671"*

[12] J. Kaiser, H. Mostafa, and E. Neftci, *"Synaptic plasticity for deep continuous local learning",* preprint arXiv:1812.10766, 2018.

[13] E. Neftci, H. Mostafa, and F. Zenke. *"Surrogate gradient learning in spiking neural networks",* arXiv preprint arXiv:1901.09948, 2019

[14] R. VanRullen, R. Guyonneau, S. Thorpe. *"Spike times make sense"*. Trends in Neurosciences, 28(1), 1–4. (2005)

[15] R. J. Williams and D. Zipser, *"A learning algorithm for continually running fully recurrent neural networks,"* Neural computation, vol. 1, no. 2, pp. 270–280, 1989.

[16] I. Goodfellow and Y. Bengio and A. Courville,*"Deep Learning"*, MIT Press, 2016. Available from: `http://www.deeplearningbook.org`

[17] F. Zenke and S. Ganguli, *"SuperSpike: Supervised Learning in Multilayer Spiking Neural Networks,"* Neural Computation, vol. 30, no. 6, pp. 1514–1541, Apr. 2018.

[18] G. Bellec, D. Salaj, A. Subramoney, R. Legenstein, and W. Maass, *"Long short-term memory and Learning-to-learn in networks of spiking neurons,"* in Advances in Neural Information Processing 23 Systems 31, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 795–805.

[19] Kandel, E. R., Schwartz, J. H., Jessell, T. M. (2000). *"Principles of neural science (4th ed.)"*. New York: McGraw-Hill.

[20] B. Gardner, I. Sporea, and A. Gru ning, *"Learning Spatiotemporally Encoded Pattern Transformations in Structured Spiking Neural Networks,"* Neural Comput, vol. 27, no. 12, pp. 2548–2586, Oct. 2015.

[21] W. Nicola and C. Clopath, *"Supervised learning in spiking neural networks with force training,"* Nature communications, vol. 8, no. 1, p. 2208, 2017

[22] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor. *"Converting static image datasets to spiking neuromorphic datasets using saccades"*. Frontiers in Neuroscience, 9:437, 2015.

[23] A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. d. Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza, J. Kusnitz, M. Debole, S. Esser, T. Delbruck, M. Flickner, and D. Modha. *"A low power, fully event-based gesture recognition system"*. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 2017.

[24] D. P. Kingma, J. Ba, *"Adam: A Method for Stochastic Optimization, "* arXiv:1412.6980.

[25] N. S. Keskar, R. Socher, *"Improving Generalization Performance by Switching from Adam to SGD, "* arXiv:1712.07628.