Efficient NN Training

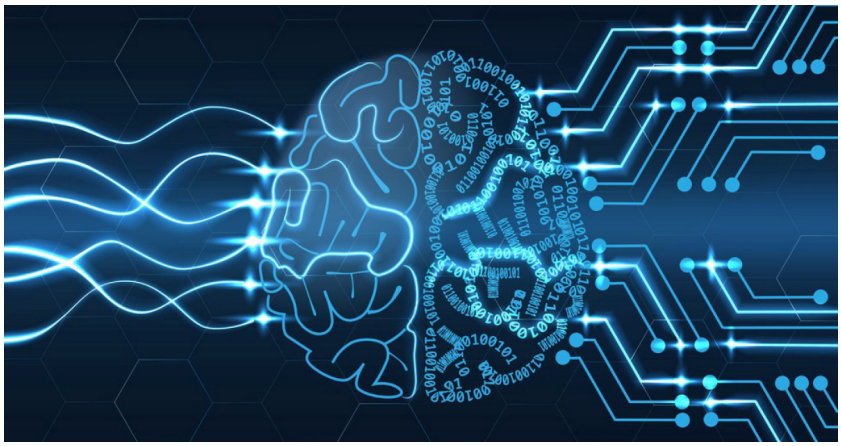
June 8, 2021

Sponsor: BrainChip Inc

Mentor: Dr. Olivier Coenen, Ph.D.

Team Members : Sonya Mohammed, Evon Yip, Loki Sassone, Nakhul Kisan

Team Lead : Sonya Mohammed



Executive Summary

Spiking Neural Networks (SNN) have a range of applications, from signal analysis to time series predictions. A major constraint on neural networks being integrated into use by a wide variety of firms is that they are quite expensive to train, especially on time variant data. These networks need to consider all time points, and optimizing each parameter using current methods, such as backpropagation, takes a lot of storage space, and as a result, computational power.

To aid this, we utilized the Pontryagin Maximum Principle (PMP) to try to increase the computational efficiency of the neural network. We created a preliminary SNN that calculates the gradient descent using the Pontryagin method. To do this we mapped the problem of neural network training into a different space by treating the neural network as a system, derived the Hamiltonian of that system, found the adjoint states of the system, and then optimized the training of the neural network's parameters using the Pontryagin method. This novel Pontryagin Maximum Principle relies on fewer training parameters meaning that less memory is used. Through this, energy is indirectly saved, lowering training costs. All in all, the training dimensions should decrease from $O(n^3)$ to $O(n^2)$.

For our project, we trained an SNN model to learn two different tasks, an XOR logic gate task and a predictive coding task. Both of these models had a very similar structure. The XOR task predicted the output of XOR logic gates using spike inputs at certain time intervals. An input spike was fed into our network at 2 time intervals, and a gating function was used to evaluate these spikes at a third time interval called go-cue, and then output is then pushed into a network of fully connected neurons. The predictive coding task optimized a spike based representation of a sine wave to reproduce the sum of sinusoidal input. Both methods were researched but predictive coding was focused on more for the final deliverable.

For training the predictive coding model, we used the Pontryagin method for the backward in time part of our model. After training through all of our epochs we received output for our predictive coding task that was relatively good with 73.33% accuracy when used with a 0.25 tolerance.

Table of Contents

I. Introduction

i. Motivation and Context

Spiking Neural Networks (SNN) have a range of applications, from signal analysis to time series predictions. A major constraint on these networks being integrated into use by a wide variety of firms is that they are quite expensive to train, especially on time variant data. These networks need to consider all time points, and optimizing each parameter using current methods, such as backpropagation, takes a lot of storage space, and as a result, computational power. Machine learning as a rapidly expanding field focuses extensively on accuracy while typically ignoring computational costs which downplay their important negative environmental and economic impacts.

We want to create a preliminary SNN that uses a gradient descent learning algorithm by way of the Pontryagin Maximum Principle (PMP) to try to increase the computational efficiency of the neural network. This Pontryagin method reduces training dimensions from $O(n^3)$ to $O(n^2)$ which saves energy and lowers training costs.

In general, the SNN is a neural network used for prediction and classification that is modeled after the brain's spiking neurons and their synapses. Our goal is to train the SNN with gradient ascent calculation using Pontryagin Maximum Principle for learning with less memory storage requirement and potentially faster compared to other neural networks. Since it's a fairly new type of neural network and the field is still experimenting with different methods of gradient ascent calculation, we chose to follow a paper [1] that uses the PMP method to apply a gradient ascent algorithm to optimize the SNN. This method is better than backpropagation through time because it relies on fewer optimization parameters, which means that it takes up less memory and can be faster to update.

ii. Importance

The SNN with backpropagation through time method already exists. If the Pontryagin implementation can perform better than the BPTT method on a simple task such as the predictive coding task, gradient ascent calculation using Pontryagin might be able to improve the efficiency of spiking neural networks in general on a much larger scale. This is important as more accurate neural networks allow greater societal impacts as neural networks are used in many fields and can help make advancements and understandings across various domains. Pontryagin also allows less computational resources due to the reduced training dimensions. This has many impacts, especially for the environment, as outlined later in this report.

This research is very beneficial to BrainChip Inc as they work extensively with neural network processors like their Akida chip which boasts high speed, ultra low power on chip learning without retraining. This research can improve and advance BrainChip Inc's current projects and future endeavors as they continue to advance the field of neural networks.

II. Technical Background

We utilize neural networks in our project that were built using PyTorch. A neural network is a model that is trained on given data that is accurately labeled. This data goes through a forward pass in the model and then travels backward in time to compute the gradients based on a loss function, typically a mean squared error loss. In the backwards in time, the weights are updated for the data so after the model is trained the test data can be passed through the model and weights to get the predicted outcome.

For this project we utilize SNNs. These are modelled after the human brain and its neurons and synapses. Each neuron has its own synaptic current that's used to determine the neuron's predicted output and voltage to pass data between neurons in the network. Spiking neural networks vary from other neural networks because they utilize discrete signals of spikes over continuous spikes as seen in Figure 1. They also are better at processing spatio temporal data as opposed to other networks like convolutional neural nets that lose spatial information. Spiking neural networks have been largely overlooked in the past but

advancement of SNNs recently has shown that they have impressive accuracies that can rival alternative neural networks.

