MSML 604 - Final Project

Problem Formulation

The colossal failure of the United States military during the Vietnam war should not be overlooked when considering the future of war, specifically its intersection with artificial intelligence. American troops were better armed, better equipped, and trained more extensively[1] than the North Vietnamese troops yet still they were forced out of the conflict. While there are several reasons we lost the war, such as its highly political nature, only two are of importance for the purposes of this discussion: local environment conditions and enemy stealth/inventiveness. American soldiers' unfamiliarity with Vietnam's subtropical climate, landscape, and foliage proved to be very consequential[1]. They stumbled through difficult terrain and encountered dangerous wildlife during long periods of blistering heat and monsoonal rain storms just looking for the enemy. The North Vietnamese Army probably assumed they wouldn't be able to successfully engage the U.S. with a head on approach, so they let nature do it instead by hiding in the mountains and their extensive underground tunnel systems[1]. In 1959 we had just under 1000 military advisors deployed in Vietnam, by 1964 President John F. Kennedy had increased that number to 23,000, and when President Lyndon B. Johnson took over the presidency in the same year, his first act was to further increase the number of American troops in Vietnam to 184,000[2]. Obviously the American military failed to learn from their mistakes during the war, but I hope that ignorance doesn't bleed into the next one because it is evident that firepower, numbers, training, and money aren't enough to win a war. The smooth propagation of expert battlefield knowledge throughout its military presence is arguably the most dangerous weapon a country can equip itself with.

There are a couple of different ways AI can be employed to help level the playing field when it comes to exploiting home field advantage, one research branch of particular interest is computational neuroscience. Recent advancements in the field, with respect to neuromorphic computing, have given us extremely low power computational hardware that is founded upon event-based processing and fine-grain parallelism. This computational efficiency opens the door for a neuromorphic-style Internet of Military Things (IoMT), defined as a complex network of interconnected military entities that interacts with and adapts to the surrounding physical environment. Entities include but are not limited to: sensors, vehicles, robots, UAVs, and human wearable devices[3]. In addition to the standard person-to-person message passing channel, most IoT architectures support person-to-environment communication as well. Here, sensory devices are installed on either natural objects such as rocks, caves, and trees or inanimate objects like billboards, buildings, and buoys[3]. Either way analog data from the local environment is detected and extracted by person-to-environment hardware entities before being passed onto person-to-person (or some flavor) devices. The point of emphasis is that IoMT devices are meant to be ubiquitous enough to form a seamless data fabric that spans the warzone.

So far we have assumed every single device in the network directly connects to the same nearby cellular tower without any issue. However in the real world, particularly in a war torn region, cellular connection is unstable and unsafe and therefore cannot be relied on. Consider a distribution of a platoon across the battlefield, where soldiers are spread out and equipped with a cell phone connected to a singular cell tower. The traffic on such a cellular network changes over time, and too many client requests (or too large of responses) causes network congestion. To combat this we formulate a resource allocation optimization problem to swap the soldier's mobile devices with antennas to maximize cell coverage. At first glance we can just just install as many antennas in the battlefield as possible to ensure a good cellular connection, but at second glance we observe the antennas can't be too close together because their signals interfere, so unfortunately this seemingly simple problem is a little more complicated under the hood. The optimal antenna placement can be found iteratively using graph-based combinatorial optimization models, specifically those that find the maximal independent set (MIS) of a network[4]. An example solution to an arbitrary device network is illustrated in Figure 1 below, where the blue circles represent soldiers with cell phones and the antennas represent soldiers with antennas:

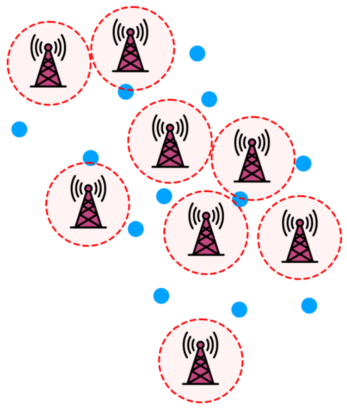


Figure 1: Soldier-Cellular Network with Antennas

Explanation of Paper

The next section, Background, consists of three subsections, Graph Theory, Spiking Up, and Neuromorphic Constraint Optimization, all of which have a unique purpose. I kept the first subsection, Graph Theory, short relative to the second, Spiking Up, because I assume that my audience is more adept at mathematical optimization than neuromorphic computing. The third subsection, Neuromorphic Constraint Optimization, highlights the intersection of ideas introduced in the previous two subsections. The last section, Discussion & Analysis, comprises three subsections: Walkthrough, Benchmarking, My Take, and What's Next? The first of the four just explains what happens during my code's execution. The second subsection provides a very promising performance benchmark of neuromorphic chips relative to the standard cpu-based architectures. In the last subsection I give my opinion of the project after having completed it.

Background - Graph Theory

MIS

The MIS is defined as the largest independent set that is not a subset of any other independent set. Intuitively, no nodes in the set share an edge. The independence number of a graph is the cardinality of the maximum independent set. In Figure 2a below, the MIS of an arbitrary IoT network is denoted by purple nodes. In the adjacent diagram (Figure 2b) the IoT network is restructured, spectral clustering or something like that, such that the MIS is connected by a single node (or node cluster).

