A Framework for Reinforcement Learning with Spike-Timing-Dependent Synaptic Plasticity in Spiking Neural Networks

12th May, 2020

Abstract

Spiking neural networks are more complex, biologically realistic neural circuit models than the commonly used deep neural networks. Spiking neurons have an internal voltage that slowly decays over time, but with input increasing voltage, have the potential to cause the neuron to fire if the voltage surpasses some firing threshold. A small spiking network is capable of learning sophisticated temporal patterns that may require a sizable deep network to compute. Backpropagation, a popular training algorithm for deep neural networks, is by design, incapable of training spiking neural networks, given their complex dynamics and recurrent nature. Instead, a reinforcement signal can be used to modulate spike-timing-dependent synaptic plasticity suggestions (RL-STDP), a biologically plausible, optimization algorithm that captures local temporal information with global feedback. This is a model-based reinforcement learning algorithm that has been previously demonstrated capable of control tasks. A major blocker for research in this domain is the lack of a malleable simulation framework, flexible enough to support the wide range of existing, and hopefully future experiment designs. In this paper, we present the simulator and surrounding infrastructure we built in an effort to promote research on reinforcement learning with spike-timing-dependent synaptic plasticity in spiking neural networks.

Keywords Spike-timing-dependent Plasticity · Reinforcement Learning · Spiking Neural Network

1 Introduction

Vertebrate brains are extraordinarily complex organizations composed of many simple stimulus filters, called neurons, interconnected by many more signal transmission junctions, called synapses. These neurons guide the animal they reside within towards survival by processing its surrounding environment and enabling adaptive intelligent decisions. Although brains are not created with an understanding of most phenomena they will encounter, over time they will improve the way they compute responses to specific stimuli, according to feedback received from sensory and evaluative systems in the animal. The direct effect of this development applies to pairs of neurons, where the influence one neuron has on another is modulated according to their previous activity and corresponding reinforcement. An innumerable quantity of such pairings exist in the brains of mammals, each adapting hedonistically, in a locally self serving way. The changes that occur across the neural network en-mass are chaotic, and collectively facilitate learning in an emergent way, relying only on relatively simple rules.

In the natural brain, small scale neuron and synapse modulation behavior relies on various underlying chemical and electrical processes. However, in a computer simulation, it is theoretically possible to abstract the brain's fundamental learning mechanics away from the many biological constraints and other complicating factors. Various attempts have been made to accomplish such a feat, namely deep neural networks which have become very popular as of recent. These models are named to represent the network topology they use, called deep feed-forward neural networks, in which neurons are arranged into a series of layers with stimulus being processed sequentially, one layer at a time. In contrast to the distributed learning process of the biological brain, deep networks are commonly trained via backpropagation, a mathematical tool derived from gradient descent.

Deep neural networks have shown capable of learning many meaningful tasks: classifying patterns in images, generating sentences [1], and playing simple video games [2]. With feedback designed for the task, they are able to learn to isolate patterns in images and other static pieces of data, with high accuracy. Although, in order for the model to comprehend the locomotion of videos, sentences and games that are played out over time, serious input pre-processing, rigging of the network topology, or some other manual system engineering is often required. Spiking neural networks, on the contrary, are especially well-suited to associative pattern detection within the temporal dynamics of the environments they operate in. These models are based around the temporal dynamics of spiking neurons, which can be integrated into recurrent networks. Unsupervised training these models is commonly accomplished with spike-timing-dependent synaptic plasticity(STDP), a type of fire together wire together rule well-known to be present in natural brains. On the other hand, designing supervised rules for spiking neural networks has proven difficult, and seen much less success. Altogether much more biologically realistic than deep neural networks, like brains do, spiking neural networks have proven suitable for time sensitive control tasks, for example based on their performance in the Gridworld, Water-Maze, and Cartpole environments with a reward modulated STDP rule(RLSTDP)[3, 4]. This project focuses on learning, the ability of spiking neural networks to discover an optimal set of synaptic weights (i.e., behaviors) for interacting with an environment, based on a feedback signal, a type of model based reinforcement learning.