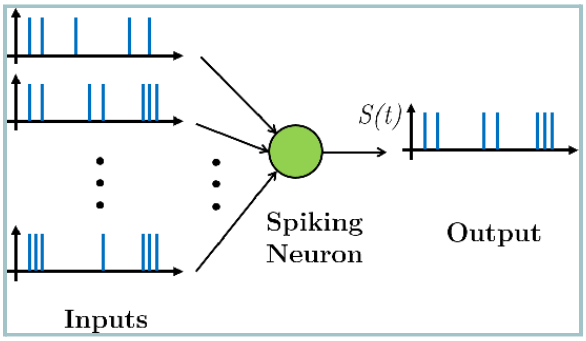


Figure 1: General spiking neural network method

III. Approach and System Design

As a brief overview, we focused mainly on the implementation of the predictive coding task and the XOR task discussed in the paper "Gradient Descent for Spiking Neural Networks" by Dongsung Huh and Terrence J. Sejnowski [1]. We studied the specific model proposed in which they used for both the predictive coding task and XOR task. We spent extensive time studying the Pontryagin maximum principle and deriving the adjoint state equations and weight updates with respect to the cost of the system. The equations derived were applied to the predictive coding task and XOR task. For both tasks, we implemented the forward in time and backward in time calculations of the system for the spiking neural network but worked more extensively on predictive coding for our final deliverable. Some research was also done on the comparison model that used backpropagation through time but we decided to use more time improving and tuning the predictive coding model for better results.

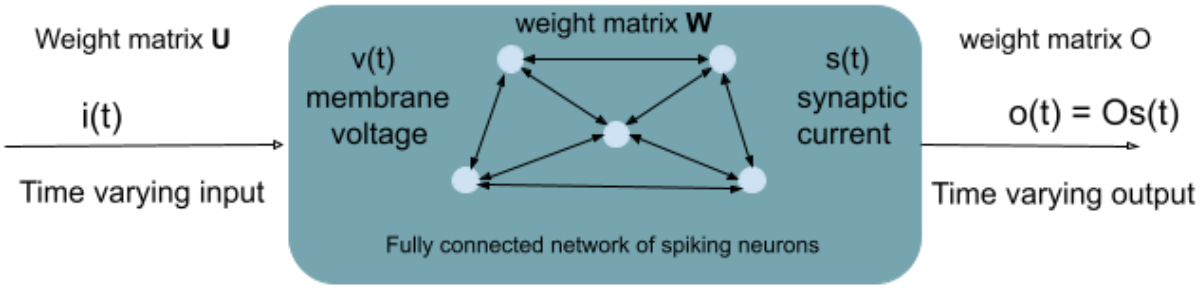


Figure 2: Our spiking neural network model implementation

Figure 2 above shows the model receives a time varying input $i(t)$ and processes it through a network of spiking neurons thus producing a time varying output $o(t)$. The internal state variables are the membrane voltage,$v(t)$ and synaptic current,$s(t)$. Since the neurons form a fully connected network, each neuron receives the time varying input and has their own individual membrane voltages. Based on the paper[1], the dynamics of the neuron network is constructed by linking the dynamics of individual neurons and synapses using a input current vector $I = Ui(t) + Ws(t) + Io$, where U is the weight matrix for the input, $W$is the weight matrix for the recurrent connectivity and $Io$is the tonic current. This current vector I for all neurons is used for the internal state variable calculations of v and s.

Lastly, the output of the network $o(t)$ is defined as the linear readout of the synaptic current such that $o(t) = Os(t)$. The paper [1] also states that the differentiable neural dynamics of the model depend only on the membrane voltage $v(t)$ and input current I for simplicity such that $\dot{v} = f(v, I)$. From the

paper [1], we penalize the readout error and synaptic activity through the cost function $l = (\| o - o_d \|^2 + \lambda \| s \|^2)/2$ with o_d as the desired output and λ as a regularization parameter. Although the figure of the model was provided in the paper [1], it was very misleading and it did not give the readers a detailed overview of the network hence making it difficult to picture the full architecture of the model. It is important to note that Figure 2 is a slightly modified version of the paper's figure based on our understanding after implementing the spiking neural network.

i. Pontryagin Maximum Principle:

Referencing the appendix of the paper on "Gradient Descent for Spiking Neural Networks" [1], the pontryagin maximum principle first obtains the hamiltonian of the system, which is

$$H = \sum_i \dot{p}_{vi} + \dot{p}_{si} s_i + p_o f_o$$

such that p_{vi} is the adjoining state of the voltage membrane, v_i is the voltage membrane, p_{si} is the adjoint state of the synaptic current, s_i is the synaptic current for neuron i. Also, f_o is the costs of the system in the s plane such that $f_o = l(s) = (\| o - o_d \|^2 + \lambda \| s \|^2)/2$. Meanwhile, we define the cost in time domain as $C = \int_{to}^{t1} l(t)\, dt$ and $p_o = -1$ since there is no terminal point constraint for the system at t_1.

From the hamiltonian of the system, the pontryagin method derives the backpropagating dynamics of the adjoint states variables for the membrane voltage v and the synaptic current s:

Adjoint state variables for membrane voltage $\dot{p}_v$, of neuron i

$$-\dot{p}_{vi} = \partial_v f_i p_{vi} - g_i \dot{p}_{si}$$

Adjoint state variables for synaptic current $\dot{p}_s$, of neuron i

$$-\tau \dot{p}_{si} = -p_{si} + l_{si} + \sum_j p_{vj} (\partial_l f_j) W_{ji}$$

with l_{si} referring to the derivative of the loss function with respect to synaptic current (s).
Using the adjoint states $\dot{p}_s$ and $\dot{p}_v$, gradient of the total cost can be obtained by integrating the partial derivatives of the Hamiltonian equation with respect to the network parameters.

From the integration , we get $\partial H/\partial W_{ij} = \sum_i \partial_I f_i * p_{vi} * s_j$, however this is in the s plane.

Hence, the gradient of the cost with respect to the network parameters.

