Text classification in memristor-based spiking neural networks

Jinqi Huang 1,∗, Alexantrou Serb 2, Spyros Stathopoulos 2 and Themis Prodromakis 2

1 Electronics and Computer Science, University of Southampton, Southampton, United Kingdom
2 School of Engineering, University of Edinburgh, Edinburgh, United Kingdom
∗ Author to whom any correspondence should be addressed.
E-mail: J.Huang@soton.ac.uk

Keywords: text classification, sentiment analysis, natural language processing, spiking neural networks, memristors, online learning, offline classification

Abstract
Memristors, emerging non-volatile memory devices, have shown promising potential in neuromorphic hardware designs, especially in spiking neural network (SNN) hardware implementation. Memristor-based SNNs have been successfully applied in a wide range of applications, including image classification and pattern recognition. However, implementing memristor-based SNNs in text classification is still under exploration. One of the main reasons is that training memristor-based SNNs for text classification is costly due to the lack of efficient learning rules and memristor non-idealities. To address these issues and accelerate the research of exploring memristor-based SNNs in text classification applications, we develop a simulation framework with a virtual memristor array using an empirical memristor model. We use this framework to demonstrate a sentiment analysis task in the IMDB movie reviews dataset. We take two approaches to obtain trained SNNs with memristor models: (1) by converting a pre-trained artificial neural network (ANN) to a memristor-based SNN, or (2) by training a memristor-based SNN directly. These two approaches can be applied in two scenarios: offline classification and online training. We achieve the classification accuracy of 85.88% by converting a pre-trained ANN to a memristor-based SNN and 84.86% by training the memristor-based SNN directly, given that the baseline training accuracy of the equivalent ANN is 86.02%. We conclude that it is possible to achieve similar classification accuracy in simulation from ANNs to SNNs and from non-memristive synapses to data-driven memristive synapses. We also investigate how global parameters such as spike train length, the read noise, and the weight updating stop conditions affect the neural networks in both approaches. This investigation further indicates that the simulation using statistic memristor models in the two approaches presented by this paper can assist the exploration of memristor-based SNNs in natural language processing tasks.

1. Introduction

Spiking neural networks (SNNs) [1], as the third generation of artificial neural networks (ANNs) [2], have become a widely-researched topic recently. Their spike-based nature and temporal information coding scheme bring advantages in biological plausibility [3, 4], sparsity [5], and stochasticity [6], further making them power-efficient tools compared with traditional ANNs. Firstly, in traditional ANNs, the algorithm core is the dot product of input vectors and weight matrices. While in SNNs, a matrix multiplication turns into an addition because, in a single time step, each input can only be 1 or 0, resulting in less computational complexity [7]. Besides, conventional ANNs require computation for every input, while SNNs reduce the number of operations by only processing received spike events. The high sparsity of the spike trains not only further reduces energy consumption with fewer operations [7], but also increases robustness if the stochastic rate coding method is used [6]. To be more explicit, when using stochastic rate coding in SNNs, multiplication can be performed implicitly by using stochastic techniques. A typical case is where 2X...
stochastic bitstreams enter an AND gate. The statistics of computation dictate that the probability of a ‘1’ at the output is the product of probabilities of ‘1’ at the input streams, thus implementing multiplication in the probability domain. Naturally, the number of ‘1’s within a time window along the output bitstream can be summed up to give an (approximate) reading of the probability number. In a similar manner, in SNNs, multiplication can be implemented implicitly, but more importantly, fault-tolerantly. Based on different neural coding methods, the same spike train can represent different information. Therefore, they have been widely used in applications spanning across different areas, such as hand gesture detection [8, 9], gait detection [10], financial time series prediction [11], music composer classification [12], real-time signal processing [13], disease detection [14], and robotics [15–17]. Neural coding methods including rate coding [18], temporal coding [19], and population coding [20] make the same spike train can deliver different information: the rate coding considers most information is delivered by the count of spikes over time, the temporal coding also considers the precise spike timing, and the population coding interprets spikes by using the joint states of multiple neurons. Among all, a vast majority of SNN designs [7, 21–29] applied the rate coding because it has the simplest spike encoding and decoding. In this paper, we will only discuss rate coding.

Despite the promising potential, hardware implementations of SNNs in conventional Von-Neumann processors can hardly compare with the biological systems regarding energy efficiency. This performance gap mainly stems from the frequent data movements between processing units and the memory to fetch SNN-specific internal variables such as membrane voltages [30, 31]. In-memory computing allows an alternative approach to implementing SNNs in hardware. The computation is executed locally without frequent data movements by combining the processing units and the memory. This working scheme removes the restriction set by the limited communication bandwidth that commonly exists in Von-Neumann processors. An SRAM-based in-memory computing system [32] has reported having the energy efficiency 10x better than google’s TPU [33]. Based on this idea, many dedicated SNN hardware designs have been developed, from fully digital [34, 35] to mixed-signal [36], from large-scale computation platforms [37] to systems-on-wafer [38]. Some designs have been reported to achieve biological speed [35] or multiple orders of magnitude superior to their conventional Von-Neumann predecessors [34].

Meanwhile, the development of emerging non-volatile technology has also seen growth in the past few years. Memristors are known as one of the most mature emerging resistive memory devices among all. The extreme downscaled size [39], the large-scale integrability [40], multi-bit storage [41], crossbar arrangement [42], ns-scale switching speed [43] and high power efficiency [44] make memristors good candidates to replace SRAM in in-memory computing applications [45–47]. Moreover, memristors behave similarly to biological synapses in neural networks resulting from their inherent physical properties inspiring their use in SNNs. As examples, [48, 49] incorporated memristors as synapses in SNNs to perform unsupervised learning using spike-timing-dependent plasticity [50]. [25, 27, 28, 51] demonstrated pattern recognition and image classification in memristor-based SNNs.