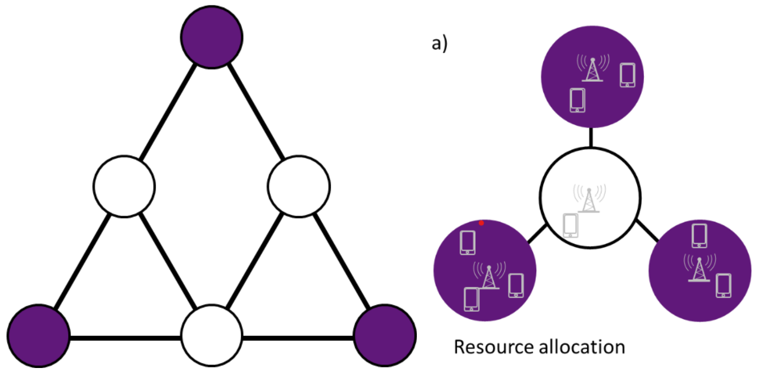
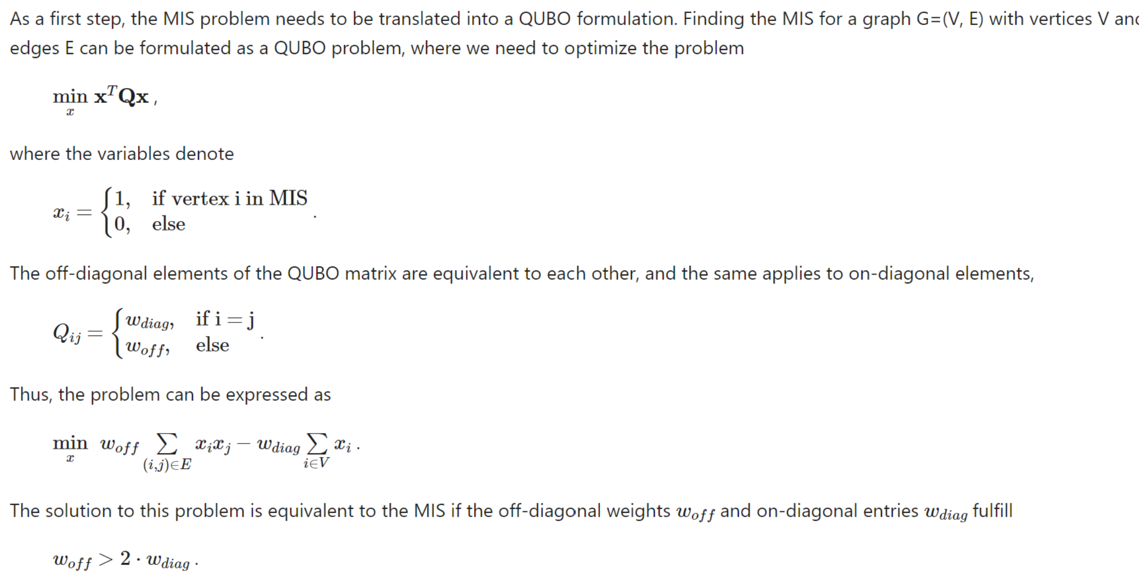


Figure 2: (a) MIS and (b) Resource Allocation - restructured graph

QUBO

Finding the MIS of a graph is a classic example of combinatorial optimization, and therefore can be reformulated into a quadratic unconstrained binary optimization (QUBO) problem. Note that I took the MIS to QUBO reformulation, defined in Figure 3 below, straight from a tutorial in the lava-opt library[5].



As a first step, the MIS problem needs to be translated into a QUBO formulation. Finding the MIS for a graph G=(V, E) with vertices V anc edges E can be formulated as a QUBO problem, where we need to optimize the problem

$$\min_x \mathbf{x}^T \mathbf{Q} \mathbf{x} \,,$$

where the variables denote

$$x_i = \begin{cases} 1, & \text{if vertex i in MIS} \\ 0, & \text{else} \end{cases} .$$

The off-diagonal elements of the QUBO matrix are equivalent to each other, and the same applies to on-diagonal elements,

$$Q_{ij} = \begin{cases} w_{diag}, & \text{if i = j} \\ w_{off}, & \text{else} \end{cases} .$$

Thus, the problem can be expressed as

$$\min_x \; w_{off} \sum_{(i,j)\in E} x_i x_j - w_{diag} \sum_{i \in V} x_i \,.$$

The solution to this problem is equivalent to the MIS if the off-diagonal weights w_{off} and on-diagonal entries w_{diag} fulfill

$$w_{off} > 2 \cdot w_{diag} \,.$$

Figure 3: QUBO Formulas

Background - Spiking Up

The Road To Biologically Plausible Machines

While there are a million different ways in which machine learning can be used to benefit society, the underlying goal of the field is to put the human brain on a computer. This subfield, dubbed computational neuroscience, works to understand the principles that govern the development, structure, and cognitive abilities of the nervous system[6]. Thus computational neuroscience can be thought of as a slow march on the yellow brick road of biological realism, starting at the Von Neumann architecture with a destination of true artificial intelligence. This march can be better explained by shedding light on how the neural network has evolved over time since its invention. As seen in Table 1 and the subsequent summaries, the history of the NN can be partitioned into three generations, each of which is explored. But since contemporary models and algorithms in computational neuroscience are obviously derived from the latest generation of biological machines, we introduce its core concepts in the next section.

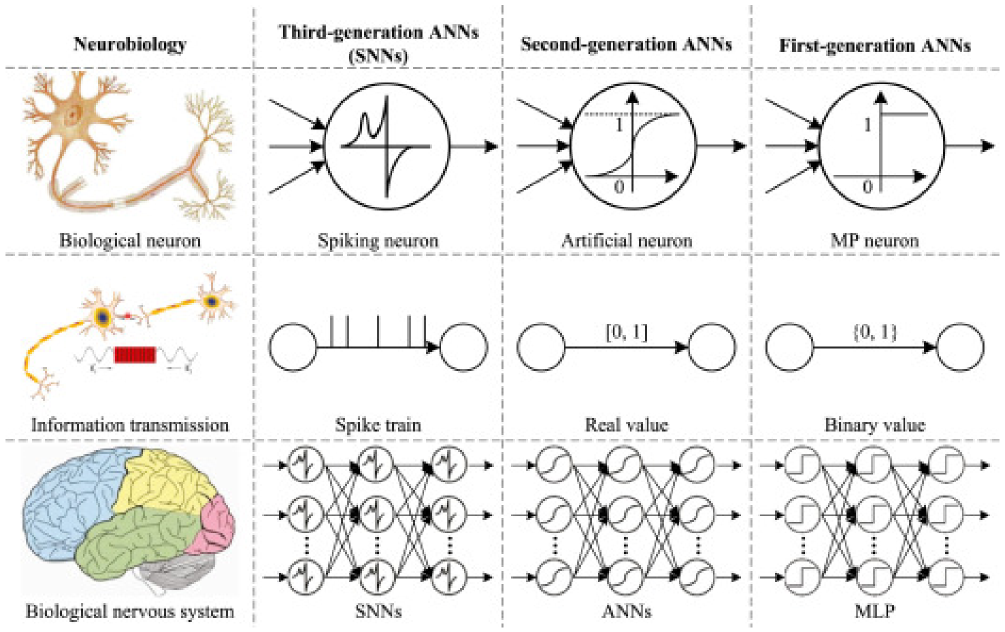


Figure 4: History of Neural Networks

1. **First-Generation** - This network was based on neurons that acted as threshold gates or perceptrons. Its output was binary, 0 or 1, and therefore did not involve a nonlinear activation function.
2. **Second-Generation** - your standard feedforward artificial neural network (ANN) along with its dozens of flavors (RNN, CNN, etc).
3. **Third-Generation** - employs spiking neurons, also known as integrate-and-fire neurons, which more closely model the biological neurons' activity relative to the first two generations.

