THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Optimized Machine Learning via Brain-Inspired Neuromorphic Algorithms

ANDRE MITRIK
SPRING 2023

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Engineering
with honors in Computer Engineering

Reviewed and approved* by the following:

Abhronil Sengupta
Assistant Professor in Electrical Engineering and Computer Science
Thesis Supervisor

John Morgan Sampson
Assistant Professor of Computer Science and Engineering
Honors Adviser

* Electronic approvals are on file.

ABSTRACT

In this work, we adopt a mindset of extending machine learning (ML) ingenuity by focusing on an alternative to a current driver of reinforced intelligence – typically labeled as artificial neural networks (ANN) – in the form of a neuromorphic brain-inspired architecture: a spiking neural network (SNN). The ever-growing field of ML offers a wide array of extremely impressive computer-driven automated operations, like image recognition, pattern analysis and data recognition service; these expenses do not come without ramifications of resource-intensive requirements and scalable computational calculations. Research of current SNNs imply that these extensive operations can be performed in similar fashions to the functionality of the cerebral cortex of the brain, ideally mimicking the efficiency of human thought-processing while reducing the latency for computations of extreme magnitudes. This thesis aims to build upon that trend of reducing latency to maximize network training efficiency.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would first like to give my thanks to Professor Abhronil Sengupta, the head of the Neuromorphic Computing Lab at The Pennsylvania State University. I am extremely fortunate and grateful to have been included in the operations of his lab and could not have continued to pique my intellectual curiosities and facilitate this research without his guidance. He has given me invaluable advice that helped me complete this thesis and will continue to help me grow as a person, and I'm proud to have been able to work under his counsel.

I would also like to thank Sen Lu, a graduate research assistant within the NCL and a primary contributor to the programming-related advancements conducted for the lab's research. His guidance on the coding side always helped me understand what research and exploration of the abstract was like, and I sincerely appreciate the time he dedicated for me to help attempt extend his and Professor Sengupta's work. I also could not have completed this thesis without his help.

Chapter 1 – Introduction

1.1 – Motivation & Rationale

In the ever-evolving field of machine learning (ML) research, the advances and progress for ML-integration into common practices becomes more evident every single day. Intrinsic to its name, “learning machines” continue to show development in their capabilities to emulate human behavior through metrics of both efficiency and effectiveness. Consumers likely interact with or witness an extension of ML-based technology daily; for instance, phones and computers take advantage of image and speech recognition, popular apps and e-commerce titans frequently utilize user-based data and tendencies to personalize their platforms for their users, and ideas as broad and abstract the commonality of self-driving cars begin to consolidate their prominence within the Internet of Things (IoT). To avoid an ingenuity plateau, however, the future demand for more resource-intensive computations and powerful systems facilitating this research grows proportionally with the pursuit of excellence.

Chapter 2 – Background & Literature Review

2.1 – Machine Learning

To understand the potential of innovation provided by machine learning, the scope of its goal and ideals necessitates concise non-technical definition. ML refers not to programs, but instead to algorithms that iterate through datapoints and make parametrized adjustments based on large datasets to satisfy prospects of formulating intelligent predictions fitted to a system. Recent applications of ML progress imply that these algorithmic techniques frequently exhibit human levels of semantic interpretation, information extraction, and pattern recognition – sometimes to an even greater accuracy than what humans are capable of [5]. Computer competence continues to expand its range of fields where applicable ML-algorithms exist. Although initially popularized for baseline experiments like image recognition, growth of underlying architecture and refined development of large datasets implies possibilities of computer-facilitated processes in other previously non-related fields like vehicle driving, adaptive chatbots, and real-time language translations. Thus, the importance of both efficiency *and* effectiveness of these algorithms becomes paramount for practical societal integration.

2.2 – Artificial Neural Networks

Elucidating technical context behind the current state of machine learning research is required prior to the proposition of this work's alternative methodology; in this case, we first define groundwork referencing Artificial (or Analog) Neural Networks (ANN), also known as feed-forward neural networks. Since an extensive amount of the innovative work for this paper

revolves around Spiking Neural Networks (SNN), this section reviews general functionality and operational explanations for common ANN architectures. This provides a strong knowledge base to better understand differences between ANN and SNN architectures.

2.2.1 – Neurons & Perceptrons

The basic operational primitives for various forms of neural networks are neurons, where a general model mimics the functionality of the neuron within the brain. Neurons transmit electrical signals between each other; in essence, they contain the capability to receive certain signals and generate other signals. Framing this within the scheme of computers, neurons can receive input data, process the data accordingly based on parameters and goals within a network model, and then yield an output. A popular alternate (and more commonly referenced) convention to neurons is the perceptron, which is essentially just an artificial neuron modeled to mimic the brain.

2.2.2 – Activation Functions

The mathematical derivations which make the perceptrons operate according to the goal of the system is where the magic happens. Within ANN architecture, outputs are not constantly generated; instead, an output signal generates when a certain threshold value is reached [14]. This threshold is explicitly defined by the network's choice of activation function, which is the generator of the output signal. Many theoretical practices utilize a step function (see Figure 1), which is an explicit jump in activation, but many larger datasets tend to err towards the

practicality of Rectified Linear Unit (ReLU, see Figure 2) or Sigmoid functions (see Figure 3) based on the network's desire for computational speed, biological realism, or other granular parameters [6]. In general, these activation functions determine the sensitivity to sending an output spike for a given perceptron within a network's architecture.