Text classification tasks, known as tasks that assign labels to input text to group into different categories, are one of the classic applications of natural language processing. Recent exploration of performing text classification tasks focuses on the deep learning solution, including using feed-forward NNs [52, 53], RNNs [54, 55], CNNs [56, 57], attention [58], hybrid models [59, 60], and transformers [61]. However, only a few works reported memristor-based SNNs for text processing tasks: Design [62] employed phase-change memory (PCM), a type of memristor device, to accelerate SNN for language modelling, and Design [63] performed a pattern generation task in recurrent SNNs with PCM models using e-prop learning rule [64]. One of the main challenges is the costly training of memristor-based SNNs for text processing tasks. Firstly, gradient-based learning rules, regardless of local (e.g. e-prop) or non-local (e.g. surrogate gradient descent [65, 66]), require averaging the accumulated errors over the spike train window that is used to represent a single continuous value. This method considers the effects of every single spike when updating weights. It is inefficient regarding the computational speed and the space efficiency, especially when the length of spike trains for representing a single numerical value is very long and when memristors are involved in the design. Besides, text classification tasks usually have high-dimensional inputs, which require large memory allocation in implementing memristor-based SNNs. The typical dimension reduction technology used in traditional ANNs such as word embeddings, (e.g. GloVe [84] and, are commonly used to map high-dimensional inputs to dense lower-dimensional representation for better space efficiency. In conventional ANNs, a word embedding layer serves as a lookup table to obtain the dense representation of the input texts. Pre-trained word embeddings are usually fine-tuned later with other network parameters in training to improve classification accuracy. During the training process, they are treated as linear layers using backpropagation (BP) [67]. However, in SNNs, the dense word representation needs to be converted to spike trains as the input to the network [68, 69]. So far, there is no theoretical support regarding how to train word embeddings in SNNs. Lastly, reading and writing memristor arrays can also add variations due to the read
noise and the write variation. One solution to these issues is to run the training in a software simulator with realistic memristor models before starting the hardware design. Compared with training in hardware, this solution offers a fast data process and intuition for the system's performance. [70] has introduced an algorithm-level simulator for memristor-based SNNs in memristor models, and Design [51] also used this solution to simulate recurrent SNNs in PCM models.

In this paper, we aim to solve the issues of high-dimensional inputs, the costly training, and the memristor non-ideality in expanding memristor-based SNNs in text classification applications by extending previously presented algorithm-level simulator NeuroPack [70] for a bespoke simulation framework with Pytorch [71] to enable GPU compatibility. We take two paths to obtain trained SNNs with memristor models: (1) by converting a pre-trained ANN to a memristor-based SNN, or (2) by directly training a memristor-based SNN. Both approaches train the network using gradient descent-based learning rules without considering the effect of every single spike to improve the computational speed and the space efficiency. In this way, we can also treat a word embedding layer as a linear layer in a conventional ANN and train it using BP. Then, we showcase a sentiment analysis task in the IMDB movie reviews dataset [72] to validate both approaches. To the best of our knowledge, this is the first demonstration of a text classification task performed in a SNN with a realistic memristor model. Lastly, we summarise the classification accuracy from both approaches and investigate how global parameters affect the system performance in both approaches.

The contributions of this work are summarised below:

(a) Solving the issues of the high-dimensional inputs, the costly training, and the memristor non-ideality for utilising memristor-based SNNs in text classification tasks by presenting and benchmarking two approaches to obtain trained memristor-based SNNs for text classification tasks.

(b) Developing a simulation framework with an empirical memristor model to demonstrate the first text classification task in SNNs with a realistic memristor model.

(c) Investigating system sensitivity to global parameters the spike train length, the weight-updating stop condition \textbf{R tolerance}, and the read noise, and explaining the effects of these parameters to the system when taking two different approaches.

To achieve these goals, we first introduce the overview of the two approaches in section 2.1 and explain the details in sections 2.2 and 2.3. Following are the method to add the memristor model in the simulation framework in section 2.4 and the weight updating scheme in section 2.5. Section 3 displays all experimental results for the text classification task in the simulation framework with memristor models, and section 4 summarises the paper.

2. Methods

2.1. Methodology overview

The main difference between ANNs and SNNs is the conveyance method: ANNs employ continuous values, whereas SNNs use 0-or-1 spikes. As a result, the basic idea to bridge the gap between two neural network architectures is to find the relations to map continuous values to spikes. [73] suggested that the spike rates of an SNN are proportional to the ReLU [74] activation outputs of the ANN equivalents with an error term that is ignorable in shallow networks. Given that inputs and weights are restricted within [0, 1] in a memristor-based SNN, the ReLU activation outputs are always non-negative. Therefore, the spike rates of an SNN are proportional to the input currents. [75] further proved that for neurons reset by subtraction, the larger the membrane voltage threshold, the smaller the error term. Additionally, weight normalisation [76] and adaptive threshold [77] are also commonly used in ANN-to-SNN conversion to match the accuracy.

These past works lay the theoretical foundation of the two approaches to obtaining trained memristor-based SNNs proposed in this paper.

Figure 1 gives the diagram of the two approaches: Starting from an untrained ANN, approach 1 trains the ANN using typical ANN-based learning rules, then converts the trained ANN to a trained SNN; approach 2 firstly converts the ANN to its equivalent SNN before training with ANN-based learning rules. After obtaining the trained SNN, memristor models are added to store the weights. In approach 1, this step is as simple as mapping the weights in memristor arrays as conductance; in approach 2, memristor resistance states (RSs) are trained alongside the whole SNN. Therefore, the evolution of RSs can be observed during the training process when using approach 2.

Now we walk through the workflow of the designed simulation framework performing the sentiment analysis task to explain the two approaches. Figure 2 shows the network we employed in the sentiment analysis task in the IMDB reviews dataset. After being tokenised using the basic English tokeniser and...
Figure 1. Two approaches to obtaining trained SNNs: approach 1 trains ANNs first (coloured orange), converts trained ANNs to equivalent SNNs (coloured green), and then maps SNN weights to memristors (coloured red); approach 2 converts ANNs to SNNs (coloured green), adds a memristor model (coloured red), and then trains memristor-based SNNs directly using ANN-based learning rules (coloured orange).