Spikes in the Brain[7]

What? A spike, or action potential, is a short burst of electrical activity which occurs when a neuron sends information down an axon, away from the cell body. A spike, roughly 100 mV in amplitude as seen in Figure 5, can be thought of as the electric current being passed to downstream neurons. A temporal sequence of these action potentials generated by the neuron are called spike trains. Neurons send messages electrochemically, or via electrically charged chemicals called ions. The most important ions in the nervous system are sodium, potassium, calcium, and chloride.

Why? Our sensory neurons encode information about our surrounding environment, like our five senses, into sequences of voltage spikes with distinct temporal patterns called spike trains. The brain communicates in the universal language of the *spike*. This form of information representation and communication allows biological neurons to disseminate information extremely efficiently, its main advantage over second generation networks. Therefore understanding the lower-level characteristics of these neurons is critical to understanding not only how our brain ingests information, but also how it is stored and used.

How? Neurons are surrounded by a thin lipid bilayer which acts as an insulator for its internal conductive saline solution from the extracellular medium. This membrane is embedded with numerous types of ion channels which allows it to control the flow of ions in and out of the neuron. These voltage-gated ion channels rapidly depolarize and repolarize the transmembrane potential by manipulating the throughput of the four previously mentioned ions. This is how spikes are generated in the brain. The following figure provides more insight into the underlying stages of an action potential:

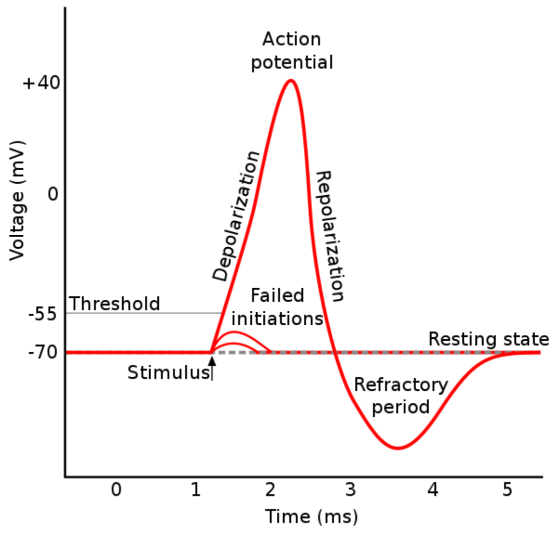
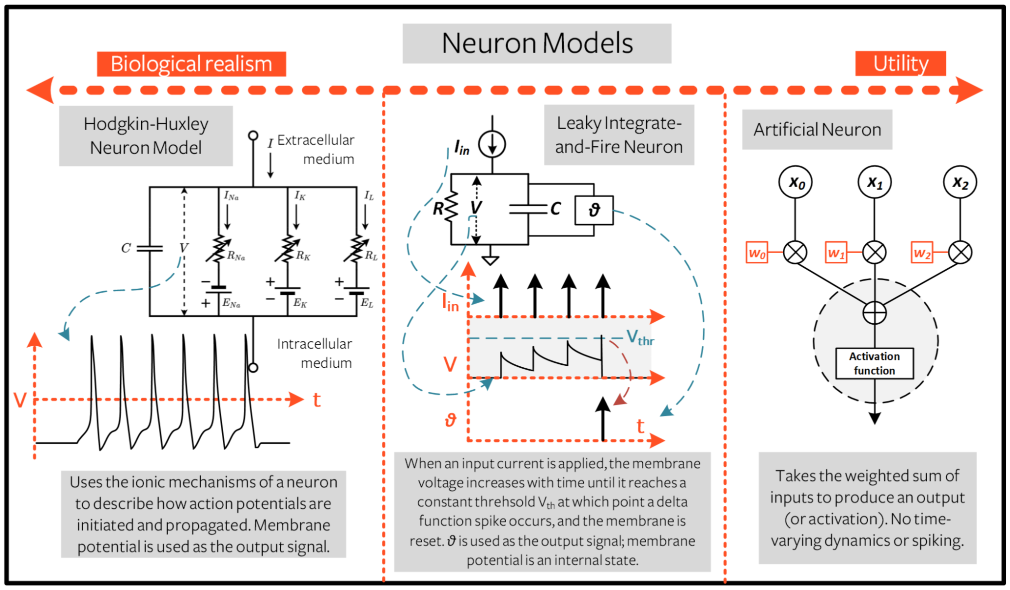


Figure 5: Labeled Action Potential Graph

Neuron Models

To properly compare and contrast the extensive and eclectic mix of neuron models in existence, we map them to the biological-realism domain. In other words, we correlate the neuron models based on a single, unifying tradeoff metric: plausibility vs utility. The tradeoff defines the inverse relationship between the neuron models' bio-realism and their actual utility to deep learning on Von Neumann machines. The bullet points below introduce each of the neuron models presented in the following figure:



Uses the ionic mechanisms of a neuron to describe how action potentials are initiated and propagated. Membrane potential is used as the output signal.

When an input current is applied, the membrane voltage increases with time until it reaches a constant threhsold V_{th} at which point a delta function spike occurs, and the membrane is reset. ϑ is used as the output signal; membrane potential is an internal state.