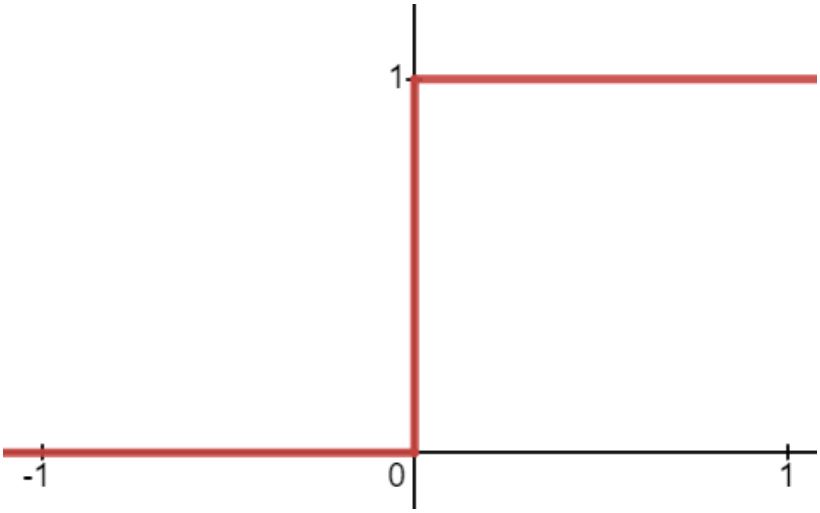

Figure 1: Step Function

$$f(x) = \begin{cases} 1, x \geq 0 \\ 0, otherwise \end{cases}$$

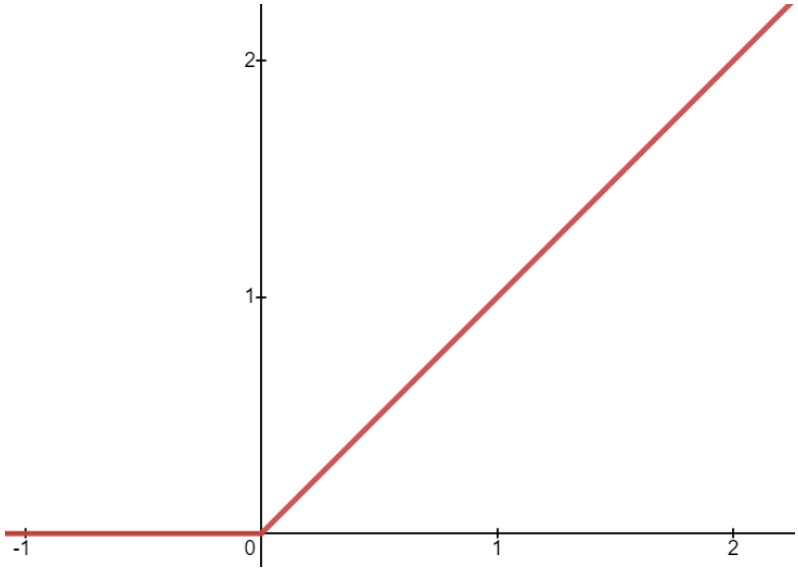

Figure 2: ReLU Function

$$f(x) = \max(0, x)$$

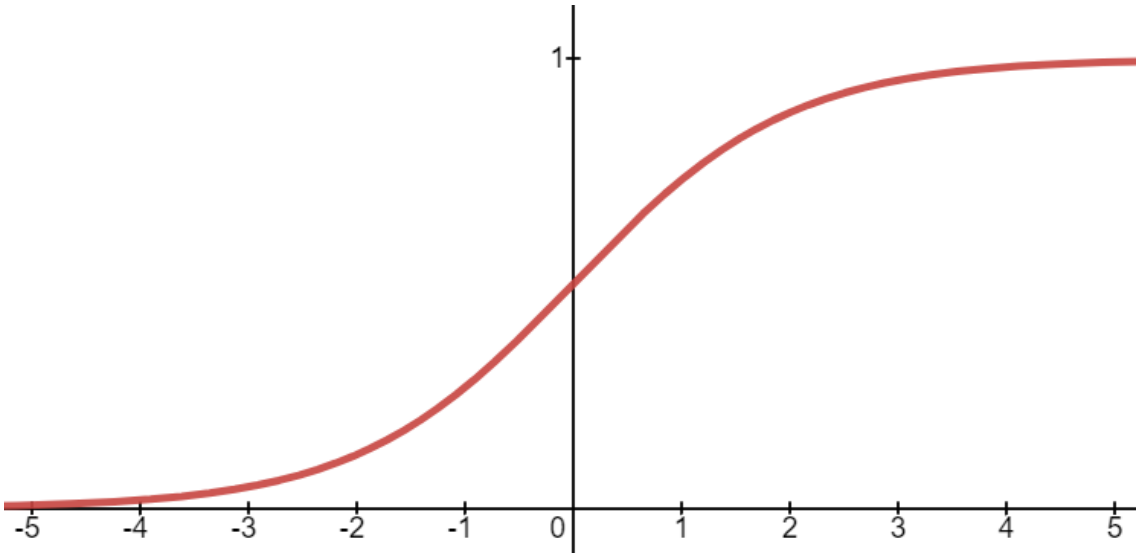

Figure 3: Sigmoid Function

$$f(x) = \frac{1}{1 + e^{-x}}$$

2.2.3 – Network Weight Architecture

The functionality of the granular-level neural network formulates around the single-unit perceptron, but to create a functional system there can be hundreds or thousands of perceptrons connected through an entire network. The idea of effective connectivity is an area with constant room for improvement, where neuroscience widely tries to adapt these networks to current understandings of interactions between brain regions and the influence that neural structures tend to exert on one another [9].

Figure 4 below shows an illustration of the interconnections between perceptrons (nodes) in a neural network. There are input and output layers, as well as an undefined amount of "hidden" layers between input and output. The goal-oriented variables are the number of nodes per layer and the amount of hidden layers dependent on the requirements of the network; as expected, the more complex the problem, the more nodes/layers required. Typically, each of the nodes will be connected to other nodes, where their interconnection is associated with a numerical "weight." The weight directly influences the likelihood of the previously discussed chosen activation function from yielding an output dependent on the current sequence of inputs.

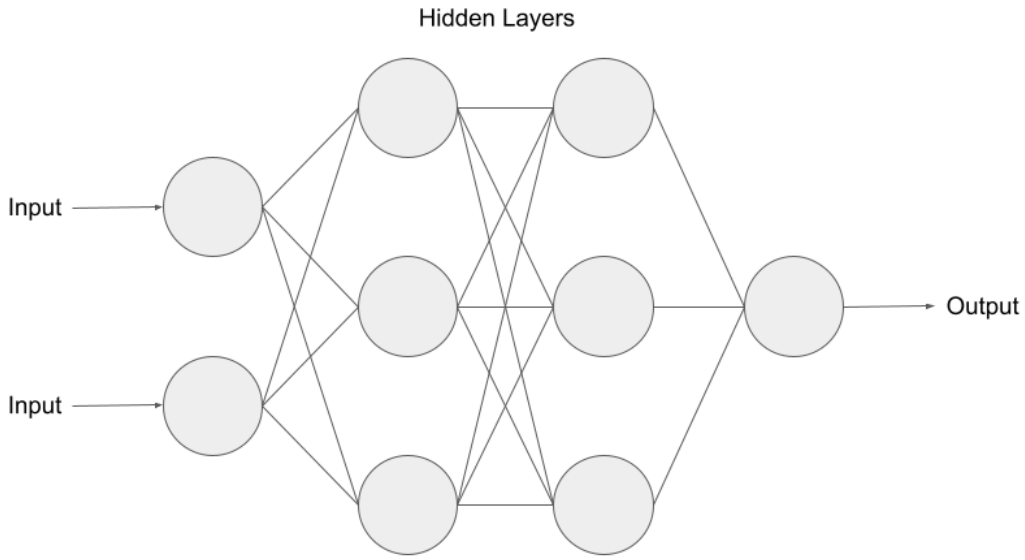

Figure 4: Sample Neural Network Model

The process of training neural networks given datasets includes two major steps that take on most of the intensive calculations for the network: forward propagation of data signals and backward propagation of resultant errors.

2.2.4 – Forward Propagation

During the forward propagation phase, also known as the training phase for a network, training data samples are provided at the input layer; these data samples are predefined with data classifications so that the network can check itself later with the correct classification based on the input. Figure 5 illustrates a visual with the same basic neural network in Figure 4 depicting how sample inputs are fed through the network to calculate activation values for nodes in each

layer, all the way until the outputs are calculated; at this point, the predictions for the applied patterns may occur within the backward propagation phase so that the network "trains" itself.