Ultimately, all thought begins with sensation. In a spiking neural network, information flow begins with a set of designated input neurons conveying sensory information gathered from the environment, relaying it into the network. Whether a stimulus is auditory, visual, or some other interpretation of the environment, it must be encoded into a spike schedule, translating input into a timed sequence of discrete blips. This dictates whether each input neuron will spike or remain quiescent at each time step over the duration of the stimulus processing period. When a neuron spikes, it releases an electrical pulse for a single time step, then proceeds to enter a refractory period under which it cannot fire again for a short time. A neuron may fire towards multiple synapses, thus being a presynaptic connection for multiple synapses that will pipe its output into a single other postsynaptic neuron. Any presynaptic current passed into a synapse is modulated by the synapse's weight before being transmitted onward. Aside from inputs, every neuron has an internal membrane potential that governs when it will fire. When a postsynaptic neuron receives current, it will increase its own membrane potential by the magnitude of the charge. If its membrane potential ever reaches a specified firing threshold, the neuron will spike, and its potential will revert to resting voltage. If the firing threshold is not achieved, the membrane potential will decay steadily and exponentially over time. Thus, a significant amount of charge must come into the neuron under a short period of time if the neuron is to fire. In a well trained network, these system dynamics should create consistent spiking trends, relative to different stimuli patterns. A set of output neurons should be designated to dictate the action chosen by the network. Functionally, these are normal neurons, though their spike trains are tracked. At the end of each processing period, the output spike trains are given to a readout function that will render an action. When applied, the action will alter the state of the environment, which will respond by rewarding or punishing the network accordingly.

Over time, the neural network will tune its synaptic weights such that the agent is compelled to respond to the environment in a way that maximizes earned reward. The more popular Backpropagation algorithm can not comprehend the spiking network's temporally dependent firing patterns. While an un-supervised Hebbian approach can not by design learn such goal-oriented patterns, it's detected patterns are useful for feature extraction, and piggybacking a reinforcement process can enable such learning in spiking networks. Specifically, an asymmetric variant of Hebbian learning called Spike-Timing-Dependent Synaptic Plasticity(STDP) is chosen. This algorithm is applied per synapse, each individually calculating how much influence the presynaptic neuron output has on the firings of the postsynaptic neuron. In the standard STDP method, a positive presynaptic influence will motivate synaptic weight potentiation, while negative presynaptic influence will provoke synaptic weight depression. If the network is to successfully maximize reward, it must act in a way that will accumulate the most utility in the long run. However, STDP alone does not consider the optimality of action choices or goal-directed learning, rendering it incapable of achieving this goal. Researchers have suggested that the modulation of STDP weight-update suggestions by a reinforcement signal can enable updating synaptic weights in a manner that can solve this problem [5, 6], known as RL-STDP. This reinforcement factor magnifies STDP suggestions by the intensity of reinforcement signal, and attempts to reduce the association between neurons if punishment is received. In a realistic environment, feedback may not be given at every time step; instead it may be distributed sparsely through time, or only when certain objectives have been reached, for example winning a game. In terms of RL-STDP, the reinforcement signal would have a value of zero for the majority of the time, effectively halting the learning process. To remedy this, an eligibility trace can be used to keep an exponentially decaying moving average of the reward received by the network (RL-STDP-ET) [6]. This allows the learning process to continue despite the difficulties of sparse feedback. These algorithms, RL-STDP and RL-STDP-ET have been analytically proven capable of supporting the emergence of learning through local synaptic changes [7].

As of recent, few papers have been published studying the learning capabilities of the RL-STDP algorithm in spiking neural networks. This is not due to a lack of potential, RL-STDP has been mathematically proven capable of learning in large recurrent networks [7], but the handful of demonstrated non-trivial tasks have only been done in topologically rigged networks [4]. Significant barriers stand in the way of researchers who wish to recreate previously completed experiments or contribute their own explorations. The most noteworthy of which being the lack of a general purpose spiking neural network simulator capable of arbitrary learning rules at each synapse. Frameworks such as PyTorch [8] and Tensorflow [9] are able to serve the wide range of research needs with deep neural networks, but they lack the functionality to simulate the temporal dynamics of spiking neurons and facilitate distributed STDP calculations in recurrent networks. Simulators built specifically for spiking neural networks exist, but each is seemingly built with a lone objective in mind, none of which are well suited for generally flexible learning algorithm design. Just as general purpose frameworks have been created for deep neural networks, a similar code base can be built for spiking neural networks. If developed, such a simulator could make research with RL-STDP in spiking neural networks considerably more accessible and promote discovery in the field.