Figure 2. Workflow of the simulation framework for performing the sentiment analysis task in the IMDB review dataset. Inputs, ANN structure, SNN-specified structure, and memristor-related modules are coloured blue, orange, green and red, respectively. $b$, $s$, $e$, and $o$ represent batch size, sentence length, word embedding dimensions, and output dimensions, respectively. We use a movie review, ‘A wonderful little production’, as an example to explain the input pre-process procedure: firstly, input sentences are tokenised to lists of words (‘[A’, ‘wonderful’, ‘little’, ‘production’]’ in the example). Secondly, the word lists are converted to lists of word IDs (‘[6, 385, 120, 370]’ in the example). Sentences in the same batch are padded to have the same length. The lists of words are further transformed to a one-hot vector, and padded one-hot vectors of the sentences in the same batch are packed into an input representation tensor with the size of ($b \times s \times v$). Notably, in Pytorch, the word ID lists do not need to be converted to one-hot representation.

transformed to word IDs, sentences in the same batch are padded to have the same sentence length and packed into an input one-hot representation tensor with the size of ($b \times s \times v$) (where $b$, $s$, $v$ are batch size, sentence length, and vocabulary size, respectively). The one-hot representation tensor is then fed to a word embedding layer to generate a dense word representation whose size is ($b \times s \times e$) (where $e$ is the word embedding dimension), followed by an average pooling layer to obtain an averaged sentence representation across the whole sentence by squeezing the sentence dimension.

In approach 1, the averaged sentence representation with the size of ($b \times e$) is sent straight to a linear layer whose weights’ size is ($e \times o$) (where $o$ is the output dimension). Specifically, there is only one output neuron in the sentiment analysis task because we only have two categories: ‘positive’ or ‘negative’. The binary cross-entropy loss is calculated once the inference results are obtained, and the errors are back-propagated to update weights in the linear and word embedding layers. After the network is trained, the conversion to an SNN starts. Firstly, the averaged sentence representation is transformed to Poisson spike trains which are...
delivered to the single-layer SNN afterwards. We use the leaky integrate-and-fire (LIF) neuron model \cite{78} in this task. Next, the weights in the SNN are mapped in memristor arrays. When executing an inference, weights are loaded from the memristor arrays. The output neuron membrane voltages are calculated according to the mathematical expression of the neuron model, and the firing states are determined by comparing the voltage with the threshold. The spike rates within a time window give the final inference results: if the spike rate is higher than 50\%, it yields a ‘positive’ result; otherwise, a ‘negative’ result is given.

In approach 2, the averaged sentence representation is likewise transformed to Poisson spike trains, and memristor arrays are randomly initialised. After an inference completes, the errors are back-propagated according to the same ANN-based learning rule used in approach 1. The weight changes are converted to the memristive RSs to be used to trigger RS updates. The errors are further back-propagated to the word embedding layer to generate new spike trains in the following iterations.

In the next two subsections, more details that need to be taken into consideration in implementing two approaches in the simulation framework will be introduced.

**2.2. Approach 1: converting a pre-trained ANN to an SNN**

Weights in the linear and word embedding layers need to be constrained in [0, 1] when training the ANN using approach 1. With this restriction, when converting the ANN to the SNN, weights can be represented according to the following equation:

\[ w = \frac{G - G_{\text{min}}}{G_{\text{max}} - G_{\text{min}}} \]  

where \( G, G_{\text{min}}, G_{\text{max}} \) are the current, the minimum, and the maximum memristor conductance, respectively. The averaged sentence representation is also restricted within [0, 1]. Therefore, when converting the ANN to the SNN, spike trains can be generated using the continuous values in the averaged sentence representation as firing rates. \cite{68} proposed a method to determine whether to fire a spike at a certain time step to generate a Poisson spike train using the equation shown below:

\[ P_t = \begin{cases} 
1 & \text{if } x_{t} > x_{\text{random},t} \\
0 & \text{otherwise}
\end{cases} , \quad 0 \leq t \leq T \]  

where \( P_t \) gives the spike generation results of the averaged sentence representation in the given time step \( t \) in the spike train that uses \( T \) time steps to represent a single continuous value, \( x_t \) is the continuous value in the averaged sentence representation, and \( x_{\text{random},t} \) is the random value generated in the time step \( t \) within [0, 1]. The averaged sentence representation values \( x_t \) as the output of the neurons in the average pooling layer range between [0, 1] in figure 2. These values are compared at each time step against random variables \( x_{\text{random},t} \) drawn from a uniform distribution and based on the comparison in equation (2) they determine the output of the spike. A matrix of random values with the size of \((e \times T)\) is initialised before the binary word embedding conversion. The averaged sentence representation is then compared to the random-value matrix to generate Poisson spike trains stored in a matrix with the size of \((e \times T)\) as inputs of the single-layer SNN.

Because the weights in the linear layer \( W \) and the averaged sentence representation \( x_t \) are both non-negative when training the ANN, the outputs of the linear layer calculated by the equation \( y = \sum W_{x_t} \) are never negative. A negative constant offset \( C \) need to be added to the ANN outputs when using the sigmoid function to map the continuous-valued outputs to probability. If the negative offset \( C \) is not used, the non-negative sigmoid function input will result in an output probability never smaller than 50\%. Thus, to make the probability mapping functional, the term passed to the sigmoid function becomes \((y + C)\) instead of \(y\). When the term \((y + C)\) equals 0, the inference result is right on the boundary of dividing two categories. Therefore, \( C = -\sum W_{x_t} = -0.5 \times 0.5 \times e \) where \( W_t \) and \( x_t \) are both the theoretical averaged values (0.5 and 0.5, respectively). However, when converted to the SNN, the criterion for dividing categories is the output spike rates rather than the sigmoid output. Therefore, a sigmoid function is not used in SNN inference, and no offset is needed in the converted SNN.