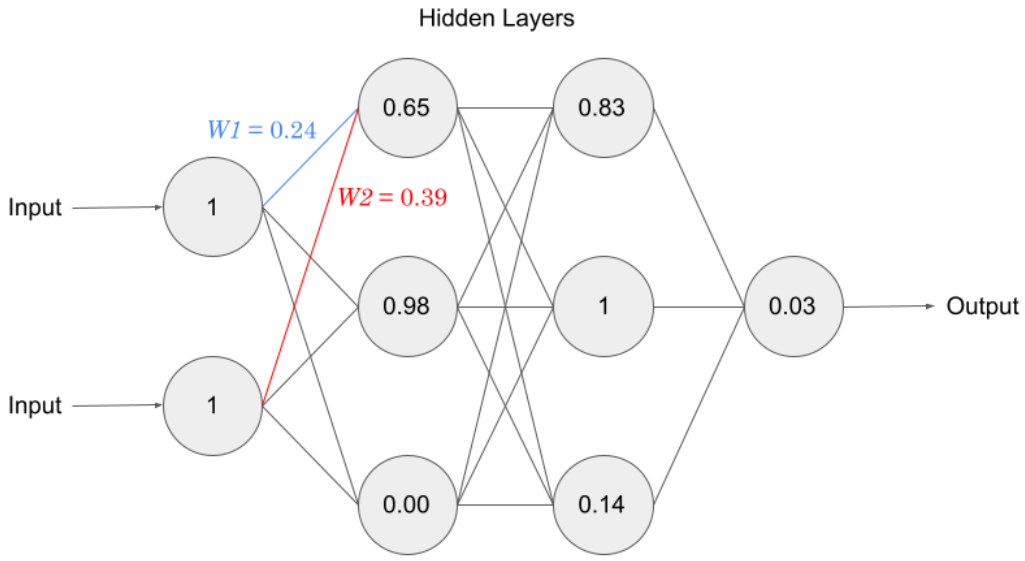

Figure 5: Sample Neural Network Model After Forward Pass

The above example utilizes a Sigmoid activation function. Each activation is calculated by computing the dot product for all associate input activation values and weights connected to those inputs; the example is color coated to display this. For the top node in the first hidden layer, the dot product x is computed as:

$$x = W1 * INPUT1 + W2 * INPUT2 = (0.24)(1) + (0.39)(1) = 0.63$$

After computing the dot product, we can reference Figure 3 and utilize x in the Sigmoid output calculation:

$$Activation = \frac{1}{1 + e^{-x}} = \frac{1}{1 + e^{-0.63}} = 0.65$$

This process remains the same for each node during the forward pass and is just one example for one node and one activation function.

2.2.5 – Backward Propagation

As aforementioned, each data sample given as input comes associated with a desired output; intrinsically, if this were not true, the network would be pointless as there would be no data for it to "reinforce" what is and is not correct. Given this information, at the end of the forward propagation phase, if there exists some output perceptron that does not match its desired output value, an error gradient value is calculated and backpropagated through the hidden layers back to the input layers where necessary adjustments to the weights of the interconnective synapses are made [1]. Being a more calculation intensive step, the backpropagation (as its name suggests) works backwards in calculating the necessary change (delta) for each node. To update the weight of a particular synapse, utilizing the learning rate α of the network (see Table 2 for supplementary information), the current activation value for the input node $I(t)$ to the synapse, and the calculated delta, the weight change calculates as:

$$\Delta w_{node} = \alpha * I(t) * \delta_{node}$$

This repeats for every synapse weight during each forward/backward pass. The process of forward and backward propagation is the core of how neural networks satisfy criteria to be "machine learning" intelligent systems and is how they adapt to given inputs and expected outputs to become trained.

2.3 – Spiking Neural Networks

As machine learning methods utilize computational models that exhibit framework and structure inspired by neural networks within the brain, models naturally differ in their manipulation of input data and the methodology of the following computations based on these inputs – such is the case with SNNs. ANNs, although proven to exhibit effective training with datasets of varying size, face growing issues with the scaling problem. In a machine learning environment where algorithmic advances yield more powerful results at the cost of harsher storage and computational requirements, there exists a question as to how growing datasets and machine learning techniques can match ambition. SNNs exhibit the potential to enable low-power event-driven neuromorphic hardware; in response, there are alterations that differ from the standard forward and backward propagation steps in an ANN [3].

2.3.1 – ANN vs. SNN Comparison

SNN architecture bases itself on the fact that biological neurons in the brain processes binary spike-based information and represent information through these spikes/discrete events. As opposed to the ANN input being static strings of data with continuously updated activation values through forward and backward propagation, SNNs perform these computations by noting the precise timing of spikes as a binary event [8]. Theoretically, since these spikes signify the only time when the system needs to dedicate power for corresponding computations, SNNs classify as event-driven computing systems with supervised learning rules that instead aim to adjust the timing of spikes between perceptrons [3].

2.3.2 – Spike Train Inputs

The alternative to the propagation and update phases within an ANN is defined by the learning rule associated with the SNN under question; additionally, the sequence in which the SNN responds to input varies. Both models, however, share a very similar backbone in their forward and backward phases. Again, inputs are no longer analog for SNNs, but can be seen as sequential "trains of input." Input streams are first sent into the input layer, and are propagated through layers on a timestep basis; these perceptrons are commonly defined as excitatory (releasing positive spikes) or inhibitory (releasing negative spikes) [4]. Each perceptron integrates the input train over time, and carries a value referred to as membrane potential. Certain events within an input train cause the membrane potential to increase or decrease, and when it reaches a certain defined threshold, there is a "spike."

2.3.3 – Integrate-Fire Spiking Neuron

Due to the existence of timestep-based input, the representation of the activation values requires an adjustment from a discrete value to a time-dependent value. As exemplified in the first figure of Ref. [8], there is a considerable similarity in characteristics to that of a ReLU referenced in Figure 2 – linearity. We define the equation below:

$$v_{mem}(t+1) = v_{mem}(t) + \sum_i w_i \cdot X_i(t)$$

In this case, there is a state variable for membrane threshold v_{mem} which accumulates i incoming spikes from input spike train X until reaching some threshold v_{th}, where an output spike then fires [12].

2.3.4 – ANN-SNN Conversion

In the scheme of popularized practice for SNN training, there are three primary steps – the first of which is direct ANN-SNN conversion. Per Ref. [12], this methodology still relies on the standard techniques of backpropagation, allowing SNNs to be trained for larger-scale problems that typical ANNs would handle. This observation is directly based on the functional equivalence of an Integrate-Fire spiking neuron to a ReLU, as shown by shared properties of linearity defined in subsections 2.2.2 and 2.3.3. Under these assumptions, transitivity applies between models leading to the possibility of ANNs trained with ReLU neurons being transformed into SNNs with IF spiking neurons without marginal losses in accuracy; this concept is explicated in further detail in [8]. Completion of this conversion leads to the following two steps of SNN conversion: backpropagation through time (BPTT) and network updates through Spike-Timing Dependent Plasticity (STDP). Note that backpropagation is now specified with parameters "through time" due to the event-driven nature of SNNs, and the necessity for timestamps associated with spikes.