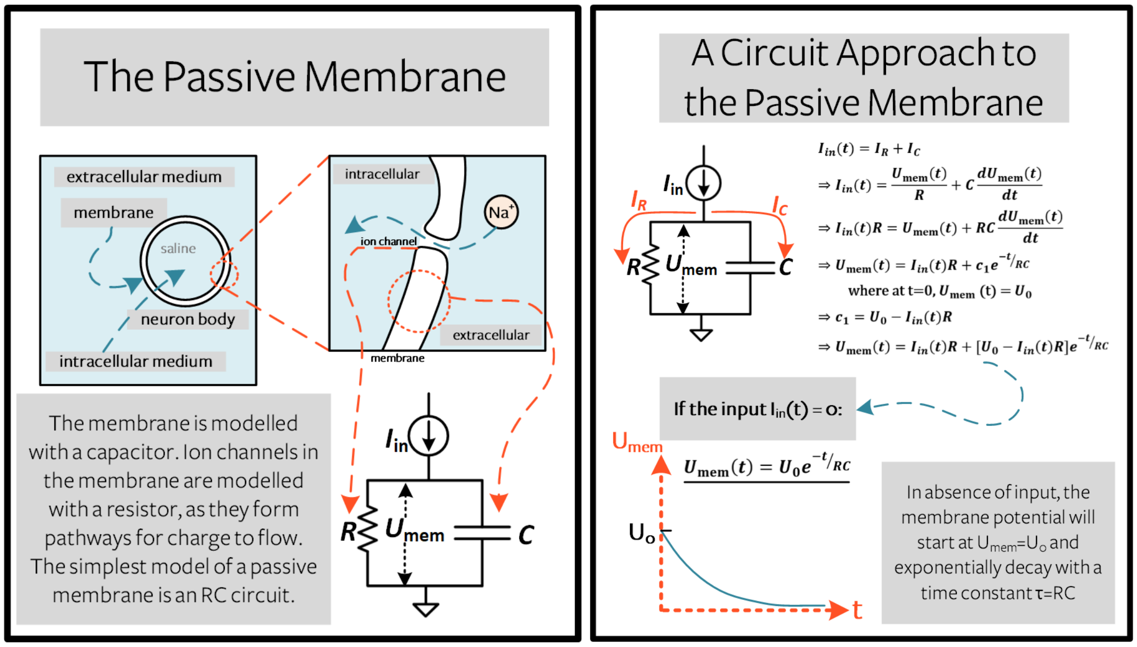
Takes the weighted sum of inputs to produce an output (or activation). No time-varying dynamics or spiking.

Figure 6: Neuron Model Trade off := Bio-realism <-> Utility

Artificial Neuron[8] - The McCulloch-Pitts neuron was conceived by Warren McCulloch and Walter Pitts in their 1943 paper, "A Logical Calculus of Ideas Immanent in Nervous Activity", and is considered the building block of the first generation NN. This model argues that neurons with a binary threshold activation function are analogous to first order logic tables (ie AND & OR).The artificial neuron model, derived from the McCulloch-Pitts neuron, is the building block of the second generation neural networks defined earlier in this section. Also referred to as the perceptron, this model is considered the foundation of deep learning as it incorporates higher-order logic (i.e. XOR or the multi-layer network).

Hodgkin-Huxley Model[9] - Regarded as one of the great achievements of 20th century biophysics, the Hodgkin-Huxley model describes how the action potentials in neurons are initiated and propagated, represented by a set of nonlinear differential equations. This model gives insight into the relationship between the flow of ionic currents across the neuronal cell membrane and the membrane voltage of the cell. Since this mathematical model has a high degree of biophysical accuracy, it is very complex and therefore not particularly useful to deep learning on Von Neumann machines.

Leaky Integrate-And-Fire Neuron Model[10] - We are looking for a good balance between these two extremes, enter the LIF neuron model. The most important distinction of the LIF neuron is that it extends its artificial counterpart to the temporal domain. Which just means that information is encoded within the timing of spikes, rather than in the individual spike. Instead of passing the sum of weighted inputs directly to the activation function, as with the artificial neuron, the LIF model integrates the input over time (with a leak). If the integration exceeds some threshold value then the LIF neuron spikes, thus that threshold is very important to the pattern of action potentials. The different components of the LIF neuron can be modeled as a resistor-capacitor (RC) circuit as seen in Figure 7a. The membrane potential equation can be found using Kirchhoff's current law, which states that the sum of all currents flowing into a node is equal to the sum of all currents flowing out of the node. The derivation for this equation can be seen in detail in Figure 7b below:



The membrane is modelled with a capacitor. Ion channels in the membrane are modelled with a resistor, as they form pathways for charge to flow. The simplest model of a passive membrane is an RC circuit.

$$I_{in}(t) = I_R + I_C$$
$$\Rightarrow I_{in}(t) = \frac{U_{\text{mem}}(t)}{R} + C\frac{dU_{\text{mem}}(t)}{dt}$$
$$\Rightarrow I_{in}(t)R = U_{\text{mem}}(t) + RC\frac{dU_{\text{mem}}(t)}{dt}$$
$$\Rightarrow U_{\text{mem}}(t) = I_{in}(t)R + c_1 e^{-t/RC}$$
where at t=0, $U_{\text{mem}}(t) = U_0$
$$\Rightarrow c_1 = U_0 - I_{in}(t)R$$
$$\Rightarrow U_{\text{mem}}(t) = I_{in}(t)R + [U_0 - I_{in}(t)R]e^{-t/RC}$$

$$U_{\text{mem}}(t) = U_0 e^{-t/RC}$$

In absence of input, the membrane potential will start at $U_{mem}=U_0$ and exponentially decay with a time constant τ=RC

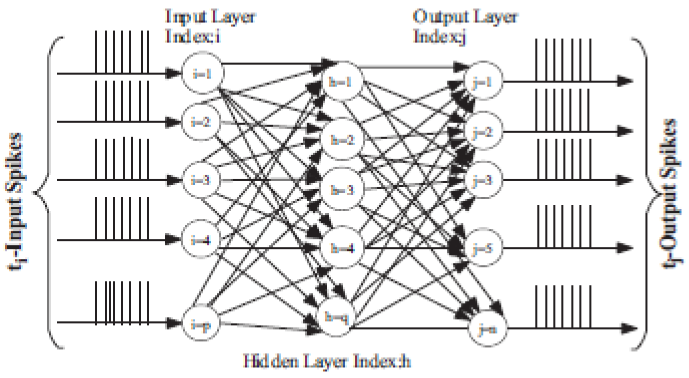
Figure 7: The Passive Membrane - (a) mapping the passive membrane to an electric circuit, and (b) applying circuit analysis to find the voltage membrane equation

This is not meant to be a comprehensive examination of the LIF neuron model, primarily because there is too much information to go over within the given time constraints. I highly advise doing your own research on the concepts I present if you want an exhaustive understanding.