The membrane voltage threshold is an extra parameter not used in an ANN. The threshold value needs to be carefully chosen to achieve comparable classification accuracy in the converted SNN. Membrane voltages that are either much higher or much lower than the threshold can cause information loss \cite{77}. A threshold value ‘in the middle’ can potentially cause the least information loss. In converted SNNs, the membrane voltages range is known since both weights and word embeddings are constrained within [0, 1]. When all inputs and weights are equal to ‘1’ along the whole spike train, this gives the highest theoretical firing rate. As explained in \cite{75}, the output firing rates of an SNN are proportional to the inputs. Therefore, when we keep the weights equal to ‘1’ and change the inputs to half ‘1’ and half ‘0’, the firing rate of the output neuron is supposed to be 50\%-setting the membrane voltage of the output neuron in this scenario as the threshold.
with the value of $0.5 \times e$ is a good starting point. It guarantees maximum firing when the inputs are maximised and that under any initial conditions, the output neuron will losslessly accumulate membrane potential in a manner commensurate to the strength of the inputs without saturation or clipping. The threshold value needs further fine-tuning based on the approximated value to achieve better performance.

### 2.3. Approach 2: training an SNN directly

Training SNNs with ANN-based gradient descent learning rules [79] can be problematic due to the non-differential property of membrane voltages. A common solution is to employ surrogate derivatives [65, 66] by using straight-through estimators [80–82]. This work proposed an alternative solution, which stems from ANN-based gradient-based algorithms, to avoid directly dealing with non-differentiable functions. In this work, we use the LIF neuron model [78] with reset by subtraction mechanism:

$$ V_t = V_{t-1} + \sum W_s x_t - Z_{t-1} V_{th} $$  \hspace{1cm} (3)  

$$ Z_t = h(V_t - V_{th}) $$  \hspace{1cm} (4)  

where $V_t$, $x_t$, and $Z_t$ denote membrane voltages, input spikes and output spikes at time step $t$, respectively. $V_{th}$ is the membrane voltage threshold, $W_s$ represents the weights in the single-layer SNN, and $h(x)$ is the Heaviside step function. For this two-category classification task, we use the binary cross-entropy cost function with a sigmoid layer:

$$ E = -(\hat{y} \log y + (1 - \hat{y})(1 - \log y)) $$  \hspace{1cm} (5)  

where $E$ represents the binary cross-entropy loss, $\hat{y}$ and $y$ are labels and the probabilities given by the network, respectively.

As the learning rule, we choose Adaptive Gradient Algorithm (Adagrad) [83], a gradient-based learning rule with an adaptive learning rate incorporating the past gradients. The learning rule is explained by the equations below:

$$ s \leftarrow s + g^2 $$  \hspace{1cm} (6)  

$$ \theta \leftarrow \theta - \eta \frac{g}{\sqrt{s + \epsilon}} $$  \hspace{1cm} (7)  

where $g$ is the gradient of the objective function over some variable of interest $\theta$, $s$ is a state variable that accumulates the square of the gradients, and $\epsilon$ is a small value to avoid dividing by 0.

Now we walk through the steps to acquire the final expression of the weight updates. For the same reason explained in section 2.2, the output neuron firing rate is always non-negative. Therefore, the output firing rate needs to add a negative offset before being passed to a sigmoid function. The offset chosen here is $-0.5$.

The output probabilities of two categories used in calculating the loss are given by passing the output firing rate with a negative offset to a sigmoid function:

$$ a = r + C $$  \hspace{1cm} (8)  

$$ y = \frac{1}{1 + e^{-a}} $$  \hspace{1cm} (9)  

where $r$ is the output neuron firing rate, $C$ is the negative offset, and $a$ is a temporary variable to save the sum of the firing rate and the offset. The output firing rates of an SNN are proportional to the equivalent ANN activation outputs in the equivalent ANN, as proven in [75]. Given that the outputs are always non-negative, we further proved that in a memristor-based SNN, the output firing rates are proportional to the equivalent ANN outputs as follows (please see the derivation in supplementary material section 1):

$$ r = \frac{V_c}{V_{th}} \frac{V_t - V_0}{V_{th} \cdot T} $$  \hspace{1cm} (10)  

Suppose the equivalent ANN has the mathematical expression shown below:

$$ V_c = \sum W_s x_c $$  \hspace{1cm} (11)  

where $V_c$ and $x_c$ are continuous-valued ANN outputs and inputs, and $V_0$ is the initial membrane voltage of a spiking neuron when processing a new sample. $V_0$ is 0 in this work because we reset the membrane voltage
accumulator before a new inference. The error term \( V_t/(V_{th} \cdot T) \) in equation (10) depends on the final membrane voltage \( V_t \) at the end of the spike window \( T \), and \( V_t \) is the continuous value that the spike train in the spike window \( T \) represents. Intuitively speaking, \( V_t \) implicitly depends on \( V_c \). Therefore, the error term \( V_t/(V_{th} \cdot T) \) is weakly correlated to \( V_c \), and this weak correlation can potentially introduce an error in the conversion of an ANN into an SNN. This error can be accumulated in a deep neural network, but it is ignorable in a shallow network [75]. If we ignore the weak dependency of the error term to \( V_c \) and use two constants \( \alpha \) and \( \beta \) to replace the error term and the constant-coefficient \( 1/V_{th} \), we can describe the output firing rates differently:

\[
r = \alpha V_c + \beta. \tag{12}
\]

Therefore, the derivative of \( r_t \) with respect to \( V_c \) equals to \( dr/dV_c = \alpha \). As a result, the gradient of the cost function over the weights in the single-layer SNN is expressed as follows using the chain rule:

\[
\frac{dE}{dW_c} = \frac{dE}{da} \cdot \frac{da}{dV_c} \cdot \frac{dV_c}{dW_c} = (y - \hat{y}) \cdot \alpha \cdot x_c. \tag{13}
\]

Similarly, the word embedding matrix can be updated by further back-propagating the errors:

\[
\frac{dE}{dW_e} = \frac{dE}{da} \cdot \frac{da}{dV_c} \cdot \frac{dV_c}{dx_c} \cdot \frac{dx_c}{dW_e} = (y - \hat{y}) \cdot \alpha \cdot W_e x_c, \tag{14}
\]

where \( W_c \) denotes the parameters in the word embedding matrix, and \( x_c \) is the one-hot input word representation.