2.3.5 – Updating the Network

Following the theme of modeling the neural cortex within a brain, the "weights" from the ANN model are commonly referred to as "synapses," simply a junction connecting two neurons – a pre-neuron, or the transmitting neuron, and the post-neuron, the receiving neuron. The methodology behind the synaptic weight updates depends on the learning rule for the model – a popular example for SNNs, as previously mentioned, is Spike Timing-Dependent Plasticity (STDP). Within this process, after a spike occurs, it gets propagated backwards to previously

connected perceptrons. The direct timing of the spikes serve the same purpose as the error gradients do for an ANN. In general, this theory states that the weight of the synapse increases (decrease) if the pre-neuron spikes before (after) the post-neuron; Ref. [2] by Sengupta, et. al includes figures with detailed illustration of these trends.

Chapter 3 – Innovative Work

The innovative portions described further in this thesis builds off the hybrid SNN training strategy described in Ref. [7]. The label “hybrid” implies this methodology facilitates an approach taking advantage of two major SNN concepts discussed in subsections 2.3.4 and 2.3.5: ANN-SNN conversion for training rate-coded deep SNNs and error-backpropagation through time.

3.1 – Codebase

This work directly uses code based on the strategy and work in Ref [8]. The goal of this thesis orients around running the BPTT algorithm on SNNs and analyzing the section of code where the layer-wise training occurs; as previously mentioned in subsection 2.2.3, many neural network architectures are built with multiple layers that each serve their own individual purpose. During the analysis of the layer-wise training, this implementation capitalized on the functionality of principal component analysis (see subsection 2.4.2) to quantifiably determine the important layers of the network; in turn, this would open the opportunity for the BPTT algorithm to selectively unroll the more important layers more times than the less import or less calculation-intensive layers. Subsection 2.4.3 explicates the specifics of the timestep alteration algorithm implemented for this work. In theory, this approach would reduce the total timesteps per epoch devoted to training a network, effectively decreasing latency while optimistically avoiding accuracy degradation during the training process. Appendix A contains the modified BPTT algorithm used in this thesis.

3.1.1 – Model

The SNN in this project trains on a VGG5 model. VGG-based models are popular architectures consisting of layers for convolution calculations and pooling, with a subsequent fully connected classification layer. The number following VGG typically indicates the number of associated layers for the architecture type. The simple and uniform design make it optimal for computer vision benchmark tests. It should be noted that this VGG5 model is smaller than the typical VGG16 or VGG19 architecture, but the assumption made for this project was that the number of layers and other features of this model still exhibited potential for yielding results that could scale in similar efficiency trends for larger problems, models, and datasets.

3.1.2 – Dataset

To complement this smaller VGG model, the CIFAR-10 dataset is another popular dataset used for image classification; the more compact images within the dataset and the 10 classes illustrated in Table 1 made it a good scalable counterpart for the model architecture.

	Class
1	Airplane
2	Automobile
3	Bird
4	Cat
5	Deer
6	Dog

7	Frog
8	Horse
9	Ship
10	Truck

Table 1: CIFAR-10 Classes

3.2 – Principal Component Analysis and Timestep Alterations

The work explicated in [13] by Lu and Sengupta (2022) powers the rationale behind the application of principal component analysis (PCA) on SNNs. To understand the importance of layer wise neuron threshold optimization within a SNN, their work hypothesized the correlation between layer importance and overall prediction capability. Specifically, it was determined that a layer's significance was proportional to the **percentage increase** in the number of principal components in comparison to the PCs of the previous layer – the code and timestep alteration algorithm for this thesis, due to the smaller model, bases decision not directly on the percentage increase of principal components between layers but instead the PC ratio relations between layers per iteration (see section 3.2.2).

3.2.1 – Applied PCA Code

At an abstract level, PCA maps uncorrelated input data point variables and computes a basis vector set maximizing the variance of the data. By calculating the principal component of each individual layer's **feature map**, the projective success potential of each individual layer can be quantified. Figure 6 below depicts the algorithm applied in this work.

```
def analyze_feature(input_tensor, ratio=0.999):
    pca = PCA(n_components=50)

    # Acquiring number of filters to the first dimension
    input_tensor = np.swapaxes(input_tensor,0,1)

    # Reshaping input to 2D array (# filters * rest flattened)
    input_tensor = input_tensor.reshape([input_tensor.shape[0], -1])

    # Dimensionality reduction on the input tensor
    pca.fit_transform(input_tensor)

    # Acquiring explained variance to represent variation in our dataset attributed to each principal component
    explained = pca.explained_variance_ratio_.cumsum()

    return (explained < ratio).sum()
```

Figure 6: PCA Algorithm

Explicating the above code, $analyze\_feature()$ takes in two parameters: an input tensor housing data in a specified number of dimensions (dependent on the layer) and a ratio representing the variance percentage R in the feature map for the layer under question. Citing the original principal component analysis in Ref. [13], the assumption that the general trend of principal components shows certain parallels with layer wise optimized neuronal thresholds inspired the following experiments to conduct calls on this function **before** the non-linear activations to display the extent of layer wise redundancy in response to tensor dimensionality increases.

3.2.2 – Creating Altered Timestep Ratios – Linear Approach

Reiterating the purpose of this thesis, after obtaining the principal components from each layer per timestep, the next step included a proposed alteration to the number of unrolling calculations executed per layer. An example output from $analyze\_feature$ may look like the following:

$$analyze\_feature = [4, 35, 9, 50, 19, 50]$$

According to this representation, each indexed principal component corresponds to the layer at the same index in the following synonymous array:

$$[Conv2d, AvgPool2d, Conv2d, Dropout, Conv2d, AvgPool2d]$$

It is important to note that this work primarily focuses on timestep alteration of the calculation-intensive **convolution** layers. Figure 7 shows the code for the timestep ratio calculation:

```
def calculate_timestep_ratios(principal_components, timesteps):
    # Size of layer_list should be 9 for the 9 different layers in the architecture
    adjusted_timesteps = []
    # Highest principal component value in principal_components
    divisor = max(principal_components)

    # Loop calculating the adjusted lesser timesteps based on the max_pc and timesteps input
    for i in principal_components:
        adjusted_timesteps.append((i / divisor) * timesteps)

    # Typecasting to ints
    adjusted_timesteps = [int(a) for a in adjusted_timesteps]
    return adjusted_timesteps
```

Figure 7: Linear Timestep Ratio Calculation Algorithm

After declaring a new array meant to hold the adjusted timesteps, the highest-valued principal component is extracted from the input parameter $principal\_components$ to be used as a divisor for the remaining timestep calculations. There is an additional input to this function, $timesteps$, which holds the model's hyperparameter for the original amount of timesteps that **every** layer would typically unroll before the modifications suggested by this project. With these variables in mind, by iterating through the principal components, each new timestep calculation simply follows the below formula:

$$adjusted\_timestep[i] = \frac{PC}{divisor} * timesteps$$

3.2.3 – Creating Altered Timestep Ratios – Non-Linear Approach

A supplementary experimental approach for discussion in Chapter 4 is a non-linear algorithmic design for the new adjusted timesteps; in this case, the function experiment with is the Sigmoid function. Re-referencing subsection 2.2.2, the sigmoid function was believed to offer potential advantages due to its tendencies to merge well with models that rely on probability as an output, and an output ranging between 0 and 1. For this application, the new implementation of $calculate\_timestep\_ratios$ shares a similar backbone shown in Figure 8.

```
def sigmoid(summed_input, scaling_factor):
    return 1/(1+np.exp(-summed_input/scaling_factor))

def calculate_timestep_ratios(principal_components, timesteps):
    # Size of layer_list should be 9 for the 9 different layers in the architecture
    adjusted_timesteps = []
    # Highest principal component value in principal_components
    divisor = max(principal_components)

    # Loop calculating the adjusted lesser timesteps based on the max_pc and timesteps input
    for i in principal_components:
        adjusted_timesteps.append((i / divisor) * timesteps)

    # Typecasting to ints
    adjusted_timesteps = [int(a) for a in adjusted_timesteps]
    return adjusted_timesteps
```