Both spiking neural networks, and the RL-STDP-ET learning algorithm, have a large number of hyper-parameters that need to be precisely selected for the model to learn effectively. Analytical methods exist to solve for many of these parameters quickly, although they have not been proven to generalize over the wide range of network and learning schemes being

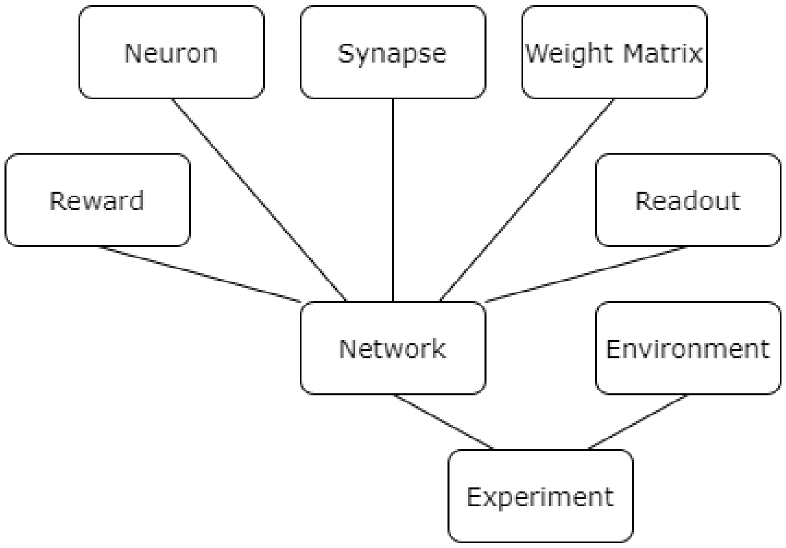

Figure 1: A diagram illustrating the structure of the core simulator.

experimented with. Formulating ways to calculate the settings for novel neural network configurations can be extremely time consuming, given the complex dynamics of the system. Instead, genetic algorithms have been proven to be an effective, automated method to find a useful set of hyper-parameters regardless of neural network or learning rule design, parametrization, or environment [10]. The only requirements are adequate time, processing power, and a model evaluation function with a gradient. It is important that spiking neural network frameworks support evolutionary optimization systems, as well as other types of high level analysis to expedite research endeavors.

In this paper, we explain the custom simulator we developed and the supporting infrastructure we built, in an effort to promote research in the domain of spiking neural networks for reinforcement learning tasks.

2 Spiking Neural Network Simulator

2.1 Design

We set out to build a framework for evaluating the ability of various spiking neural network designs to perform in control learning tasks. Though the core neural network simulator will allow for arbitrary network dynamics and learning rules, the surrounding infrastructure will be constructed specifically for control tasks with feedback. Optimally, little work will be required to retrofit this software package to fit additional use cases. We hope this simulator will empower researchers and engineers to recreate previously completed experiments and to conduct novel explorations with a minimal amount of time spent programming. In an effort to make this simulator as accessible as possible, we chose to use the programming language Python to build it. Python is widely used in machine learning, and has a low barrier to entry for both new and existing programmers. Though the language is not inherently built for high performance or speed, which could become a hindrance, given the large amount of matrix math required for these simulations. To remedy this, we elected to use Numpy's ndarray data structure for all matrices as well as its highly efficient array operations [11]. Altogether, the dream is that this simulator will suffice for the wide range of use cases of RL-STDP in spiking neural networks in an accessible way.

2.2 Implementation

We start with the individual components of the neural network: input, neuron, readout, reward, synapse and weight matrix. Each piece has its own folder containing the corresponding template.py file. Every template file has a single generic class that dictates how the respective can be interacted with by the rest of the network. The first element defined in each class is a variable NECESSARY_KEYS, which is a dictionary pairing the expected configuration keywords with descriptions. This is followed by the constructor function, called __init__ in Python, that contains boilerplate necessary to process the configuration values given. It pulls the keywords requested in NECESSARY_KEYS into member variables prefixed with an underscore, ie self._keyword. The template's constructor may also include basic variable definitions or setup, if they are anticipated to be used by every spiking neural network. The rest of the template class is made up of a series of method layouts, expected to be defined by users in their implementations of the part. These specify function names, parameter lists and the range of return values. Each method is designed to assume the least about how it will be used, allowing for the most freedom of use.