In Adagrad, the change amounts are decided by the term \( g/(\sqrt{s} + \epsilon) \). Both \( g \) and \( \sqrt{s} \) are proportional to \( \alpha \). Therefore, \( \alpha \) can be cancelled out in the expression. The final expression to be used in Adagrad without \( \alpha \) is simplified as follows:

\[
g_e = (y - \hat{y}) \cdot x_c \tag{15}
\]

\[
g_c = (y - \hat{y}) \cdot W_e x_c. \tag{16}
\]

Compared to accumulating errors through the whole spike train using surrogate gradient descent, which can be expressed as the equation below, our method improves the computational speed and the space efficiency by skipping the step of accumulating errors across the spike window.

\[
\frac{dE}{dW_s} = \frac{dE}{da} \cdot \frac{da}{dz_t} \cdot \frac{dz_t}{dy_t} \cdot \frac{dy_t}{dV_t} \cdot \frac{dV_t}{dW_s} = (y - \hat{y}) \cdot \frac{1}{T} \sum_t h'(V_t - V_{th}) x_t. \tag{17}
\]

If using a binary activation [82] shown in equation (18) to replace the Heaviside step function and choosing a threshold equal or larger than the half of the maximum possible membrane voltage to avoid information loss, the gradient calculated by surrogate gradient descent (equation (17)) can be simplified to \((y - \hat{y}) \cdot x_c/(2V_{th})\). Cancelling out the constant coefficient \(1/(2V_{th})\) gives the same simplified gradient expressions shown in equations (15) and (16).

\[
h'(V_t - V_{th}) = \begin{cases} 
\frac{1}{2V_{th}} & \text{if } 0 < V_t < 2V_{th} \\
0 & \text{otherwise}.
\end{cases} \tag{18}
\]

### 2.4. Adding memristor models

This work employs the statistical memristor model proposed by [84] to predict the behaviour of memristors during the training or inference process in memristor-based SNNs. The switching dynamics of memristor devices are shown below for convenience:

\[
\frac{dR}{dt} = \begin{cases} 
A_p(-1 + \exp(|v|/t_p))h(r_p(v) - R)[r_p(v) - R]^2 & \text{if } v > 0 \\
A_n(-1 + \exp(|v|/t_n))h(R - r_n(v))[R - r_n(v)]^2 & \text{otherwise}
\end{cases} \tag{19}
\]
where \( A_{p,n} \) denotes the scaling factor, the term \((-1 + \exp(|v|/t_{p,n}))\) reflects the exponential increment between the switching rate and the bias voltage \( v \), and the rest of the equation portrays the dependency on the current memristive RSs and the dynamic upper/lower boundaries \( r_{p,n} \) represented as the equation below with fitting parameters \( a_{0p,n} \) and \( a_{1p,n} \):

\[
r_{p,n} = a_{0p,n} + a_{1p,n}v.
\]  

(20)

In practice, all parameters can be extracted using the method proposed by [85] with the aid of characterisation tools [86, 87]. This work takes these parameters to predict the memristive RSs given the current memristive RSs and the bias voltages. Using Pytorch, all memristive RSs are updated and accessed simultaneously. Given the pulse voltage \( v \) and the pulse-width \( pw \), the duration is converted to iteration times given by \( N = \text{int}(pw/dt) \) and memristive RSs are updated in loops with equation \( \Delta R = \sum(dR/dt) \).

2.5. Weight updating scheme

After obtaining the expected weights to map in memristor arrays, the essential step is to convert the expected weights to expected memristive RSs. Rewriting equation (2) gives the expression of the expected memristive RSs given the expected weights:

\[
R = \frac{1}{w\left(\frac{1}{R_{\text{min}}} - \frac{1}{R_{\text{max}}}\right)^{\frac{1}{R_{\text{max}}}}} + \frac{1}{R_{\text{max}}}
\]  

(21)

where \( R \) denotes the expected memristive RS, and \( R_{\text{min}} \) and \( R_{\text{max}} \) are the lower and the upper RS boundaries that are the minimal/maximal of a set of \( r_{p,n} \) values with different bias voltage \( v \). As shown in equation (19), the switching dynamics of memristor devices are non-linear and governed both by bias voltages and the current memristive RSs. Therefore, it is challenging to trigger memristors to reach certain RSs. [70] proposed a ‘predict-write-verify’ loop to find the parameter sets (including bias voltages and the pulse-width) that can lead to memristive RSs closest to expected. Given multiple sets of bias voltages and the pulse-width, firstly, predict all resulting memristor RSs, and find the one that contributes to the shortest distance to the expected RSs. Next, apply the pulse with this chosen parameter set, and verify the actual new RSs. Repeat the same procedure until the differences between the actual RSs and the expected RSs are within an acceptable range defined by parameter \( \text{R tolerance} = |R_{\text{expected}} - R_{\text{actual}}|/R_{\text{expected}} \). Considering a possible scenario of infinite loops caused by the read noise, another parameter \( \text{maxN} \) is also supplied to restrict the maximum iteration times.