Figure 8: Non-Linear Timestep Ratio Calculation Algorithm

As stated before, the only difference in application from the approach described in subsection 3.2.2 is that the original input parameter $principal\_components$ is elementwise ran through the sigmoid function, and thereafter the same divisor-based calculation is made to return

the adjusted timesteps. The sigmoid function in our case utilized a scaling factor for its output which facilitated a wider variety of results and applicable experiments discussed in Chapter 4.

3.3 – Application During BPTT

To describe the integration of code formulated in section 3.2 within the SNN training environment, the pseudocode in Figure 9 enumerates the basic nested functionality of the scope of the BPTT process and where the work in this thesis made alterations to the existing codebase. To translate the pseudocode and consider the application in the setting of the actual changes, one can reference the file inside the repository linked with Ref. [8], $vgg\_spiking.py$. To see the entire algorithm connected to this pseudocode, please reference Appendix A.

```
def BPTT(self, network):
    1. Declare empty array layer_list that will hold references to each layer in the architecture
    2. If this is not the first timestep iteration, load PCs from previous iteration (torch.load)

    IF (FIRST TIMESTEP)
        HARDCODE adjusted_timesteps
    ELSE
        LOAD PREV. LAYER PRINCIPAL COMPONENTS
        SET adjusted_timesteps = calculate_timestep_ratios() or calculate_timestep_ratios_sigmoid()

    3. Now that we have the maximum "allowable" timesteps per layer, begin looping through timesteps and layers

    FOR (timestep) IN (self.timesteps):
        Get pointer to first layer in network

        FOR (layer) IN (feature_layers)
            ADD layer TO layer_list
            IF (adjusted_timesteps[layer] > timestep)
                DO NOT UNROLL (continue)

    ##### UNRELATED CODE TO implementation... #####

    4. Declare empty array principal_components and fill it with analyze_feature
    FOR (layer) in (layer_list)
        principal_components.append(analyze_feature(layer, variance))

    5. Save PCs for next iteration (torch.save)
```

Figure 9: BPTT Modification Pseudocode

As per the heuristic chosen, each timestep ratio is based on the principal component assessment for the previous iteration. Capabilities through the PyTorch deep learning library employs the possibility of saving an object to the disk file through $torch.save()$ and subsequently loading the previous iteration results from the disk through $torch.load()$. This allows for efficient implementation of the current required PC analysis ratios for the event-driven nature of SNNs.

The array $adjusted\_timesteps$ serves as an overhead to the process of looping; the calculated adjusted timesteps array is structured such that it index-matches the same order of layers in $layer\_list$. Thus, memory allocation remains efficient, as assessing whether or not a layer i in the layer list needs to be unrolled can be checked by comparing the value at

$adjusted\_timesteps[i]$ to the current iteration, $timestep$, and taking necessary action – either unrolling the layer or trying to improve latency by continuing through the loop and not wasting time on potentially insignificant operations.

Chapter 4 – Testing & Procedures

4.1 – Prerequisites

Before delving through the application of the aforementioned functions, an explanation of the testing environment and arguments into the SNN training scheme are necessary to explain the constants and variables for contextual background.

4.1.1 – Environment

As expected per the nature of neural network training and their computationally expensive processes, two primary external environments were utilized for the purposes of implementing the innovative work described in Chapter 3. The primary tests were conducted on an Oracle Compute Cloud, with supplementary testing taking place on two separate servers associated with the Neuromorphic Computing Lab at Penn State; one server utilizing an NVIDIA RTX 2080Ti (11 GB of RAM), and another utilizing an NVIDIA RTX A5000 (24 GB of RAM). These graphics processing units allow for scalable testing that would have otherwise been extremely strenuous (and impossible after a certain point) on a local testing machine.

4.2.2 – Test Arguments

Table 2 illustrates the environment arguments utilized uniformly for each test.

Argument	Value	Description
Dataset	CIFAR-10	An object recognition computer vision dataset. This subset contains 60,000 32x32 colored images containing an object classified under one of the ten classes in Table 1 in subsection 3.1.2.
Batch Size	64	Total number of samples processed per training iteration; for this example, e.g., within the 60,000 total images in the CIFAR-10 dataset, 64 images get processed per forward/backward pass.
Epochs	30	Total number of passes through the dataset; each epoch is complete when the model has processed every sample in the dataset
Architecture	VGG5	The descriptor for the network's framework; as previously mentioned in subsection 3.1.1, the VGG5 model is a smaller VGG model with less intensive depth and smaller convolutional/connected layer count
Learning Rate	0.0001	The scale size for how dramatic the adjustments are in response to the gradients calculated during the BPTT algorithm
Scaling Factor	0.7	The preprocessing factor by which a feature layer becomes normalized to ensure that the range of values are consistent
Weight Decay	0.0005	A "penalty term" to the loss function in the training process that serves as a regularization method to prevent overfitting by promoting smaller weights and "decaying" the adjustments after BPTT
Momentum	0.95	A factor that helps the convergence of the optimization processes by pushing the gradient updates from the loss function in the direction of the previous gradient; this effectively keeps the learning process from continually bouncing back and forth to accelerate convergence
Dropout	0.3	The factor referencing a technique to prevent machine learning overfitting by taking perceptrons at the given rate within the entire schema and "dropping" them out during the training phase (setting them to zero). This enforces more robust feature learning processes.

Table 2: Experimental Hyperparameters

4.2 – Model Output

The benchmarks used (which this thesis attempts to improve upon) for comparison were evaluated and averaged with the hyperparameters specified in Table 2 using a pretrained ANN provided by the codebase referenced in [8] and the pretrained model associated with the VGG5 CIFAR-10 architecture. To condense a 30-epoch set of data into readable results, every fifth epoch result comprises Table 3 – this still accurately shows the trend of the data through the entire experiment (and this theory held true for the experiments to follow).

There are two overhead metrics surrounding the data in Table 3; training and testing are treated as different entities. However, the loss (disparity between output and expected) and the accuracy (validating correct categorization within the dataset) for both categories remain important to see trends and training status of the entire network and different instances in time.