Users may establish new or different part dynamics by creating a file in the specific piece's folder that contains a class that inherits the template. There, particular mechanics can be defined. For speed and conciseness, only one of each piece will be created per simulation. This means the single object should process behavior for a whole set of such items; for example, the neuron class serves as the interface for the whole array of neurons. If configured properly, any piece that respects the laid out template can be made use of by the network object.

The network class is the foundation of the simulator that facilitates interactions between individual components: neuron, synapse, readout and weight matrix. Just the same as any of the individual parts, the network class can be templated in order to modify or add behavior, though it is functionally complete as it stands. Before the network can be used, two extra class variables must be defined: config and _template_parts. config is the configuration specification for the network and all of its parts, a dictionary with all necessary keywords mapped to explicit values. _template_parts is a dictionary that defines which specific instance will be used for each piece type. A list of all necessary parts that need to be set can be found in the network class, named NECESSARY_PARTS. Both class variables can be implemented using the setattr function or by templating the network and defining them in the child class. These values are kept separate from the main class definition in an effort to encourage their distinct configurations for every script that uses the network.

Upon initialization of the network, particular keywords laid out in NECESSARY_KEYS are pulled from the configuration and all pieces are instantiated, with the whole configuration being passed to each one. The network contains a function, tick that defines how stimulus will be processed and responded to. It takes only the current state of the environment as a parameter, which will be passed onto the input generator to produce a set of spike trains. For the duration of the stimulus processing period, the network will 'think'. At the beginning of each time step, the output of each neuron will be compiled into an array based on the input neuron's schedule and all of the other neuron's membrane potentials. This list contains either a zero or the firing magnitude of each neuron, with indices corresponding to specific neurons. Then, the synapse weight matrix is multiplied by these neuron outputs in order to calculate how much current should be given to each neuron. Membrane potentials are updated accordingly. The neuron spike log, which tracks what neurons fired at each point in time, is updated. At the end of each time step, the spike log is given to the synapses to perform the STDP calculation. Finally, when the processing period is over, the output neuron's spike trains will be given to the readout function, which will dictate which action the network selected in response to the given stimulus. This action is returned by the function. All in all, the network class serves as a relatively simple way to aggregate the behavior of the system.

Neural networks may have a purpose, as they are can be tools meant to complete a task. In reinforcement learning, they can play games over and over again, with the intent to achieve as much reward as possible. One environment is an interface for a game that an agent may interact with and receive feedback from. This simulator contains a template, called RL, that outlines the structure for this type of reinforcement learning environment, with a similar role to the templates for network pieces. Every user defined environment is expected to inherit this template and flesh out definitions for CONFIG_DESCRIPTIONS, PRESETS and every unimplemented function laid out in it. CONFIG_DESCRIPTIONS is dictionary pairing configuration keywords and descriptions. PRESETS is a dictionary with preset names corresponding to keyword definitions, potentially allowing for multiple game specifications. Presets can be overridden by the user, and any keywords defined in the class variable config will take precedence over presets. Similar to the config attribute in the network object, config can be defined using the setattr function or from within a child class of the environment. If configured correctly, the network can interact with the environment.

Upon initialization, the environment parses its configuration settings and loads them into the member variable params. When ready to start a new game, the initial state can be read by calling GetInitialState with no parameters. It is important that this function is called whether the information is used or not, as games may use it as a reset function. When the agent has processed this information and is ready to react, the DoAction method may be called with the action and current state as parameters. This updates the environment accordingly then returns the new state and status of the game, whether it is finished or not. DoAction can be called endlessly until the game is finished. Over time, RL will automatically track the state of the environment and which actions were taken in response. After taking an action, the reward function, defined as a piece of the network, may be used to value the choice relative to the initial state. We believe this environment framework will convey all necessary information in a generic-enough manner, allowing for the construction of games ranging from simple logic gates to the complex control in the cart pole task.