When implementing this weight updating scheme, a critical design choice is to choose pulsing parameter sets. Suppose the provided pulsing parameter can only lead to small changes in RSs. In that case, it will take multiple steps to reach the expected RSs and potentially lead to a significant error when reaching the maximum iteration times set by the \( \text{maxN} \). If parameter sets only allow large-step RS changes, the targeted RSs may be ‘missed’. Therefore, a practical solution is to match the required RS changes and the estimated RS changes that the selected parameter sets can result in. For approach 1, the required RS changes usually are relatively large. Therefore, parameter sets that can lead to large-step and small-step RS changes are both needed: the former is used to make RSs converge quickly, and the latter is for fine-tuning. Approach 2, however, only needs parameter sets that lead to relatively smaller-step RS changes because the required RS changes in each training step are relatively small. To make sure the weight updating can meet the stop condition set by the \( \text{R tolerance} \), for both approaches, parameter sets that can lead to RS update smaller than the RS margin defined by the \( \text{R tolerance} \) need to be chosen.

3. Results

3.1. Experiment overview

We demonstrate a sentiment analysis task with the IMDB movie reviews dataset to validate the two approaches to obtaining a trained memristor-based SNN [88]. The dataset includes 25k highly polar movie reviews training samples and 25k test samples. Inspired by the GitHub project [89], we firstly use the training samples to build a vocabulary to pre-process the input data (figure 3(a)). We count the word frequency in the training samples and put those words that appear over 10x times into the vocabulary. This step creates a vocabulary of 20 473 words. Next, we split the original training set into a sub-training set with 17.5k samples and a validation set with 7.5k samples. After that, an input review is tokenised using the basic English tokeniser and further converted into a word ID vector. The one-hot representation of the word ID vector is then fed to the neural network to start an inference. The word embedding layer contains pre-trained GloVe word embeddings [90] with the embedding dimension of 100. For the training and the inference processes,
Figure 3. (a) Building a vocabulary using the training samples. Step 1: Create a look-up table for storing word frequency. Step 2: Omit the words that appear less than 10x to reduce the vocabulary size. Step 3: Iterate through all samples to transform them into word IDs using the vocabulary. Words are tagged as `[unk]` if they cannot be found in the vocabulary. When packed into batches, samples in the same batch are padded with the `[pad]` token to have the same sentence length. (b) Training and validation accuracy evolution curves in the ANN (approach 1), the SNN (approach 2), and the SNN with memristor models (approach 2).

please refer to section ‘Methods’. In this experiment, the network is trained for five epochs. For a single training epoch, after training using the entire training set with 17.5k samples, the training is switched off, and 7.5k samples from the validation set are fed into the network to validate the training effect for this epoch. The training and the validation set samples are shuffled before starting another epoch. The trainable network parameters that lead to the smallest validation loss are reloaded to the network for testing after training. 25k samples from an independent test set are then sent to the network. Experiment parameters for the two approaches are summarised in table 1, accuracy evolutions during the training process in figure 3(b), and the final test accuracy in table 2. The baseline test accuracy is 86.02%, given by the ANN in approach 1. According to [89], to achieve higher classification accuracy, more complex architectures are needed such as transformers [91]. We can notice from table 2 that, in approach 1, the test accuracy degradations from the ANN to the SNN and the SNN to the SNN with memristor models are very small (0.13% and 0.01% respectively). In approach 2, the degradation from the ANN to the SNN is 0.1%, whilst the test accuracy drops 1.06% by adding memristor models.

3.2. Results analysis of approach 1

Now we look into the results from approach 1. Figure 4(a) shows how the spike train length for representing a single numerical value affects the classification accuracy. From the curve, we can see that the accuracy shows an ‘increased-and-saturated’ tendency along with the increase of the spike train length because the spike trains are stochastic. The longer the spike trains, the more accurately they can represent the numerical values. However, increasing the length will not change the spike representation accuracy when the spike train is long enough. Therefore, system performance plateaus. Longer spike trains also require longer runtime to iterate through the whole spike train for processing a single sample. Therefore, we choose $T = 1k$ for experiments as the standard configuration in this section, though longer spike trains can result in more accurate representations.

System sensitivity to the $R_{\text{tolerance}}$ when using approach 1 is given in figure 4(b). In theory, the larger the $R_{\text{tolerance}}$, the larger the maximum errors between the expected memristor RSs and the actual values. Therefore, a large $R_{\text{tolerance}}$ value could lead to performance degradation. However, as shown in the bar chart, the system is robust to large $R_{\text{tolerance}}$ values. To investigate the reasons, we plot the Euclidean distance between the expected and the actual weights along with the increase of the read noise (figure 4(c)). Even when the $R_{\text{tolerance}}$ is as large as 20%, the Euclidean distance is not higher than 0.06. This indicates that the system can tolerate variations in exporting weights into memristor arrays when converting a trained ANN to an SNN with memristors.

Similarly, approach 1 also exhibits robustness to the read noise of the read operation for memristor arrays (figure 4(e)). In practice, the instruments used to read memristive RSs introduce read noise, potentially affecting system performance. The read noise in the framework is simulated by adding a random number that does not exceed a certain percentage of the memristor RSs. In other words, when we say the read noise is 10%, it means that the errors caused by the read noise are no more than ±10% of the memristor RSs (figure 4(d)). The read noise is both included in the prediction stage of the weight updating scheme and the
Table 1. Parameters used in the sentiment analysis tasks baseline configuration for approach 1 and 2. Different parameter values for different approaches are highlighted in light grey.