Epoch	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Peak Accuracy	Time
1	1.7345	0.8614	0.6425	0.8604	0.8604	00:10:26
5	0.3778	0.8840	0.4372	0.8722	0.8722	00:10:24
10	0.3154	0.8971	0.4059	0.8767	0.8767	00:10:27
15	0.2944	0.9026	0.3923	0.8786	0.8792	00:10:25
20	0.3034	0.8983	0.2890	0.9767	0.8870	00:10:29
25	0.2934	0.9008	0.3766	0.8878	0.8894	00:10:29
30	0.2845	0.9020	0.3750	0.8899	0.8901	00:10:26

Table 3: Comparison Output Pre-Experiment

4.3 – Linear Timestep Adjustment Experiments

As discussed in Chapter 3, there were two primary experimental subsets that yielded notable results: PCA analysis with both linear and non-linear timestep adjustment. For linear adjustment, three experiments were conducted with variation being made by altering the **explained variance** within the principal component analysis. The explained variance directly affects the principal component count per layer, as will be shown in future tables, thus also yielding different theoretical timesteps. The context variables for each experiment are covered in this chapter, and all the associated results and analysis occur in Chapter 5.

4.3.1 – Linear Experiment 1: Variance = 0.8

Principal Component Explained Variance: 0.8		
Layer Type	**Principal Components**	**Theoretical Adjusted Timesteps**
Convolutional 2D	4	8
Average Pooling 2D	35	70
Convolutional 2D	9	18
Dropout	50	100
Convolutional 3D	19	38
Average Pooling 2D	50	100

Table 4: PCs and Timesteps for Linear Experiment 1: Variance = 0.8

4.3.2 – Linear Experiment 2: Variance = 0.99

Principal Component Explained Variance: 0.99		
Layer Type	**Principal Components**	**Theoretical Adjusted Timesteps**
Convolutional 2D	9	18
Average Pooling 2D	50	100
Convolutional 2D	39	78
Dropout	50	100
Convolutional 3D	50	100
Average Pooling 2D	50	100

Table 5: PCs and Timesteps for Linear Experiment 2: Variance = 0.99

4.3.3 – Linear Experiment 3: Variance = 0.9

Principal Component Explained Variance: 0.9		
Layer Type	**Principal Components**	**Theoretical Adjusted Timesteps**
Convolutional 2D	6	12
Average Pooling 2D	47	94
Convolutional 2D	15	30
Dropout	50	100
Convolutional 3D	32	64
Average Pooling 2D	50	100

Table 6: PCs and Timesteps for Linear Experiment 3: Variance = 0.9

4.4 – Non-Linear Timestep Adjustment Experiments

The subset of experiments within the non-linear Sigmoid application grouping does not focus on altering the variance, but instead the aforementioned **scaling factor** chosen used for the Sigmoid function output. Through binary search logic and trial and error, it was generalized that for scaling factor k, an optimal range for yielding non-negligible timestep differences fell in

range $25 < k < 50$; the two extremes in this range were utilized for variance values 0.90 and 0.95.

4.4.1 – Non-Linear Experiment 1: Variance = 0.9, k = 25

Principal Component Explained Variance: 0.9, Scaling Factor: 25		
Layer Type	**Principal Components**	**Theoretical Adjusted Timesteps**
Convolutional 2D	6	63
Average Pooling 2D	47	98
Convolutional 2D	15	73
Dropout	50	100
Convolutional 3D	32	88
Average Pooling 2D	50	100

Table 7: PCs and Timesteps for Non-Linear Experiment 1: Variance = 0.9, k = 25

4.4.2 – Non-Linear Experiment 2: Variance = 0.9, k = 50

Principal Component Explained Variance: 0.9, Scaling Factor: 50		
Layer Type	**Principal Components**	**Theoretical Adjusted Timesteps**
Convolutional 2D	6	72
Average Pooling 2D	47	99
Convolutional 2D	15	78
Dropout	50	100
Convolutional 3D	32	89
Average Pooling 2D	50	100

Table 8: PCs and Timesteps for Non-Linear Experiment 2: Variance = 0.9, k = 50

4.4.3 – Non-Linear Experiment 3: Variance = 0.95, k = 25

Principal Component Explained Variance: 0.95, Scaling Factor: 25		
Layer Type	**Principal Components**	**Theoretical Adjusted Timesteps**
Convolutional 2D	7	64

Average Pooling 2D	50	100
Convolutional 2D	22	80
Dropout	50	100
Convolutional 3D	44	96
Average Pooling 2D	50	100

Table 9: PCs and Timesteps for Non-Linear Experiment 3: Variance = 0.95, k = 25

4.4.4 – Non-Linear Experiment 2: Variance = 0.95, k = 50

Principal Component Explained Variance: 0.95, Scaling Factor: 50		
Layer Type	**Principal Components**	**Theoretical Adjusted Timesteps**
Convolutional 2D	7	73
Average Pooling 2D	50	100
Convolutional 2D	22	83
Dropout	50	100
Convolutional 3D	44	96
Average Pooling 2D	50	100

Table 10: PCs and Timesteps for Non-Linear Experiment 4: Variance = 0.95, k = 50

Chapter 5 – Results & Analysis

For result analysis of both linear and non-linear experiments in comparison to the model data, over a course of thirty epochs, the **peak converged** training accuracy (which typically occurred between epochs 23 and 30) is used for direct reference to avoid redundancy of explicating trends of data from 30 epochs. Aside from the training accuracy, the tables for each experiment categorization also includes average time per epoch, expected latency change, actual latency change, and then metric comparisons to the model (in percentages). Expected latency reduction is based directly on the theoretical timesteps associated with each experiment in Chapter 4 – for instance, for the 6 primary layers covered, they were initially unrolled for a value of 100 timesteps each (600 total). The expected timesteps is the sum of the adjusted timesteps after the call to $calculate\_timestep\_ratios()$, and the expected latency is a direct percentage calculation based on the new timestep count (dividend) compared to the original 600 (divisor). With this data, the calculation of imporance is the ratio between the $\Delta Latency$ and $\Delta Accuracy$. Ideally, a ratio higher than 1 implies the latency reduction was numerically more efficient than the accuracy degradation as a result – a very optimized setup theoretically yields a ratio significantly higher than 1 with minimized accuracy degradation in relation to the model.

5.1 – Linear Experiments

5.1.1 – Linear Experiment Result Data

Experiment	Peak Training Accuracy	Average Time Per Epoch (30 total)	Theoretical Latency Reduction
4.3.1	67.24%	00:07:06	45.33%
4.3.2	73.31%	00:09:42	17.33%
4.3.3	83.88%	00:08:29	19.15%

Table 11: Linear Experiment Training Results

Experiment	ΔAccuracy vs. Model	ΔLatency vs. Model	$\frac{\Delta\text{Latency}}{\Delta\text{Accuracy}}$
4.3.1	-33.3%	-32.47%	0.98
4.3.2	-17.88%	-15.28%	0.85
4.3.3	-16.71%	-20.1%	1.20

Table 12: Linear Experiment Result Comparisons to Model

5.1.2 – Discussion

The set of linear-oriented timesteps adjustments, when viewing the direct ratio of latency to accuracy change, yielded values less than ideal. It is a known fact that even the most miniscule tweaks to hyperparameters and established algorithmic approaches can have butterfly effects of unpredictable results; this example reflects this postulate. In light of this first set of experiments failing to retain a solid latency to accuracy ratio, one positive to look at is the result from experiment 4.3.3 – the only experiment in the subset that yielded a ratio higher than 1. This epiphany raised a few points which led to some experimental tweaks in the non-linear experiments:

- It was entirely possible to use principal component analysis to ensure a latency tradeoff was quantifiably more efficient than accuracy degradation; experiment 4.3.3 confirmed this trend, but the missing piece is still retaining accuracy.
- The proposed linear approach created a very violent disparity in timesteps between layers depending on the predefined arguments.