Gradient of the cost with respect to W in time domain is

$$\partial C/\partial W_{ij} = \int_{to}^{t1} \partial_I f_i * p_{vi} * s_j \, dt$$

Gradient of the cost with respect to U in time domain is

$$\partial C/\partial U_{ij} = \int_{to}^{t1} \partial_I f_i * p_{vi} * i_j \, dt$$

Gradient of the cost with respect to I_o in time domain is

$$\partial C/\partial I_{o_i} = \int_{to}^{t1} \partial_I f_i * p_{vi} \, dt$$

Gradient of the cost with respect to O in time domain is

$$\partial C/\partial O_{ij} = \int_{to}^{t1} -p_o(o_i - o_d) * s_j \, dt$$

With the help of some sessions with our mentor to walk through the derivations and re-deriving the gradients in our own time, we verified that the adjoint states and the gradients presented are theoretically and mathematically correct. Although the derivations of the adjoint states were verified, there was a lot of difficulty making a connection between our general spiking neural network model and the pontryagin method. We see the pontryagin method as an indirect method of solving the neural network training by mapping the problem onto a different plane. As the pontryagin approach is used in control theory to find the best possible control for a dynamic system, the approach maps the cost function of the system into the s plane and derives the adjoint states. Hence, to implement this method, we deduce that we have to calculate internal state variables such as input current vector I, membrane voltage v, and the synaptic current s of the network by first going forward in time. Then we can solve for the adjoint states equations going backward in time and update the weights using the gradient of the cost with respect to the network weights $W,\ U,\ I_o$,and O.

After verifying and rederiving the gradients, we proceed to implement the spiking neural network using the pontryagin approach.

ii. Predictive Coding Task with Pontryagin Method:

For the predictive coding task, we have 2 different sum of sinusoidal waves, both which have an approximate period of $T = 1200ms$ and two output channels. The inputs are fed into $N = 30$ non-leaky integrate and fire (NIF) neurons that form a fully connected and recurrent network. It is fully connected since each neuron is connected to every other neuron and the input is introduced to every neuron. It is also fully recurrent as the network connects all the synaptic current outputs of neurons to the inputs of all neurons. Hence, the dynamics of the neural network is constructed by linking the dynamics of individual neurons and synapses using the input current vector I, as discussed in the general SNN model. The non-leaky integrate and fire feature can can be observed from the membrane dynamics equation of the predictive task: $f(v,\ I) = I$, which affects the solutions to our adjoint state equations, solved backwards in time. Lastly we have two time constants. The fast time constant $\tau_s = 10ms$is used to calculate the synaptic current whereas the slow time constant $\tau_{sf} = 1ms$ is used for the output.

In order to use the adjoint state equations we need to solve for all of the components.
From the paper, the membrane voltage dynamics for the predictive coding task is defined as input current vector itself thus giving us,

$$f(v, I) = I$$

$$I = \sum_N I_N = \sum_N \sum_n W_{Nn} s_n + \sum_N \sum_k U_{Nk}\ i_k + I_{oi}$$

Where I is the input current vector,W is a weight matrix of $R^{N\,x\,N}$($N = 30\ neurons$), s is the synaptic current of R^{Nx1}, U is a weight matrix of $R^{N\,x\,number\ of\ inputs}$ ($number\ of\ inputs = 2$ for the two input channels), i is the time varying input (sum of sinusoidal wave) and $I_{oi}$is the tonic current of R^{Nx1} .

From this membrane voltage dynamic, we get two derivatives of f. The first is the derivative of f with respect to v and the second is derivative of f with respect to I for i^{th}neuron,

$$\partial_v f = 0$$
$$\partial_{Ii} f = 1$$

Hence applying the adjoint states equations of $\dot{p}_v$ derived from the pontryagin method, we get

$$\dot{p}_v = -g\dot{p}_s$$

Because the term is zeroed out as $\partial_v f = 0$. Whereas $\dot{p}_s$ of the predictive coding task is

$$-\tau\dot{p}_s = -p_s + \partial_s l + W^T p_v$$

Some changes was made such as $\sum_j p_{vj}(\partial_l f_j) W_{ji}$ where j represents the neuron number and this is summing $p_{vj} * 1 * W_{ji}$ across neurons (which our code does using matrix multiplication) is represented as $W^T p_v$ for simplicity. Since the paper [1] did not specify the value of τ, we set that $\tau = \tau_s = 10ms$ since it involves calculation for the synaptic current (s). Note that $\partial_I f = 1$ for all neurons and l_{si} that refers to the derivative of the loss function with respect to synaptic current s is rewritten as $\partial_s l$.
To find the derivative of the loss with respect to the synaptic current, $\partial_s l$, we first need to look at the general output equation of the network $o = Os$ where O is the readout matrix of $R^{inputNum\ x\ N}$ and cost function $l = (\| o - o_d\|^2 + \lambda\| s \|^2)/2$, then using the chain rule, we derive that

$$\partial_s l = p_o \sum_k (o_{Nk} - o_{kd}) \sum_N O_{kN} + \lambda \sum_N s_N$$

For $k = number\ of\ inputs$ and $N = 30\ neurons$.

For the implementation,

We first define our input wave's frequency and the time frame we run our model over. We chose frequencies of 1/40 Hz and 1/1200 Hz which gives us a sum of sinusoidal wave with an approximate period of $T = 1200\ ms$ for our 2 inputs that we tested over a total time frame of 120 ms to follow the paper[1]. We also increased the time step from 0.1 ms to 0.8 ms to reduce the amount of samples taken. This resulted in obtaining 25 samples every 20ms and allowed the network to learn the appropriate number of samples.

The predictive coding input and desired output is calculated for each time step in the forward in time section of the code. We calculate 4 separate sine waves (s1, s2, s3, and s4) and then sum 2 of them, each with different frequencies ($f1$, $f2$), to create a new wave form. We then stack the 2 inputs to create our vector. Note that the sum of sine waves is created such that $s1 + s2$ and $s3 + s4$ have amplitudes ≤ 1

$$s1 = 0.6 * sin(2 * \pi * f1 * time)$$
$$s2 = 0.4 * sin(2 * \pi * f2 * time)$$
$$s3 = 0.7 * sin(2 * \pi * f1 * time)$$
$$s4 = 0.3 * sin(2 * \pi * f2 * time)$$

$$\text{input matrix} = \begin{matrix} |s1 + s2| \\ |s3 + s4| \end{matrix}$$

We initialized the weights U, O, I_o, and W. We found that different weight initializations heavily affected the output of the network. We went with random initializations for U, O, and W where U is in the range of [-1,1], O is in the range of [-0.05,0.05], and W is in the range of [0,1]. We left I_o initialized to 0.