| Parameters                  | Approach 1 | Approach 2 |
|-----------------------------|------------|------------|
| Training set size           | 17.5k      | 17.5k      |
| Validation set size         | 7.5k       | 7.5k       |
| Test set size               | 25k        | 25k        |
| Vocabulary size             | 20,473     | 20,473     |
| Embedding dimension         | 100        | 100        |
| Output dimension            | 1          | 1          |
| Batch size                  | 1          | 1          |
| Offset                      | −25        | −0.5       |
| T                           | 1k         | 1k         |
| $V_{th}$                    | 50         | 56.75      |
| $\eta$                      | 0.05       | 0.05       |
| $\epsilon$                  | $1 \times 10^{-8}$ | $1 \times 10^{-8}$ |
| Array size                  | 10 $\times$ 10 | 10 $\times$ 10 |
| $A_p$                       | 0.21389    | 0.21389    |
| $A_n$                       | −0.81302   | −0.81302   |
| $a_{tp}$                    | 37 087     | 37 087     |
| $a_{tn}$                    | 43 430     | 43 430     |
| $a_{1p}$                    | −20 193    | −20 193    |
| $a_{1n}$                    | 34 333     | 34 333     |
| $t_p$                       | 1.6591     | 1.6591     |
| $t_n$                       | 1.5148     | 1.5148     |
| Positive pulse magnitude (V)| 0.9        | 0.9        |
| Positive pulse duration (us)| 1, 2, 10, 20, 50, 100 | 1, 2, 10, 20, 50 |
| Negative pulse magnitude (V)| −1.2       | −1.2       |
| Negative pulse duration (us)| 1, 2, 10, 20, 100, 1000, 2000, 5000 | 1, 2, 10, 20, 100 |
| $dt$ (ms)                   | 1          | 1          |
| R tolerance                 | 0.05%      | 0.05%      |
| MaxN                        | 5          | 5          |

Table 2. Test accuracy for approach 1 and 2.

|                  | ANN | SNN | SNN with memristors | SNN | SNN with memristors |
|------------------|-----|-----|---------------------|-----|---------------------|
| Approach accuracy (%) | 86.02 | 85.89 | 85.88               | 85.92 | 84.86               |

inference. In this simulation, even with 20% read noise, which means the error between the memristor RSs shown by the instruments and the actual values is up to 20% in the worst case, the system can still achieve a classification accuracy of 81.79%. Figure 4(f) shows the Euclidean distance between the actual and the measured weights with different read noise values. The Euclidean distance caused by 20% read noise is around 0.105 (figure 4(f)).

3.3. Results analysis of approach 2

Now we investigate how the R tolerance affects weight updating when using approach 2. Figure 5(a) plots the expected weight values evolution in the synapse 0 in epoch 1 when training the memristor-based SNN directly (in the blue curve) and the actual weights written in memristors (as orange dots). The curves are plotted every 175 samples. The expected weight (in the blue curve) progressively converges to the optima. However, the actual weights written in memristors stop updating after a certain point because the weight updating is cut off by the R tolerance when the weight change is smaller than the value set by the R tolerance. We envisage that the larger the R tolerance value, the earlier the weight updating stops. [70] has also pointed out the early stop in training a memristor-based SNN when using the R tolerance as the weight updating stop condition. Figures 5(b) and (c) give the training/validation accuracy curves and the final test accuracy with different R tolerance values. The accuracy shows no noticeable change when the R tolerance is smaller than 4.0%. However, when the R tolerance is increased to 5%, the network cannot give reasonable classification predictions due to the early cut-off.

Contrary to the read noise-robustness of the system built using the ANN conversion, directly training the memristor-based SNN is sensitive to the read noise. Figure 5(d) depicts the measured weight evolution in synapse 0 during the training epoch 1 with different read noise values when the R tolerance is 0.5%. Weights are plotted every 175 samples. The ideal situation gives the baseline without the read noise in blue. The
Figure 4. Result analysis for converting an ANN to a memristor-based SNN. (a) Test accuracy vs spike train length, where a train represents a single continuous value. (b) Test accuracy vs the \( R_{\text{tolerance}} \) values when mapping weights to memristor RSs. (c) Weight standard deviations vs \( R_{\text{tolerance}} \) when mapping weights to memristor RSs. (d) The measured weights along with the actual weights with the read noise of 10%. (e) The test accuracy dependency on the read noise. (f) Weight standard deviations vs read noise.

Figure 5. Result analysis for training the memristor-based SNN directly. (a) Weight evolution curve for synapse 0 in the single-layer SNN during training epoch 1. The curve is plotted every 175 samples. The calculated expected weights are plotted in blue, and the actual weights mapped into memristor arrays are in orange. Notably, memristive devices stop updating once the requested updates drop below the \( R_{\text{tolerance}} \). (b) and (c) Training/validation and test accuracy with different \( R_{\text{tolerance}} \) values. (d) and (e) Measured weight evolution for synapse 0 in the training epoch 1 with different read noise values when the \( R_{\text{tolerance}} \) is 0.5% and 3.0%. The curve in blue is the baseline without the read noise. Weights are plotted every 175 samples. (f) Shmoo plot for the test accuracy (%) with different read noise values (\( x \) axis) and different \( R_{\text{tolerance}} \) (\( y \) axis). Positive and negative results are coloured blue and yellow, respectively.

weight in the baseline converges to an optimal state with the value around 0.328, and it converges to around 0.330 with 0.1% read noise. However, the weight evolutions during epoch 1 with the read noise higher than the \( R_{\text{tolerance}} \) (specifically, with 0.5% and 1.0% read noise) fail to converge. We speculate this is because weights cannot be correctly updated when the variations introduced by the read noise exceed the acceptable range defined by the \( R_{\text{tolerance}} \). The effects can be multi-folded: the relatively large measured errors confuse the weight updating scheme to determine the weight change magnitudes or even directions,
introduce weight wobblings, and make the R tolerance unable to be used as an effective stop condition. With 3.0% of R tolerance, which is higher than the largest read noise we have tested, weights in all conditions converge (shown in figure 5(e)). When the R tolerance is higher than the read noise, the major errors during the weight updating are introduced by the R tolerance rather than the read noise so that the impact of the read noise can be ignored. Two features can be noticed in figure 5(e)): firstly, more significant wobblings exist along with the increase of the read noise, with the largest wobbling of ∼±0.005; secondly, the larger the read noise, the farther the measured convergent state apart from the baseline, with the largest distance of 0.03. However, both the fluctuations and the distances seem to be within acceptable ranges. Figure 5(f) gives the test accuracy with different read noise and R tolerance. We conclude that when the R tolerance is higher than the read noise, weights converge, and the test accuracy shows no noticeable difference from the baseline; otherwise, weights fail to converge, and the final test accuracy shows negative results.