The [Feedforward] Spiking Neural Network

Architectures[11]

The generic feedforward SNN architecture, as illustrated in the next figure, is pretty similar to that of the ANN in terms of layers: both have input, hidden, and output layers. The only difference worth mentioning in this section is that both the input and output of this network are in spike form.



Architecture of a multilayer spiking neural network.

Figure 8: FF-SSN

But neurons in the brain are innately recurrent, which means they have a lot of feedback connections, so we can reformulate the spiking neuron into a discrete, recursive form. This recurrent form is well set up to exploit the developments in training RNNs and sequence based models, as seen in the following figure:

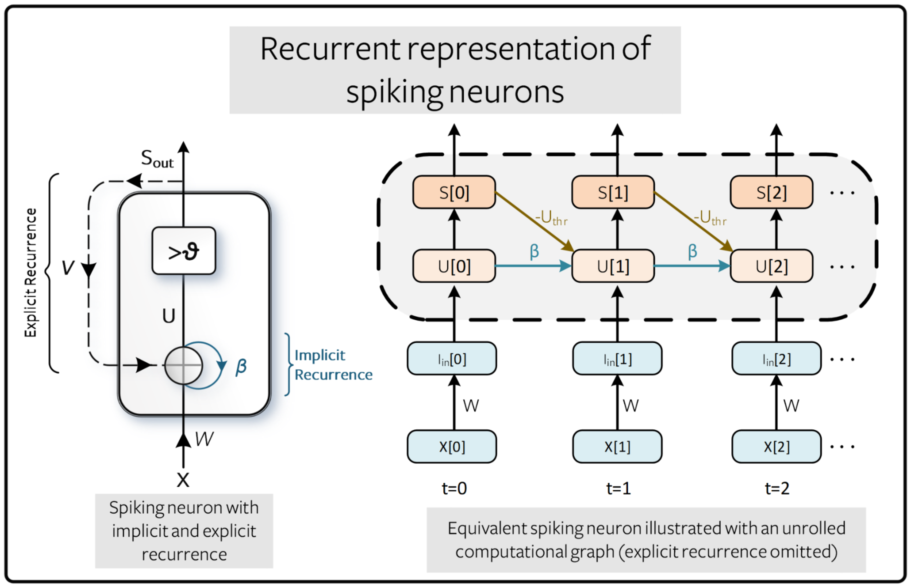

Recurrent representation of spiking neurons
S_out
Explicit Recurrence
V
>ϑ
U
β
Implicit Recurrence
W
X
Spiking neuron with implicit and explicit recurrence
S[0]
S[1]
S[2]
-U_thr
U[0]
U[1]
U[2]
β
I_in[0]
I_in[1]
I_in[2]
W
X[0]
X[1]
X[2]
t=0
t=1
t=2
Equivalent spiking neuron illustrated with an unrolled computational graph (explicit recurrence omitted)

Figure 9: R-SNN

Training[12]

SNNs are difficult to train with backpropagation due to something called the dead neuron problem, which refers to the inherent non-differentiability of spikes. As of now there is no learning algorithm built expressly for SNNs, which is an open problem in computational neuroscience. Although this is probably one of the bigger drawbacks of SNNs, once an SNN learning algorithm is designed, the ANN will be considered outdated. In the meantime there are a few short-term workarounds which allow SNN training, we will focus on the most popular one: surrogate gradient descent. This technique applies some smoothing function (ie sigmoid, fast sigmoid, etc) to the Heaviside step function to ensure its gradient isn't almost always zero. This backalley technique to SNN training is explained visually in the coming figure:

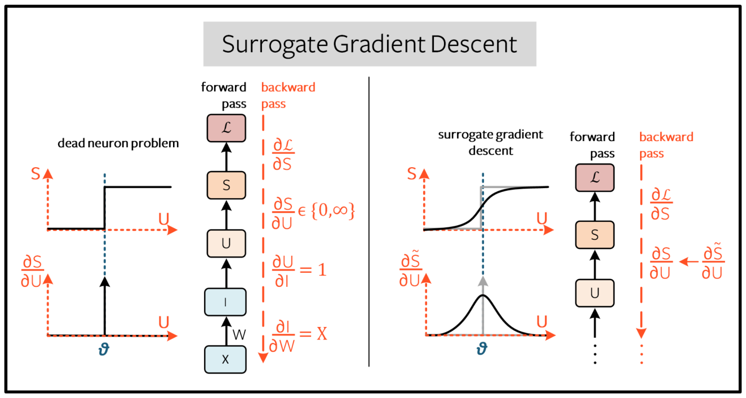

Surrogate Gradient Descent
dead neuron problem
forward pass
backward pass
$\mathcal{L}$
S
U
I
W
X
$\frac{\partial \mathcal{L}}{\partial S}$
$\frac{\partial S}{\partial U} \in \{0,\infty\}$
$\frac{\partial U}{\partial I} = 1$
$\frac{\partial I}{\partial W} = X$
$\frac{\partial S}{\partial U}$
ϑ
surrogate gradient descent
$\frac{\partial \tilde{S}}{\partial U}$

Figure 10: Surrogate Gradient Descent

The figure above only depicts the gradient calculation for a single timestep, but the backpropagation through time (BPTT) algorithm computes the gradient of the loss with respect to all descendants and sums them. The BPTT algorithm is used to train ANNs and therefore SNNs because there's no existing spiking learning algorithm.

SNNs Versus ANNs

The advantage of the biological neuron relative to its artificial sister lies in its ability to disseminate information extremely efficiently. SNNs employ event-driven processing which means only new information is transmitted, as opposed to transmitting at every propagation cycle. Event-driven processing contributes to the firing sparsity and power efficiency of spiking neural networks, making them more potent than their artificial counterparts[13].