SNN model implementation (Moving forward in time) — In the forward in time procedure, we needed to calculate the state variables (our voltage and synaptic current). This is done by defining current I, which is essentially a neuron's input modified by the weights $W, U,\ I_o$ and the synaptic current s.

$$I = W^* s\ + U * input\ + I_o$$

Next, we calculate the state variables, synaptic current (s) and gated voltages (g) by passing the voltages (v)through a gating function that has a threshold of +1 and -1, allowing voltage to pass through when it reaches the threshold. Note that ∂_t is the time step such that $\partial_t = 0.8$as discussed earlier whereas $\tau_s = 10ms$ is the fast time constant for the calculations involving the synaptic current.

$$v = v + \partial_t I$$

Voltage v is fed into the gating function to return gated voltage g and calculate synaptic current s,

$$s = s + \partial_t * (-s + g * I)/\tau_s$$

Finally, we calculate the network's outputs o,and also the desired output od with$\tau_{sf} = 1ms$as stated previously.

$$o = O * s$$

$$od[time] = od[time - 1] + \partial_t * (-od[time - 1] + inputs[time])/\tau_{sf}$$

Note: $\partial_t$is just the time step (how quickly we move through the forward in time) and $time$ represents the current time step our state variables were calculated for being calculated

Adjoint states calculation (moving backwards in time) — Since we are training in batches of size 15, we first run through the time steps in the feed forward, stop after every 15 milliseconds to run backwards in time, and calculate the adjoint state variables and the parameter gradients with respect to cost. These gradient equations are given in the previous section on the Pontryagin Max Principle. Whereas the adjoint states equations for the predictive coding task were previous derived

Weight Updates — We ended up updating our parameters using Standard Gradient Descent (or ascent in the case of using the Pontryagin method) with a learning rate i.e. $W = W + dW$, but experimented with other methods, such as an adam optimizer.

The two figures below give a simple visualization of our network and the implementation of the forward in time and backwards in time process.

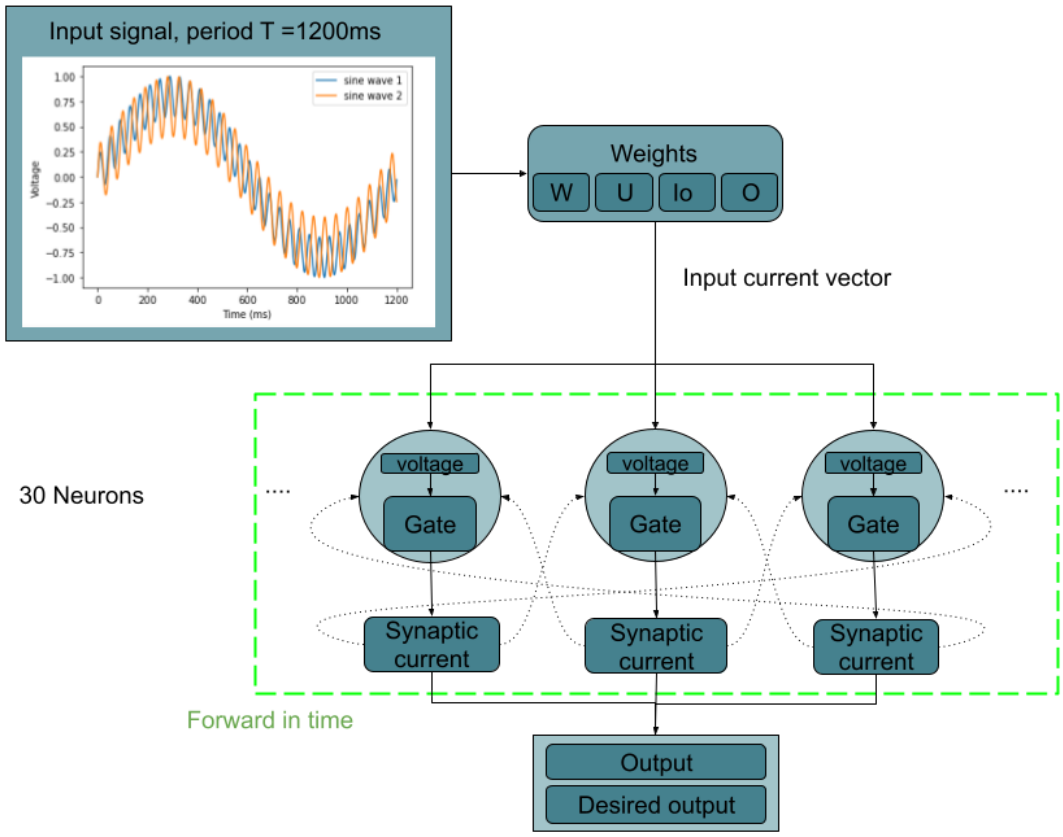


Figure 3: The green dotted box shows the forward in time process of the predictive coding task for a fully recurrent network of 30 neurons.

Internal state variables such as membrane voltage v, gated voltage g,and synaptic current $s$are calculated for each neuron. Note that the dotted lines show the dynamics of the recurrent network of neurons which is constructed by linking each neuron and their synaptic current using the input current vector.