The amount of code required to handle the network-environment interactions is small, less than twenty lines. However, the network and environment class have little built in variable tracking and logging. Such functionality adds a significant amount of boilerplate code that would have to be repeated in every script meant to study the network. In general, repeated code should be avoided, because all instances must be updated to suit every change and in this context, it weakens the reproducibility of experiments. Thus, we built an experiment object that takes a network and environment as parameters then can execute a simulation while tracking and logging relevant variables. An experiment can be run and rerun with full, automatic resets in between. This reduces the setup burden for users, alleviating the need to custom write run functionality. It can also be used programmatically for higher level analysis, such as running a series of experiments to see how the network responds to parameter changes or performing an evolutionary search for system hyper-parameters. Because each experiment is isolated, multiple instances can be run in parallel. This experiment object will allow user time to be better spent customizing network and environment functionality, and it will make the construction of higher level analysis tools easier.

2.3 Validation

The simulator depends on the ability of a large number of components to act and interact as they were designed to. Ensuring that each piece can dependably process and respond to the full range of parameter combinations can become a large task if done manually. Instead, this validation process has been automated with unit tests. A test case exists for each core piece of the simulator using the Python unittest framework. Each test case contains multiple methods corresponding to methods or attributes in the particular piece being tested. These methods are meant to ensure all aspects of the specific feature are respected across a wide range of conditions. Most test cases have a run_all_types method that will set up all its methods to automatically run on all instances of the specific piece. The runall.sh script will start all test cases. Any failed tests will show up in the terminal with the test name, piece instance name, and reason for failure. Existing tests are structured to serve as a quick and informative way to validate the structure, robustness and some simple dynamics of the network components. This pre-made testing suite can save users a significant amount of time having to manually debug or custom write tests for each instance.

Requiring users to algorithmically validate every set of mechanics for a particular piece is at best, burdensome. Many of the network components have complex dynamics that will often render unit tests more convoluted than the object being tested. As long as all implementations of a given piece have been proven to respect their template, it is possible to visualize their responses to stimulus in order to confirm they behave as expected. Thus, we built a set of visualizations in Jupyter Notebooks according to the needs of our experiments. Though these will not necessarily work for every case, they are designed to scale. One can simply determine which instance of the piece should be used, set the global configuration variable accordingly, and run the notebook. Visualizations serve as a powerful tool to ensure individual components of the network operate as they were designed to.

Even if every piece of the spiking network functions perfectly in isolation, it is still important to validate the system as a whole. This is the best way to test the communication between parts, and is arguably the most important step. To accomplish this, we implemented two experiments from the paper, "Reinforcement Learning Through Modulation of Spike-Timing-Dependent Synaptic Plasticity" [6] in our simulator. This paper derives two biologically plausible reinforcement modulated spike-timing-dependent synaptic plasticity rules, one with (RM-STDP-ET) and one without (RM-STDP) an eligibility trace. These learning rules serve as a baseline or are very similar to the algorithms used in many successive papers [4, 12]. Likewise, the STDP algorithm we used is very similar to the RM-STDP-ET rule detailed in this paper. [6] also provides straightforward experiments to demonstrate the power of the given algorithms.

We recreated the XOR gate experiments with rate and temporally-coded inputs. An XOR gate, which stands for exclusive-OR gate, takes in two binary (true or false) values and responds with a single binary value of its own. The response of the gate is calculated as follows, if neither or both of the inputs are true, it will return false, if a single input is true, it will return true. For each exercise, a single hidden layer, fully connected feed-forward network with a single output neuron was used. The input-hidden layer composed of 50% excitatory neurons, with $spike\_output = firing\_magnitude$, and 50% inhibitory neurons, with $spike\_output = -firing\_magnitude$. The hidden-output layer consisted exclusively of excitatory neurons. Although only a few spiking neurons (5-10) are required to build a XOR gate, the rate-coded experiment uses 121, and the temporally-coded experiment uses 23 neurons. For robust learning, it is important to have large groups of neurons learn the same task. [13] demonstrated that an aggregate of neural responses is superior to single neuron performance when learning by RL-STDP, and that performance increases with the number of neurons.