4. Discussion and conclusion

In this paper, we developed a simulation framework incorporating an empirical memristor model to demonstrate the first text classification task in memristor-based SNNs with the IMDB movie reviews dataset. Based on a previously presented algorithm-level simulator NeuroPack [70], we focus on the simulation of the algorithm of a memristor-based SNN in performing the text classification tasks only. Other memristor-based SNN simulation frameworks including NeuroSim [92] and MNSIM [93] aim to emulate the circuit-level performance mainly, such as the power consumption. The most similar work to this work is [63], which utilised a PCM model-based SRNN to perform a sequence prediction task.

To explore how memristor non-ideality affects the system performance, we used (1) an empirical memristor model with non-linear switching dynamics to represent the memristor non-linearity, (2) the R tolerance to represent the write errors during the weight updating process, and (3) the read noise to represent the noise disturbance during the read operation. Naturally, there are other non-ideal effects such as stuck-at faults [94] and the line resistance [95] that arise when utilising memristor-based SNNs. We did not study these factors in this work to keep the focus on algorithm-level simulation including 'non-ideal but functioning devices’ only. However, we are working towards making this work involve more circuit-level and device-level factors in the next stage to provide more precise simulation results. Broadly, we expect line resistance to introduce distortions in our ability to program devices by affecting voltage delivery to the device being programmed (potentially fixable by transferring to current-mode driving or using adaptive voltage control and write-verify schemes). We would also expect stuck-at faults to add extra degradation to accuracy, similar to what happens with software neural network models with such faults injected into them [96], although it is difficult to predict exactly whether this will catastrophically interfere with the inaccuracies arising from e.g. the non-zero R tolerance.

We took two paths to obtain trained memristor-based SNNs by (1) converting a pre-trained ANN to a memristor-based SNN or (2) training a memristor-based SNN directly using the equivalent ANN-based learning rules. We elaborated on the specific details of estimating the critical parameters to guide users to achieve comparable performance when taking the presented methods. We achieved the test accuracy of 85.88% and 84.86% from the two approaches with only 0.14% and 1.16% degradations, respectively, given the baseline accuracy of 86.02% from the equivalent ANN. Finally, we investigated how the SNN stochasticity and memristor non-idealities affect the system performance. We concluded that it is possible to achieve very similar performance in simulation from ANNs to memristor-based SNNs and from non-memristive synapses to data-driven virtual memristive synapses. Proven by demonstrating a sentiment analysis task, using the simulation framework presented in this paper can accelerate research in employing memristor-based SNNs in text classification tasks by overcoming the lack of learning rules and memristor non-idealities.

Data availability statement

The data that support the findings of this study are openly available at the following URL/DOI: https://github.com/hjq310/text-classification-in-memristorsnn.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.
Author contributions

JH developed the simulation framework, conducted the simulation, and wrote the manuscript. AS reviewed and polished the manuscript. SS extracted device parameters for the simulation. TP provided high-level advice for writing improvement.

ORCID iDs

Jinqi Huang https://orcid.org/0000-0003-3913-0067
Alexandrou Serb https://orcid.org/0000-0002-8034-2398
Spyros Statapopoulos https://orcid.org/0000-0002-0833-6209
Themis Prodromakis https://orcid.org/0000-0002-6267-6909

References

[1] Sengupta A, Ye Y, Wang R, Liu C and Roy K 2019 Going deeper in spiking neural networks: VGG and residual architectures Front. Neurosci. 13 95
[2] Maass W 1997 Networks of spiking neurons: the third generation of neural network models Neural Netw. 10 1659–71
[3] Long I and Fang G 2010 A review of biologically plausible neuron models for spiking neural networks AIAA Infotech Aerosp. 3540 2010
[4] Duwek H C and Tsur E E 2021 Biologically plausible spiking neural networks for perceptual filling-in in Proc. Annual Meeting of the Cognitive Science Society vol 43 (available at: https://escholarship.org/uc/item/3gm896kr)
[5] Peter Tang P T, Lin T-H and Davies M 2017 Sparse coding by spiking neural networks: convergence theory and computational results (arXiv:1705.05475)
[6] Ahmed K, Shrestha A, Qiu Q and Wu Q 2016 Probabilistic inference using stochastic spiking neural networks on a neurosynaptic processor 2016 Int. Joint Conf. on Neural Networks (IJCNN) pp 4286–93
[7] Khodamoradi A, Denolf K and Kastner R 2021 S2n2: a FPGA accelerator for streaming spiking neural networks The 2021 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays pp 194–205
[8] Cheng L, Liu Y, Hou Z-G, Tan M, Du D and Fei M 2021 A rapid spiking neural network approach with an application on hand gesture recognition IEEE Trans. Cogn. Dev. Sys. 13 151–61
[9] Kim S, Park S, Na B and Yoon S 2020 Spiking-yolo: spiking neural network for energy-efficient object detection AAAI (https://doi.org/10.1609/aaai.v34i07.6787)
[10] Lee W W, Kukreja S L and Thakor N V 2017 Cone: convex-optimized-synaptic efficacies for temporally precise spike mapping IEEE Trans. Neural Netw. Learn. Sys. 28 849–61
[11] Reid D, Hussain A J and Tawfik H 2014 Financial time series prediction using spiking neural networks 2014 IEEE/ACM Int. Symp. on Nanoscale Architectures (NANOARCH) pp 1–8
[12] Payvand M, Rofeh J, Sodhi A and Theogarajan L 2014 A CMOS-memristive self-learning neural network for pattern classification applications 2014 IEEE/ACM Int. Symp. on Nanoscale Architectures (NANOARCH) pp 92–97