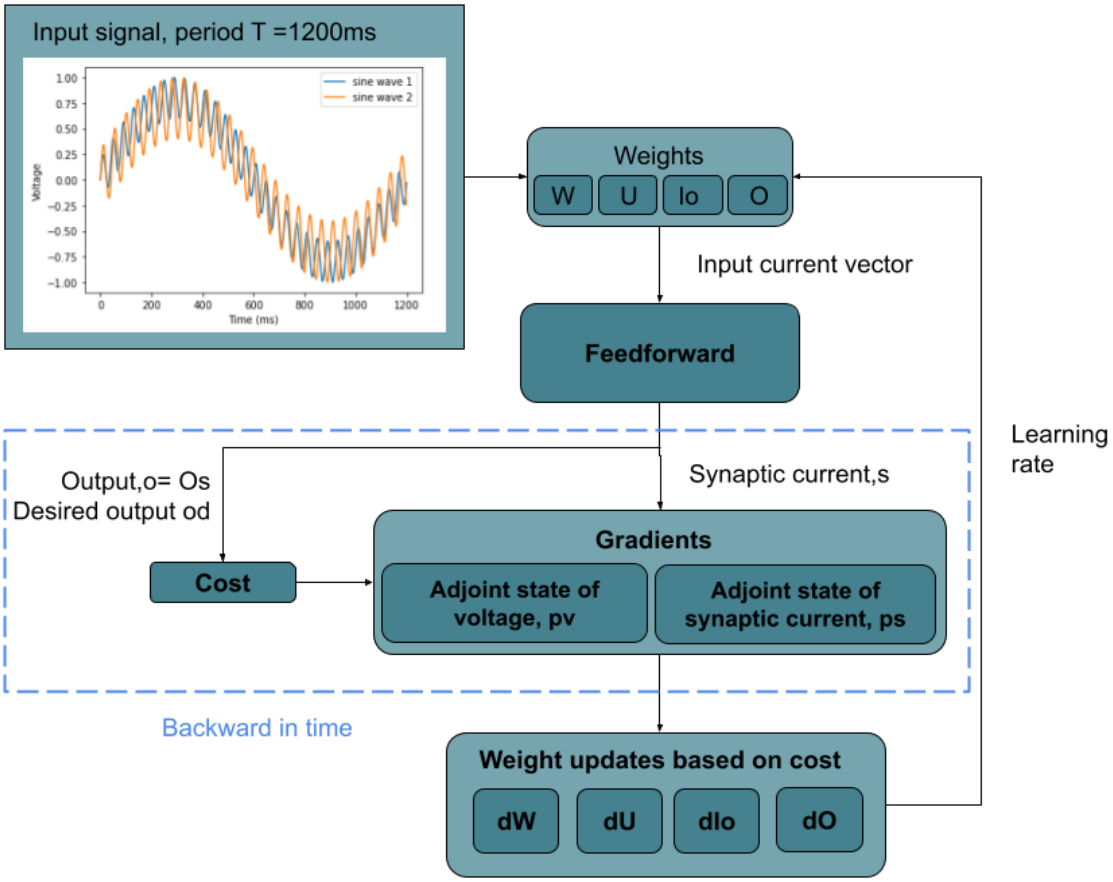


Figure 4: The blue dotted box shows the backward in time process for the predictive coding task. Calculations of the cost and adjoint states of the voltage v and synaptic current s were made backward in time. After completing the calculations for all the batches, weight updates based on the cost of the network was calculated and the weights were updated with a learning rate of 0.01

iii. Predictive Coding with SGD Pseudocode:

1. Define variables
   - Set amplitude and input signal frequency, total time
   - Set neurons threshold voltages = ± 1, time constants τ_s and τ_{sf}
   - Set epochs and update's learning rate,batch size = 15, inputs channel = 2, outputs channel=2

2. Initialize weights to random values
   - Synaptic current weight matrix W
   - Input signals weight matrix U
   - Readout matrix O
   - Vector tonic current Io

for each epoch:
    reset current_batch_iteration = 1

    3. Define input and implement forward in time of fully connected network of neurons

```
for i in range(total_time):
        Stack s1 and s2 as inputs = [s1,s2] where s1, s2 are sums of numpy sine waves

    4. Implement the SNN model
       - Input current I = W*s + U*inputs + Io for all 30 neurons
       - Calculate neuron membrane voltage
       - Gating function for voltage membrane that returns an arbitrary value of 1/∂ or 0
       - Reset membrane voltage to 0 if membrane voltage  v >=1
       - Calculate synaptic current
       - Calculate desired output od and output o = Os

    5. Go backwards in time with Pontryagin method after each batch
    if at the end of batch or at the end:
        for j in range(i, current_batch_iteration * batch_size, -1):
            Calculate PMP adjoint states of synaptic current and membrane voltage
        Update current_batch_iteration after backpropagation

    6. Update weights W,U,O and Io using Standard Gradient Ascent ( W = W+dW)
```

One of the main challenges our team faced with the predictive coding task was simply visualizing the problem. Most of our team is familiar with neural networks with a feed forward structure (similar to convolutional networks), and had difficulty visualizing a fully recurrent, spiking neural network where synaptic current from each neuron is fed into all of the other neurons. The research paper [1] was vague about the structure of the model. As a result, we initially built our model with a flattened, recurrent structure where 29 neurons would learn the input and a 30th final neuron would interpret these results using the synaptic current these other neurons fed it to generate output. Through discussions with our mentor we discovered this was the wrong methodology and we actually needed to change the structure so that there was no final neuron. Rather, input needed to be fed into all of the neurons separately, evaluated using the weights and passed through the model forward in time, and then output needed to be calculated from the synaptic current of each neuron along with the readout (O) of each neuron. This makes sense because the readout (O) was one of the parameters we were optimizing.

iv. Predictive Coding with Adam Optimizer PSEUDOCODE:

```
1. Define variables
2. Initialize weights
3. Define the adam optimizer class and create instance
        class AdamOptim():
         - def __init__()
         - def adam_update():  updates the weights using mean and variance of the gradient

for each epoch:
        reset current_batch_iteration =  1
        zero out the gradients (i.e. dW = np.zeros())
        4. Define input and implement forward in time for fully connected network
         for i in range(total_time):
                5. Implement the SNN model
```

6. Go backwards in time with Pontryagin method after each batch
if at end of batch or at the end:
for j in range(i,current_batch_iteration * batch_size, -1):
7. Calculate adjoint states for synaptic current and membrane voltage
8. Update current_batch_iteration after backpropagation
9. Update weights using adam optimizer after going through all the batches for one epoch
w_raw = concatenation of flatten weights;
dw_raw = concatenation of parameter costs;
w_adam = adam.update(time, w_raw,dw_raw)
extract weights W,U,O,Io from w_adam and reshape

Before including the Adam optimizer, we first tried updating the parameters using Standard Gradient Descent (SGD), meaning W = w + dW, but our mentor suggested the use of Adam optimizer to get better results. Adam updates our weights by applying a transformation of the mean and variance of the parameter gradients to our parameters, and it also keeps track of the two throughout all of the epochs to prevent divergence of the parameters. We ran into some trouble with our implementation with Adam and were not getting as good results when we compared to our SGD. Future teams may have to debug this to create results that are more comparable to the paper's [1].