The paper, [6] declares that they generate initial synaptic weights over the full range of possibilities, 0-5, though our network had diminishing success with initial values outside of 0-0.2. The network processed each stimulus for 500 time steps without resets in between. The STDP rule only applied to pre- and post-synaptic spikes within a window of 20 time steps of each other. Similarly, neuron membrane potentials took 20 time steps to fully decay while the eligibility trace took 25 time steps. Unlike the paper, our network could not successfully learn without random neuron fires, which could potentially create an "exploration" phenomena. At every time step, 15% of neurons would randomly fire, excluding inputs and those in a refractory period. For computational efficiency, instead of keeping a trace of received reinforcement, we tracked STDP suggestions, and only applied them to the weight matrix when a reward or punishment was given. Our learning algorithms are isometric in terms of the information used.

The state of the environment always consisted of two random binary values, selected anew at every game step. The action chosen by the network was never directly considered. Reward was distributed upon every output neuron spike, +1 if the expected XOR gate response to the state was true and -1 if the expected value was false. For the rate-coded XOR problem, input neurons were split into two groups, each corresponding to a single input. If the input was false, the respective group would not fire. Otherwise, if the input was true, the neurons would fire according to a Poisson process of 40Hz or in our case, a roughly equivalent uniform random rate of 0.08. In the temporally-coded XOR problem, each input controlled a single neuron. Prior to training, two spike trains would be randomly generated according to a Poisson process of 100Hz.

Training would be considered successful if both of the mean output rates for the input sequences 0, 0 and 1, 1 were lower than both mean output rates for the sequences 0, 1 and 1, 0, ie $max(mean\_rate(0,0), mean\_rate(1,1)) < min(mean\_rate(0,1), mean\_rate(1,0))$. Under this, [6] with its RL-STDP-ET rule achieved a 98.2% success rate on rate-coded XOR and a 99.5% success rate for temporally-coded XOR over 1000 trials each. We achieved a 100% success rate on rate-coded XOR and a 98.4% success rate for temporally-coded XOR over 1000 trials each. Thus, we claim our simulator has been successfully validated in a holistic sense.

3 Hyper-parameter Search & Optimization

3.1 Design

Many hyper-parameters need to be configured in a spiking neural network with RL-STDP. Thus the space of possible parametrizations is large, while potentially few are useful. Evolutionary computing algorithms, a brand of meta-reinforcement learning, have been demonstrated capable of traversing ginormous spaces to find satisfactory, if not optimal solutions for a plethora of tasks [14]. Thus, we built our own evolutionary computation system within the simulator. In an effort to be as flexible as possible, we structured this into two separate elements: 1) one to manage genotypes and perform necessary calculations with them, including crossover, mutation, and random generation. 2) a second to solve for genotype fitness in arbitrary games.

3.2 Implementation

The first element of our evolutionary computing scheme is the Population, which is responsible for curating genotypes. In this context, a genotype is a dictionary providing values to a set of keywords. The list of necessary keywords is given to the Population through the parameter GENOTYPE_CONSTRAINTS as a dictionary listing keywords with their respective constraints. Constraints are defined using a list, for discrete variables, and a two tuple, (min_value, max_value), for continuous variables. At any point in time, the Population will carry a set of genotypes with a size given by n_agents. This can be a static integer value or a schedule held in a list.

When the Population is initialized, a set of initial genotypes will be generated randomly within the given constraints. After this, a fitness function can be applied to each member of the group via the evaluate method. The fitness function assigns a score to a genotype, with higher scores being more valuable. This is not a method of the Population class, it is instead defined by a MetaRL Environment object which will be discussed later. In this context, the fitness function will bring about most of the evolutionary algorithm's computational complexity. It will be run many times per epoch in the evaluate method, each call should execute multiple network learning trials. We take steps to mitigate this potential surge in run time. Functionality has been made to cache genotype fitness values, the size of which can be determined with the n_storing parameter. This will remove the need to repeatedly score enduring genotypes. Fitness calculations may be parallelized, with the maximum number of Python processes defined by the n_process parameter. This means multiple genotypes may be scored in parallel, though speed enhancements are capped by the number of hardware threads available. At the end of the evaluate method, a checkpoint is created. This is a Python pickle object containing all Population configuration parameters and a dictionary of the active genotypes mapped to their calculated score. The read_population method was created to reload the population from a checkpoint if ever the need be. Ideally this will prevent many scenarios where all progress is lost due to hardware limitations or failure.