Background - Neuromorphic Constraint Optimization[14]

Introduction

Several SNN algorithms have been developed in recent years to solve well-defined computational problems using spike-based temporal information processing. When implemented on neuromorphic architectures, these algorithms promise speed and efficiency gains by exploiting fine-grain parallelism and event-based computation. We will focus on one of these algorithms specifically, the QUBO formulation. The Lava library uses a spiking Boltzmann machine to solve QUBOs.

Architecture[5]

Neurons represent states of binary variables with either a spike or silence, as depicted in Figure 11. When a neuron's internal state exceeds some threshold, it generates an action potential and the state variable switches to one. Once a spike occurs at neuron *i*, the synapses forward the spikes in accordance with the off diagonal weights of the QUBO matrix to other neurons *j*. Positive weights decrease odds that the connected neuron will spike, and vice versa for negative weights. The on-diag weights of the QUBO matrix act as a bias to the neuron's state variable, thereby continuously increasing the probability of a neuron spike.

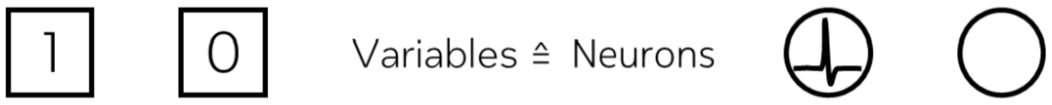


Figure 11: Variable-Neuron Relationship

The bar graph on the left of Figure 12 depicts how the cost function is minimized over a short period of time. It is important to note that the neuron spiking dynamics allow the solver to better persist through local minimums during optimization. The diagram on the right of Figure 12 sheds light on the spiking Boltzmann network, specifically one can observe its size (number of nodes), synaptic connections (curves connecting nodes), and internal neuron states (spike or silence).

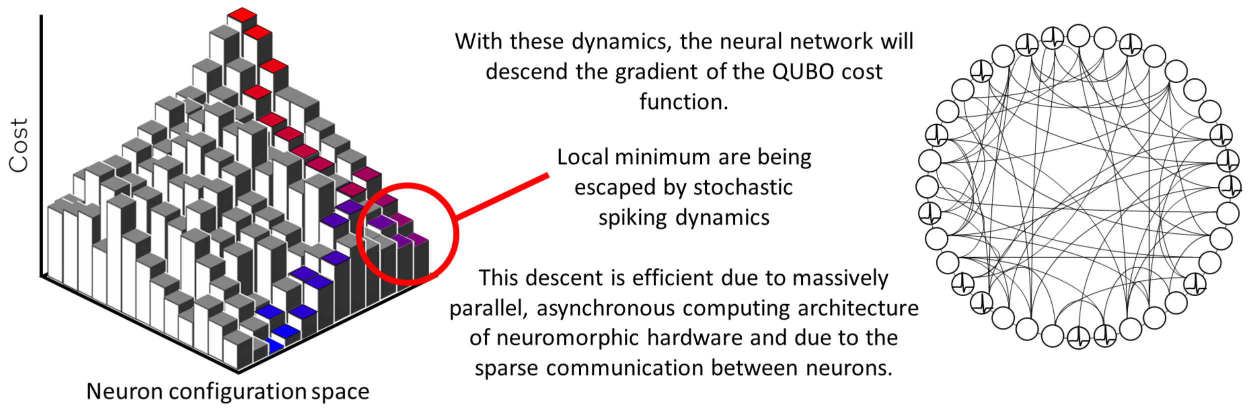

Cost
Neuron configuration space
With these dynamics, the neural network will descend the gradient of the QUBO cost function.
Local minimum are being escaped by stochastic spiking dynamics
This descent is efficient due to massively parallel, asynchronous computing architecture of neuromorphic hardware and due to the sparse communication between neurons.

Figure 12: QUBO Cost Function Minimization

Discussion & Analysis

Walkthrough

- The first step to solving this antenna placement optimization problem is to generate the adjacency matrix of an undirected graph with random connectivity between nodes. The client-specified parameters in this step, connection_prob and num_vertices, define the topology of the graph network. An example graph network, with connection_prob=0.35 and num_vertices=25, is shown in the following figure:

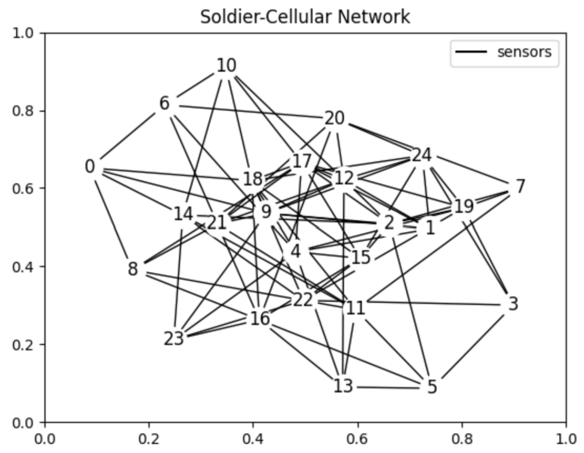
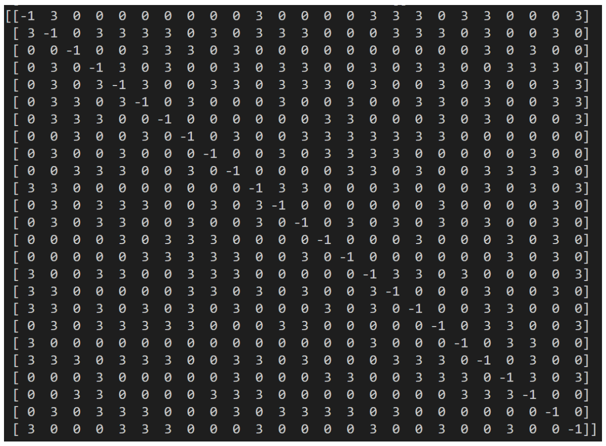