v. XOR Task with Pontryagin Method

Our second task was to implement XOR logic on a delayed learning task. We used discontinuous input spikes that caused our network to evaluate an output at a later time interval. This task was a little similar to the previous one and can be thought of as essentially the predictive coding task with perforated input and a time delayed output. One difference, though, was that the membrane voltage dynamics equation is different from the predictive coding task. Therefore, the equations for the adjoint state equations were also derived differently. The XOR voltage dynamics are defined as:

$$f(v, I) = (1 + cos(2\pi v))/\tau_v + (1 - cos(2\pi v))I$$

where τ_v = 25ms

Due to this difference in membrane dynamic, our adjoint state variables $\dot{p}_v$ and $\dot{p}_s$ become:

$$\dot{p}_{v_i} = -(1 - cos(2\pi v_i)) * p_{v_i} + g_i * \dot{p}_{s_i}$$

$$\tau\dot{p}_{s_i} = p_{s_i} + p_o * (\sum_j (o_j - o_{dj}) * \sum_i O_{ji} - \lambda) - \sum_j p_{v_j} * (1 - cos(2\pi v_i)) * W_{ji}$$

where$\tau = 20ms$,
Also, since this is defined as our the membrane dynamics for one of our state variables, our voltage update becomes:

$$v = v + \partial_t * ((1 + cos(2\pi v))/\tau_v + (1 - cos(2\pi v))I)$$

Where ∂_t is the time step.Besides these differences in variable definitions, the forward and backwards in time sections of the XOR follows the exact same procedure as the predictive coding with the one caveat being that the gating function is only evaluated at certain intervals.

In general, the delayed-memory task performs the exclusive-or (XOR) operation on the input history stored over extended duration. The network receives binary pulse signals of either a positive or negative through an input channel and a go-cue, taking values [0,1], through a different channel. A go-cue is used to tell the network when to train and output either a positive, negative or null output. For

our implementation, this go-cue essentially fires the gating function whenever a value greater than 0 is encountered. The network should output either a positive or negative signal depending on if the two binary inputs match the XOR gate logic. If the network senses only one input signal then the network should generate a null output. We used three different time constants to control when the model would input each of the positive/negative signals and for the go-cue. The first time constant, $\tau_f = 5ms$, was used to determine when the model would get the first input signal. The second time constant, $\tau = 20ms$, was used to determine when the model would get the second input signal. The third and last time constant, $\tau_v = 25ms$,was used for the go-cue to determine when the model would output a positive, negative or null output depending on the two previous input signals.

vi. XOR PSEUDOCODE:

1. Define variables
   - Set amplitude and input signal frequency, total time
   - Set neurons threshold voltages = ± 1, time constants $\tau_{v,}$ τ_f andτ
   - Set epochs and update learning rate, inputs channel = 1, outputs channel=1
   - *Pre-compute go_cue, input spikes, and desired output*
2. Initialize weights to random values
   - Synaptic current weight matrix W
   - Input signals weight matrix U
   - Readout matrix O
   - Vector tonic current I

for each epoch:

reset current_batch_iteration = 1

3. Define input and implement forward in time of fully connected network of neurons

for i in range(total_time):

- Feed input current I = W*s + U*inputs + Io to all 80 neurons
- Calculate neuron membrane voltage
- Gating function for voltage membrane that returns an arbitrary value or 0
  - Tried firing at the go_cue
- Reset membrane voltage to 0 if membrane voltage v >= 1
- Calculate synaptic current
- Calculate desired output od and output o = Os

4. Go backwards in time with Pontryagin method
5. Calculate PMP adjoint states of synaptic current and membrane voltage
6. Update weights W,U,O and Io using Standard Gradient Ascent (W = W+dW)

The XOR task faced similar challenges to the predictive coding task along with us having trouble outputting predictions at a delayed time interval. We tried firing the gating function at go-cue, meaning we tried calculating the output only at the time intervals we had input and then summing, but this did not work. Right now, we are using an alternative method where our lead is using the gating function to evaluate at the go-cue, but a better solution is required. Future teams would need to debug this issue in order to create accurate results for the XOR task. One other suggestion, besides toggling the gating function, could be to train in batches of 25 ms so that each batch observes one instance of XOR input and output.

IV. Results

We were able to complete the predictive coding task. We were able to achieve an accuracy of 0.733 when we used a tolerance of 0.25 for the difference between desired output and our model's output. Our best model also had a standard deviation of 0.1358 for the values between the desired output and the model's output. The two graphs below show the graphs of the two desired outputs and two predicted outputs from our model. As you can see the output follows the desired output pretty well but we didn't have a model that could do the predictive coding task as accurately as the paper [1] was able to do.

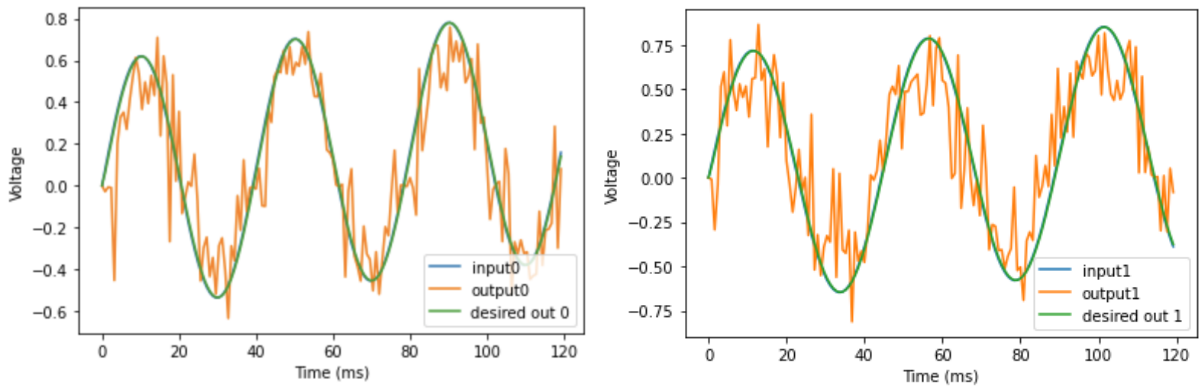


Figure 5: The model's predicted output is the orange line and the actual desired output is the green line.

In contrast with the results of the research paper [1] we studied below, we were not able to obtain as accurate results.