The non-linear experiments, as per formulated suggestion and in response to the above realizations, built on previous tests by trying to shift a focus **primarily** on retaining accuracy first, and then reducing latency utilizing the Sigmoid function approach. Consequently, we would expect to create a more realistic subset of adjusted timesteps that does not alter the backbone of the existing timesteps as dramatically.

5.2 – Non-Linear Experiments

5.2.1 – Non-Linear Experiment Result Data

Experiment	Peak Training Accuracy	Average Time Per Epoch (30 total)	Theoretical Latency Reduction
4.4.1	81.47%	00:09:14	12.67%
4.4.2	82.88%	00:09:28	10.27%
4.4.3	83.88%	00:09:47	10%
4.4.4	85.72%	00:10:02	8%

Table 13: Non-Linear Experiment Training Results

Experiment	ΔAccuracy vs. Model	ΔLatency vs. Model	$\frac{\Delta Latency}{\Delta Accuracy}$
4.4.1	-8.46%	-12.10%	1.24
4.4.2	-6.88%	-9.84%	1.43
4.4.3	-5.75%	-6.84%	1.19
4.4.4	-3.69%	-4.76%	1.29

Table 14: Non-Linear Experiment Result Comparisons to Model

5.2.2 – Discussion

Compared to the results in the previous experiment set, the ratios of latency to accuracy in this subset proved to be more worthwhile – a testament to the suspicions proposed in subsection 5.1.2 potentially holding true. In this instance, taking advantage of the Sigmoid function for timestep adjustment in a non-linear function yielded consistent latency to accuracy ratios exceeding 1 – almost reaching a factor as high as 1.5. However, the issue persists in relation to the accuracy degradation; the lower the accuracy degradation, the more difficult it proved to be to also reduce the latency at a factor significantly higher than 1 to yield marginally impactful results.

5.3 – Analysis

Evaluating the data, a few points of contention for potentially improving results remained related to both the procedural approach to the algorithm applied in this thesis and to more generalized environment facts, including (but not limited to) model size and slight differences in methodology from related works and references.

5.3.1 – Potential Procedural Sources of Error

The heuristic idea of varying timesteps (unrolling less important layers "less") conceptually made sense but proved to be more difficult to implement than expected given the predefined codebase. As a byproduct of this architecture, the work in this thesis was a thought process intended to "limit" the number of times we access and unroll a feature layer, a restructuring of the entire BPTT algorithm may have been required to truly unroll the layers less times. Restructuring the algorithm may not be appropriate without an extensive amount of tweaking to hyperparameters and environment variables, however, as there is also a very high chance that the pretrained ANN models are saturated to the architecture in the existing codebase. In general, there could certainly be a much more efficient place to alter the work done by the BPTT algorithm, but the associated effort involved with restructuring insinuates potential issues.

5.3.2 – Scalable Model and Dataset Sizes

To reiterate, this thesis built its approach based on a smaller (and significantly more uncommon) VGG5 model as opposed to more popular VGG16 and VGG19 architectures. While scalability is implied due to the frames of each model and their incorporation of calculation-intensive convolution layers and fully connected layers (among others), it is not guaranteed until assessed. From a theoretical standpoint, architectures with more layers take much more time to make computations due to the existence of significantly more calculation-based convolutional layers – this introduces a much larger space of memory and graphics card utilization that can be optimized with a new approach. Additionally, a larger model also likely contains more weights

to be adjusted; in general, a larger model may have provided more room to improve upon existing results.

5.3.3 – Heuristic Differences

Restating one of the focal points in Ref. [6], their reference (which served as the backbone to this work) to principal component analysis trends relating to optimized thresholds was based on **principal component percentage-wise changes** between layers from iteration to iteration, and not the direct value of principal components at each layer per iteration. This is another byproduct of a smaller model size; the percentage-wise changes, for a VGG5 architecture, were too minor and the layer count was too small to utilize this approach, meaning that the work in this thesis built off an *extension* of a proven trend. Although there is a conceptual similarity between the two ideas, it could have been enough to prevent significant accuracy retention within the testing phase. It would likely be beneficial to see a similarly applied algorithm from this work onto a larger model and dataset (like VGG16 on the CIFAR100 dataset) to once again observe if this suspicion holds true, but that scalability proof is outside the scope of this thesis.

Chapter 6 – Conclusion

6.1 – Overview

In this paper, we proposed an implemented a variation of principal component analysis to a training algorithm for an increasingly popular driver for machine learning techniques – spiking neural networks. By applying a principal component analysis technique to the existing architecture of our machine learning model under question, and quantifiably assessing and categorizing each layer in terms of its relative importance to the entire model, a restructuring of unrolling steps within the backpropagation through time algorithm aided the goal of reducing latency for efficiency representing that of a human brain, to a tradeoff degree between latency and accuracy as high as 1.43.

6.2 – Reflection and Future Works

After reflecting upon the entirety of this project's undertaking and the procedures associated with it, there were some parametrized and strategical flaws, but the learning process throughout the thesis experimentation process remained extremely educational to understand the inner machinations of spiking neural networks and how they continue to grow in practicality and popularity. Additionally, the experiments offered positive prospects towards this same concept potentially working more practically and effectively on a larger scale. Hopefully the near future will open opportunities for similar implementations on larger models, datasets, and problems to truly test the extent of success that spiking neural networks hold in the field of machine learning and neuromorphic computing.