Figure 13: Adjacency Matrix Graph Topology

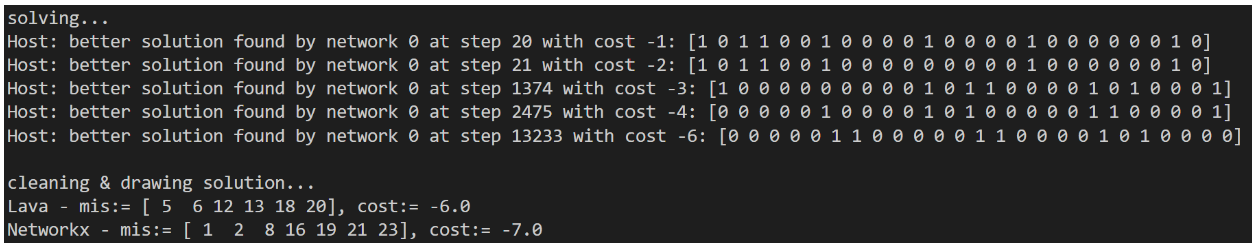
- Next, we map the connection matrix to a QUBO matrix whose solution corresponds to the MIS of the graph. The client-specified parameters in this step, w_diag and w_off, are scalars that are added to and multiplied by the adjacency matrix (the diagonals of the matrix are zero so the w_diag scalar must be added, not multiplied). An example QUBO matrix, with w_diag=1 and w_off=6, is shown in the figure below:



```
[[-1  3  0  0  0  0  0  0  0  0  3  0  0  0  0  3  3  3  0  3  3  0  0  0  3]
 [ 3 -1  0  3  3  3  3  0  3  0  3  3  3  0  0  0  3  3  3  0  3  0  0  3  0]
 [ 0  0 -1  0  0  3  3  3  0  3  0  0  0  0  0  0  0  0  0  0  3  0  3  0  0]
 [ 0  3  0 -1  3  0  3  0  0  3  0  3  3  0  0  3  0  3  3  0  0  3  3  3  0]
 [ 0  3  0  3 -1  3  0  0  3  3  0  3  3  3  0  3  0  0  3  0  3  0  0  3  3]
 [ 0  3  3  0  3 -1  0  3  0  0  0  3  0  0  3  0  0  3  3  0  3  0  0  3  3]
 [ 0  3  3  3  0  0 -1  0  0  0  0  0  0  3  3  0  0  0  3  0  3  0  0  0  3]
 [ 0  0  3  0  0  3  0 -1  0  3  0  0  3  3  3  3  3  3  3  0  0  0  0  0  0]
 [ 0  3  0  0  3  0  0  0 -1  0  0  3  0  3  3  3  3  0  0  0  0  0  3  0  0]
 [ 0  0  3  3  3  0  0  3  0 -1  0  0  0  0  3  3  0  3  0  0  3  3  3  3  0]
 [ 3  3  0  0  0  0  0  0  0  0 -1  3  3  0  0  0  3  0  0  0  3  0  3  0  3]
 [ 0  3  0  3  3  3  0  0  3  0  3 -1  0  0  0  0  0  0  3  0  0  0  0  3  0]
 [ 0  3  0  3  3  0  0  3  0  0  3  0 -1  0  3  0  3  0  3  0  3  0  0  3  0]
 [ 0  0  0  0  3  0  3  3  3  0  0  0  0 -1  0  0  0  3  0  0  0  3  0  3  0]
 [ 0  0  0  0  0  3  3  3  3  3  0  0  3  0 -1  0  0  0  0  0  0  3  0  3  0]
 [ 3  0  0  3  3  0  0  3  3  3  0  0  0  0  0 -1  3  3  0  3  0  0  0  0  3]
 [ 3  3  0  0  0  0  0  3  3  0  3  0  3  0  0  3 -1  0  0  0  3  0  0  3  0]
 [ 3  3  0  3  0  3  0  3  0  3  0  0  0  3  0  3  0 -1  0  0  3  3  0  0  0]
 [ 0  3  0  3  3  3  3  3  0  0  0  3  3  0  0  0  0  0 -1  0  3  3  0  0  3]
 [ 3  0  0  0  0  0  0  0  0  0  0  0  0  0  0  3  0  0  0 -1  0  3  3  0  0]
 [ 3  3  3  0  3  3  3  0  0  3  3  0  3  0  0  0  3  3  3  0 -1  0  3  0  0]
 [ 0  0  0  3  0  0  0  0  0  3  0  0  0  3  3  0  0  3  3  3  0 -1  3  0  3]
 [ 0  0  3  3  0  0  0  0  3  3  3  0  0  0  0  0  0  0  0  3  3  3 -1  0  0]
 [ 0  3  0  3  3  3  0  0  0  3  0  3  3  3  3  0  3  0  0  0  0  0  0 -1  0]
 [ 3  0  0  0  3  3  3  0  0  0  3  0  0  0  0  3  0  0  3  0  0  3  0  0 -1]]
```

Figure 14: QUBO Matrix

- We then pass the QUBO matrix into Lava's solver and iteratively find better and better solutions to our problem. There are several client-specified parameters in this step but the ones worth mentioning, timeout and target_cost, define what I think of as the "optimization vector" which consists of a direction (target_cost) and magnitude (max number of timesteps). The following figure shows the terminal output of my code, partitioned into two parts: (i) the output of the iterated solver, with timeout=20,000 and target_cost=-10, and (ii) the lava-based and networkx-based solution, note that they may not be the same as the Lava solution is subject to the two aforementioned constraints:



```
solving...
Host: better solution found by network 0 at step 20 with cost -1: [1 0 1 1 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0]
Host: better solution found by network 0 at step 21 with cost -2: [1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0]
Host: better solution found by network 0 at step 1374 with cost -3: [1 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 0 1 0 0 0 1]
Host: better solution found by network 0 at step 2475 with cost -4: [0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 1 1 0 0 0 0 1]
Host: better solution found by network 0 at step 13233 with cost -6: [0 0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0 0 1 0 1 0 0 0 0]

cleaning & drawing solution...
Lava - mis:= [ 5  6 12 13 18 20], cost:= -6.0
Networkx - mis:= [ 1  2  8 16 19 21 23], cost:= -7.0
```