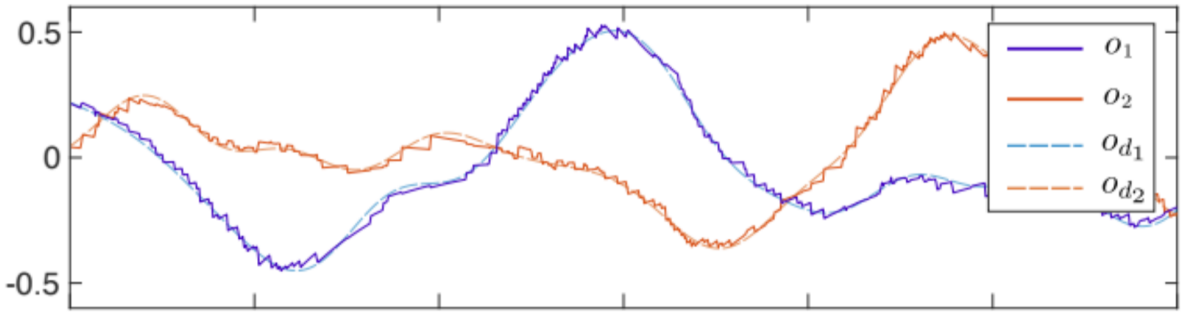


Figure 6: The studied research paper's [1] output and desired output, the dotted lines are the desired outputs and solid lines are the obtained output.

While optimizing our results for the predictive coding task, we used ablation study to determine the best values for the hyperparameters delta, epochs, learning rate, and lambda. We ran different values of all of these hyperparameters and determined the best values by graphing the added magnitude of cost, od-o. The four graphs below show the graph of using different hyperparameter values versus added magnitude of cost.

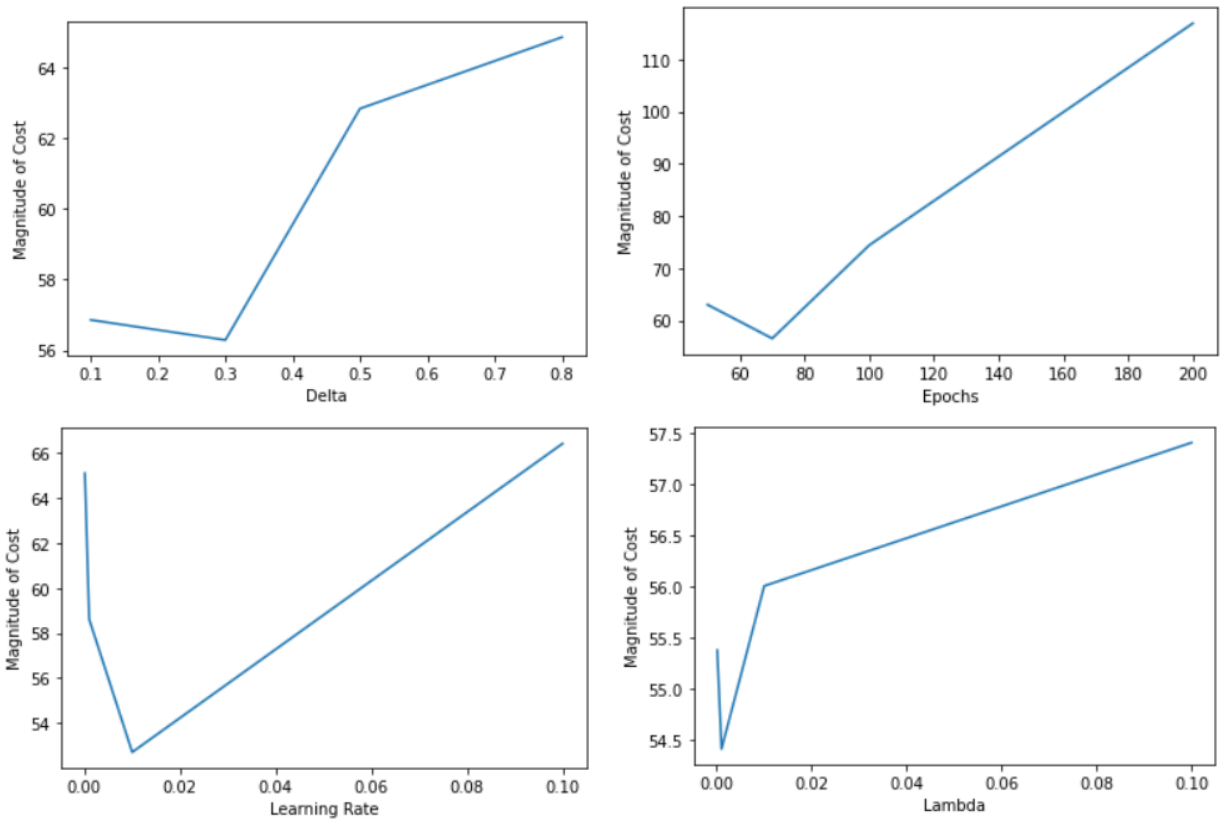


Figure 7: Plots of different hyperparameter values versus cost magnitude of the model.

After running the ablation study we determined that the best values for the four hyperparameter values were ∂=0.3 (which is used in gating function for voltage membrane that returns an arbitrary value of 1/∂ or zero), learning rate=0.01, epochs=70, and cost function regularization parameter λ =0.001.

```
Comparing output with desired output:

Standard deviation of abs(o-od):  0.13059409


Accuracy:  0.76  with tolerance:  0.25
*Note that this is not the most suitable method of accuracy measurement
```

Figure 8: Resulting standard deviation of absolute difference between output o and desired output od, including a measure of accuracy with tolerance of 0.25

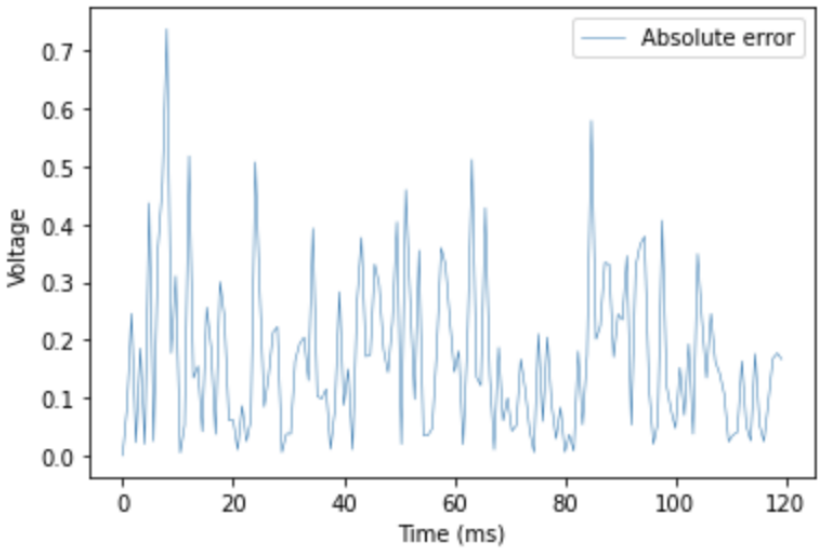


Figure 9: Plot of absolute error of desired output and output.