Appendix A – Modified BPTT Code

```
def bptt(self, x, find_max_mem=False, max_mem_layer=0):
    self.neuron_init(x)
    max_mem = 0.0

    layer_list = [np.array([]) for i in range(len(self.features))]

    prev_pcs = 'principal_components.pt'
    exists = os.path.exists(prev_pcs)

    if exists:
        prev_principal_components = torch.load('principal_components.pt')
        adjusted_timesteps = calculate_timestep_ratios(prev_principal_components,
self.timesteps)
    else:
        adjusted_timesteps = [100, 100, 100, 100, 100, 100]

    for t in range(self.timesteps):
        out_prev = self.input_layer(x)
        for l in range(len(self.features)):
            if isinstance(self.features[l], nn.Conv2d):
                if find_max_mem and l == max_mem_layer:
                    if (self.features[l](out_prev)).max() > max_mem:
                        max_mem = (self.features[l](out_prev)).max()
                    break
                if len(layer_list[l]) == 0:
                    layer_list[l] = self.features[l](out_prev).detach().cpu().numpy()

                else:
                    layer_list[l] = layer_list[l] +
self.features[l](out_prev).detach().cpu().numpy()

                mem_thr = (self.mem[l] / self.threshold[l]) - 1.0
                out = self.act_func(mem_thr, (t - 1 - self.spike[l]))
                rst = self.threshold[l] * (mem_thr > 0).float()
                self.spike[l] = self.spike[l].masked_fill(out.bool(), t - 1)
                self.mem[l] = self.mem[l] + self.features[l](out_prev) - rst
                out_prev = out.clone()
                if t > adjusted_timesteps[l]:
                    continue

            elif isinstance(self.features[l], nn.AvgPool2d):
                layer_list[l] = self.features[l](out_prev).detach().cpu().numpy()
                out_prev = self.features[l](out_prev)
                if t > adjusted_timesteps[l]:
                    continue

            elif isinstance(self.features[l], nn.Dropout):
                layer_list[l] = self.features[l](out_prev).detach().cpu().numpy()
                out_prev = out_prev * self.mask[l]
```

```
                if t > adjusted_timesteps[l]:
                    continue

        if find_max_mem and max_mem_layer < len(self.features):
            continue

        out_prev = out_prev.reshape(self.batch_size, -1)
        prev = len(self.features)

        for l in range(len(self.classifier) - 1):
            if isinstance(self.classifier[l], nn.Linear):
                if find_max_mem and (prev + l) == max_mem_layer:
                    if (self.classifier[l](out_prev)).max() > max_mem:
                        max_mem = (self.classifier[l](out_prev)).max()
                    break

                mem_thr = (self.mem[prev + l] / self.threshold[prev + l]) - 1.0
                out = self.act_func(mem_thr, (t - 1 - self.spike[prev + l]))
                rst = self.threshold[prev + l] * (mem_thr > 0).float()
                self.spike[prev + l] = self.spike[prev + l].masked_fill(out.bool(), t
- 1)

                self.mem[prev + l] = self.mem[prev + l] +
self.classifier[l](out_prev) - rst
                out_prev = out.clone()

            elif isinstance(self.classifier[l], nn.Dropout):
                out_prev = out_prev * self.mask[prev + l]

        if not find_max_mem:
            self.mem[prev + l + 1] = self.mem[prev + l + 1] + self.classifier[l +
1](out_prev)

    principal_components = []
    for x in layer_list:
        if len(x) == 0:
            continue
        principal_components.append(analyze_feature(x, ratio=0.9))

    torch.save(principal_components, 'principal_components.pt')

    if find_max_mem:
        return max_mem

    return self.mem[prev + l + 1]
```

Bibliography

[1] A. Renner, F. Sheldon, A. Zlotnik, L. Tao, and A. Sornborger, "The backpropagation implemented on spiking neuromorphic hardware," 2021.

[2] A. Sengupta, A. Banerjee, and K. Roy, "Hybrid spintronic-CMOS spiking neural network with on-chip learning: Devices, circuits, and systems," *Physical Review Applied*, vol. 6, no. 6, 2016.

[3] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, "Going deeper in spiking neural networks: VGG and residual architectures," *Frontiers in Neuroscience*, vol. 13, 2019.

[4] D. Zhao, Y. Zeng, and Y. Li, "Backeisnn: A deep spiking neural network with adaptive self-feedback and balanced excitatory–inhibitory neurons," *Neural Networks*, vol. 154, pp. 68–77, 2022.

[5] J. A. Nichols, H. W. Herbert Chan, and M. A. Baker, "Machine learning: Applications of artificial intelligence to imaging and diagnosis," *Biophysical reviews*, 2019. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/30182201/. [Accessed: 15-Jan-2023].

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[7] N. Rathi and K. Roy, "Diet-SNN: A low-latency spiking neural network with direct input encoding and leakage and threshold optimization," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–9, 2021.

[8] N. Rathi, G. Srinivasan, P. Panda, and K. Roy, "Enabling deep spiking neural networks with hybrid conversion and Spike timing dependent backpropagation," *arXiv.org*, 04-May-2020. [Online]. Available: https://arxiv.org/abs/2005.01807. [Accessed: 19-Jan-2023].

[9] N. Talebi, A. M. Nasrabadi, and I. Mohammad-Rezazadeh, "Estimation of effective connectivity using multi-layer Perceptron Artificial Neural Network," *Cognitive Neurodynamics*, vol. 12, no. 1, pp. 21–42, 2017.

[10] P. U. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers in Computational Neuroscience*, vol. 9, 2015.

[11] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer, “Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing,” *2015 International Joint Conference on Neural Networks (IJCNN)*, 2015.

[12] S. Lu and A. Sengupta, “Exploring the connection between binary and spiking neural networks,” *Frontiers in Neuroscience*, vol. 14, 2020.

[13] S. Lu and A. Sengupta, “Neuroevolution guided hybrid spiking neural network training,” *Frontiers in Neuroscience*, vol. 16, 2022.

[14] S.-H. Han, K. W. Kim, S. Y. Kim, and Y. C. Youn, “Artificial Neural Network: Understanding the basic concepts without mathematics,” *Dementia and Neurocognitive Disorders*, vol. 17, no. 3, p. 83, 2018.

Andre J. Mitrik

EDUCATION

The Pennsylvania State University, University Park, PA
Schreyer Honors College
Bachelor of Science in Computer Engineering
Graduation: May 2023

WORK EXPERIENCE

Philips Respironics
Site Reliability Engineer Intern *May-August 2022*

- Worked alongside Philips' site reliability and RAD-IT team under an AGILE methodology
- Specialized in upgrading and modernizing Philips' Developer Dashboard site
- Facilitated dashboard's transition from buildpacks to a containerized Docker application
- Deployed new features via CICD pipeline through GitLab, TeamCity, & CORMS process

Performance Test Engineer Intern *May-August 2021*

- Facilitated Philips' performance testing team's transition to a new HTTP and GUI test tool
- Stress tested features released by software developers for pipeline deployment
- Developed a notification database in Java for Philips' Care Orchestrator interface
- Operated cohesively with teams in Pittsburgh and Bangalore

RESEARCH & PROJECTS

Neuromorphic Computing Lab Research
Schreyer Honors College at The Pennsylvania State University *2021-2023*

- Researched machine learning algorithms promoting brain-inspired computer efficiency
- Optimized backpropagation algorithms on spiking neural networks (SNN)
- Performed principal component analysis experiments on machine learning architecture
- Conducted algorithmic performance tests to contribute research in a final honors thesis

Biomechanical Motion Tracking for Golf Studies
Side Project *2020-Present*

- Studying fast-motion biomechanical data and abstract forces in successful golfers
- Categorizing isolated body-part movements for Gears and SportsBox databases
- Creating event-driven optimized approaches to maximize ground reaction forces promoting quantifiably efficient energy delivery in Swing Catalyst software
- Connecting 3D forces and anatomical geometry to golf ball flight laws

Sphero Robotics Intern
Penn-Trafford High School *2018-2019*

- Refactored foundational code powering robotically executed physical applications
- Updated internal software development kits and application programming interfaces
- Coded sensor, light, and movement recognition modules for a Sphero BOLT
- Structured Sphero BOLT prototype motor functionality unit tests

COURSE-WORK

Object Oriented Programming in Java & Python	Operating Systems (UNIX)
Computer Architecture	Database Management Systems (SQL)
Data Structures & Algorithms	Artificial Intelligence
Low-Level Systems Programming (Bash/Shell)	Machine Learning

SKILLS

Programming Languages: Java, Python, C, TypeScript, HTML5, CSS3, SQL, LaTeX
Software/Platforms: Rally, TeamCity, Github/GitLab, Docker, Office, MATLAB, Angular