Figure 15: Terminal Output Of Solver

- Now we graph the Soldier-Cellular Network for a second time, but this time we color the nodes according to our solution vector. Note that I didn't want to spin my wheels trying to draw these graphs according to Figure 1, where the approximate MIS (I don't know if that network is actually optimal, I found the photo on Google) is denoted with antenna icons. An example colored network is illustrated in Figure 16:

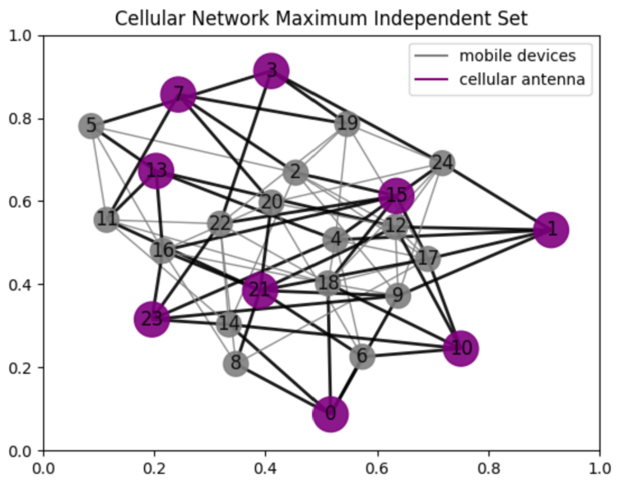


Figure 16: Optimized Soldier-Cellular Network

Benchmarking
It is important to note that the code was run on my local laptop, a cpu-based architecture, not the neuromorphic chip that executes the Lava software, codenamed Loihi 2, which comes from Intel Labs. Thus there isn't an actual jump in computational efficiency when my code is run, which means I can't provide experimental results that compare the computational efficiency of both hardwares. However Intel provides a very promising benchmark in their Lava QUBO tutorial[5] which I refer to as a theoretical result because I personally cannot attest to its validity, stated as follows: "In the previous work, we found that our first generation Loihi 1 chip can solve QUBOs more than 17x faster and more than 670x more energy efficient than the CPU-based solver QBSolv…". Lava is a very young, constantly updated software which is probably why the engineers at Intel Labs have yet to publish an updated tutorial for the second generation neuromorphic chip. But I think it's safe to assume that the corresponding metrics for Loihi 2 only widen the gap with respect to runtime and energy efficiency.

Benchmarking Continued…
The previous paragraph discussed a couple of benchmarks for Intel's first generation Loihi board, whereas this one focuses on the capabilities of a relatively newer neuromorphic hardware out of Intel, named the Kapoho Point. Similar to Intel's previously released neuromorphic boards, this one is named after Hawaii's volcanic geography. It's an ultracompact board around the size of a credit card with four Loihi 2 chips on either side, for a total of eight Loihi 2 boards. This device is able to solve optimization problems with up to eight million variables and with up to 1000x more energy efficiency than a standard CPU-based solver[18].

In another work[19] a graph partitioning QUBO problem was mapped to the TrueNorth neuromorphic hardware, a device that came out of IBM in the past couple years. It is benchmarked, using energy efficiency and graph partition sizes as metrics, against DWave optimization solvers, which rely on a quantum computer. As observed in Table 1 in their paper, the TrueNorth-based solver was able to solve all six of the proposed graph partitioning problems, in contrast to the DWave solver which was only able to solve three. But out of those three, the neuromorphic implementation found the solution using less energy than the DWave solver.

There are very few research papers out there that compare the performance and proficiency of the QUBO solver on neuromorphic chips to that of other architectures, these were the only ones with any relevance to this project.

My Take

While I unfortunately wasn't able to see the huge jump in efficiency on my standard Von-Neumann architecture, I think I benefited from this project in a few other ways. First, I have a much better understanding of mathematical optimization from both a theoretical and programming perspective than I did six months ago. I realized the field is more important than I thought, as a lot of neighboring research disciplines rely on its capabilities. I would like to better understand the role of optimization in the brain given that it naturally aligns with SNN dynamics, but there is almost no research going on in the subfield which makes it kind of difficult. While antenna placement may be useful in a military context, I think there are more exciting QUBO formulations being worked on[16], like the vehicle routing problem (and all its flavors) and the max flow problem[17]. But all in all, I am happy I was able to formulate a problem that connect three topics of interest: what I learn from school (mathematical optimization), what I personally want to work on (computational neuroscience), and what I learn from work (neuromorphic computing applications in battle strategy).

Work Cited

1. https://alphahistory.com/vietnamwar/vietnam-war-soldiers/#:~:text=Difficult%20climate%20and%20terrain,-Australian%20troops%20moving&text=The%20effectiveness%20of%20American%20soldiers,and%20inventiveness%20of%20their%20enemy.
2. https://en.wikipedia.org/wiki/Vietnam_War
3. https://en.wikipedia.org/wiki/Internet_of_Military_Things
4. https://arxiv.org/pdf/2205.08500.pdf
5. https://github.com/lava-nc/lava-optimization/blob/main/tutorials/tutorial_02_solving_qubos.ipynb
6. https://en.wikipedia.org/wiki/Computational_neuroscience
7. https://en.wikipedia.org/wiki/Action_potential
8. https://home.csulb.edu/~cwallis/artificialn/History.htm
9. https://en.wikipedia.org/wiki/Hodgkin%E2%80%93Huxley_model
10. https://en.wikipedia.org/wiki/Biological_neuron_model
11. https://snntorch.readthedocs.io/en/latest/tutorials/tutorial_3.html
12. https://snntorch.readthedocs.io/en/latest/tutorials/tutorial_5.html
13. https://en.wikipedia.org/wiki/Spiking_neural_network
14. https://github.com/lava-nc/lava-optimization
15. https://blog.xa0.de/post/List-of-QUBO-formulations/
16. https://qiskit.org/documentation/optimization/tutorials/07_examples_vehicle_routing.html
17. https://ieeexplore.ieee.org/document/9395703
18. https://www.enterpriseai.news/2022/09/28/intel-labs-launches-loihi-2-based-kapoho-point-board/
19. https://iopscience.iop.org/article/10.1088/2634-4386/ac889c
20.