Note that the accuracy measurement in Fig. 8 is not the most suitable because the predictive coding task is not a classification task. Hence, a plot for the absolute error between the desired output and output is made.

V. Project Impacts

i. Environmental:

Currently most neural networks do not emphasize reduced computational power. Machine learning as a field is actively growing and emphasis is placed on having more accurate networks which often means an increase in parameters and layers in neural networks which drives up computational power [2]. This paper's method of utilizing the Pontryagin Maximum Principle in a spiking neural network allows an increase in accuracy as well as reduced training parameters from N3 to N2. Reduced dimensions can decrease computational power which will save energy and create more sustainable neural networks. This is an important consideration when deciding what machine learning model to use and how they will perform at larger scales. For example, large NLP neural networks, such as GPT-2, can produce tons of CO_2 emissions, up to about 626,155 pounds. This is the same amount of emissions that roughly 6 petrol cars emit in their entire lifespans [3]. So, if the use of SNN with PMP method is able to decrease power consumption and training time by even just a few percentage points, at scale, this could translate to tons of carbon emissions being cut-out. Hence, the reduction from N3 to N2 is a significant difference and will drive down computational power which reduces the amount of heat produced and energy used in computing for a more sustainable environment.

ii. Economics:

The model of a trained SNN is a cost effective alternative to some other types of NN used within the finance industry. Improved efficiency and energy consumption at a larger scale will greatly reduce the cost for industries even more. This is important to the industry because many neural networks are used to detect financial fraud detection, management, and forecasting. For example, as stated in the financial time series prediction study using SNNs, networks were evaluated using a real trading measurement to

view the overall profitability in a year [5]. The results showed that spiking networks outperformed other NNs such as the multilayer perceptrons and dynamic ridge polynomial neural network by as much as 5% when predicting financial time series datasets. This will decrease the cost of operating a NN in the field of finance as Linthucum, chief cloud strategy officer at Deloitte, claimed that “Costs will come down as these systems become much more optimized, using less storage and compute power” [6].

One constraint of this project is that expansion to a larger scale will require funding for the equipment and time. The research and formulations of the SNN gradients using the Pontryagin Maximum Principle requires extensive knowledge on both classical mechanics and control systems. The initial training of the spiking neural network consisting of 30 to 80 interconnected neurons will require funding for equipment and systems with a large memory capacity and that can handle the heavy processing. Since this theory of using PMP for gradient ascent has not been proven to perform better for all tasks, it might not receive much attention and support. Also, since a fully interconnected neural network model is not the most effective way of training, more research needs to be done to find a more effective network model.

VI. Conclusion:

In summary, our team was able to successfully implement the Pontryagin Maximum Principle for the backwards in time in a spiking neural network of the predictive coding task. The model achieved relatively good accuracy on the output for the predictive coding task. For the delayed XOR task we were also able to implement the model using the Pontryagin method. Unfortunately we were not able to achieve highly accurate results as we had to shift our focus to predictive coding.

However, the XOR model we built can be further improved on by future teams to increase the accuracy. The team can aim to fine tune the model by running ablation studies and verifying that the architecture for the XOR model is accurately implemented as we believe it to be. They can use the suggestion stated previously on training in batches of 25 ms so that each batch observes one instance of XOR input and output besides toggling the gating function. . They can also work further on tuning the predictive coding model and implementation of the adam optimizer so we can have more comparable results to the research paper [1] we followed for our project. A second team can work on the backpropagation through time method for unfolding the RNN and building a comparison model to verify if Pontryagin runs more accurately and quickly as our goal was to have Pontryagin be better in accuracy and efficiency so that it can be used on a larger scale to lower computational costs that impact the environment and economic resources.

The Pontryagin method’s strengths are that it requires less training parameters, i.e. a reduction from $O(n^3)$ to $O(n^2)$. This is a significant difference that can lead to greater societal impacts. Training neural networks is immensely time and energy consuming and on a larger scale this method can greatly reduce the amount of time, energy, and resources consumed. Apart from environmental and economic impacts that were discussed in detail above, more efficient neural networks are critical to advancing and improving the machine learning industry. This is a rapidly growing field and as this technology is being utilized there is more emphasis on faster innovation and results despite computation costs. However, having more efficient systems can allow faster and less costly training which allows more innovations in the field to occur. Spending time researching theories that can increase efficiency is very important. Neural networks make predictions that improve society as they are used across various fields and are rapidly entering new ones. More efficient neural networks are critical to the growth of the machine learning industry and the Pontryagin method that we researched is one method that can help improve neural networks to work toward achieving a more efficient machine learning industry.

We can see the great impact that the Pontryagin method, and other network efficiency methods can have. Considering the short ten weeks we had to thoroughly understand the technicalities of the Pontryagin method and the complexities in applying it to a spiking neural network model with a fully

recurrent network structure that none of us had previous experience in, we consider our predictive coding task to be a success.

VII. References:

[1] Huh, Dongsung & Sejnowski, Terrence J. "Gradient Descent for Spiking Neural Networks." 32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Dec3-8, 2018, Montréal, Canada

[2] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, Hakan Grahn, "Estimation of energy consumption in machine learning", Journal of Parallel and Distributed Computing, Volume 134, 2019, Pages 75-88, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2019.07.007.

[3] Brownlee, Jason. Gentle Introduction to Adam Optimization Algorithm for Deep Learning. Machine Learning Mastery. 2021. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[4] Strybell, Emma., Ganesh, Ananya., McCallum, Andrew. "Energy and Policy Considerations for Deep Learning in NLP." University of Massachusetts Amherst 2019. https://arxiv.org/pdf/1906.02243.pdf

[5] Reid D, Hussain AJ, Tawfik H. "Financial Time Series Prediction Using Spiking Neural Networks." PLoS ONE 9(8): e103656. 2014. https://doi.org/10.1371/journal.pone.0103656

[6] Gossett, Stephen, "How to reduce AI computing costs." builtIn. June 2020. https://builtin.com/artificial-intelligence/ai-computing-cost-reduction