After the existing population is evaluated, a new generation may emerge. The update method exists to construct a new populace by order of the survivor, crossover, mutate and random genotype operators. Based on population size, a percentage of the group will be generated by each operator. These percentages are configured by their corresponding parameters: survivor_rate, crossover_rate, mutation_rate and random_rate. The genotype operators used in this simulator were selected to encourage the population to quickly search the solution space, a high selection pressure, while making a strong effort to avoid converging around a single genotype, a long takeover time. Ideally, with parameters set accordingly, the population will perform a breadth first search, biased around high performing genotypes. The survivor operator simply copies the highest fitness genotypes from the previous generation, a survivor_rate of 20% means the highest scoring 20% of the previous generation live on. A complementary parameter max_age may be used to limit the lifespan of genotypes in an effort to prevent population takeover by high scorers. The crossover operator takes two genotypes and produces two offspring. A random keyword is chosen as the splitting point that determines where both parent genotypes are broken in half. Both offspring are forged into existence using a section from each parent. The mutate operator randomly selects a genotype from the original population and replaces the value of a single keyword with a new, random value within the respective constraints. mutate_eligible_pct defines the percent of the population, sorted by fitness, that is eligible to be considered by the mutation operator. The random operator randomly generates a new genotype, selecting a value for each keyword with uniform sampling over possible constraints. All operators may be reconfigured by overriding their respective methods by means of the setattr function or by creating a child class. In some scenarios, it may be desirable to increase the selection pressure of the evolutionary algorithm. This can be accomplished by increasing the survivor_rate, decreasing the random_rate or by setting mutate_eligible_pct to a low value. Altogether, the Population object facilitates the evolution of a set of genotypes over time.

Similar to the reinforcement learning environments, our evolutionary system has its own, simpler version called MetaRL Environments. In the same way, the MetaRL Environment is a template that will not serve its purpose unless fleshed out for a specific game implementation. This object is designed to handle the contextual aspects of the evolutionary process, such as defining or pre-processing the GENOTYPE_CONSTRAINTS, setting up the game and using it to score genotypes. Its primary component is the get_fitness method. This method takes a genotype as a parameter and returns a scalar fitness value. In order for the Population to effectively arrive at a satisfactory or otherwise optimal genotype, fitness values should carry some gradient information where possible. Without fitness, the evolutionary algorithm effectively performs a breadth first search. However, a high fitness value found at one point in the space will directly bias the algorithm to more thoroughly search that region via the crossover and mutation operators, and indirectly bias it via the survivor operator. Gradient information can help further direct the focus of the search, giving it the traction to ascend towards an optimum and decreasing the time to find a solution. This MetaRL Environment class is kept very generic, it can support a wide range of games from classic evolutionary computing benchmarks like the Meta N Queens Problem up to finding solution sets for a neural network.

A pre-configured MetaRL environment for finding solutions to and optimizing spiking neural network setups has been built, aptly named EvolveNetwork. A network and RL environment are required upon initialization, though their class variables, config and _template_parts may be left undefined. The MetaRL environment also requires definitions for GENOTYPE_CONSTRAINTS and a similar value STATIC_CONFIG. The get_fitness method will instantiate the network and RL environments with configurations based on STATIC_CONFIG and the given genotype, with the genotype holding priority. These objects will be used to create an experiment as defined in section 2. Multiple trials of this experiment will be run, the exact number given by the environment parameter n_reruns, which has a default value of 5. Fitness is calculated as the average accuracy over all trials. A log file will be created that aggregates meaningful data from the simulations. EvolveNetwork should provide a solid baseline for meta optimizing neural networks with our simulator.

In order to run the evolutionary computing system as it has been laid out, a user should first define all necessary configuration including GENOTYPE_CONSTRAINTS and other MetaRL environment parameters, as well as the Population's settings. Upon initialization, the population will automatically generate the first round of random genotypes. Then, until a satisfactory solution is found or the population converges, the population's fitness should be calculated with Population.get_fitness and used to produce a new generation with Population.update repeatedly.

3.3 Validation

It is important to have an objective measure of the capability the evolutionary algorithm has to find solutions. For this, we built an environment for the 8 Queens problem, MetaNQueens. This game in particular was chosen because of its quick fitness calculation and very large search space containing few solutions. The game is played by placing N Queens, 8 in our benchmark, on an 8x8 chess board without any others lying in the same horizontal, vertical or diagonal lines. Genotypes are defined by providing x and y coordinates, each between 1 and 8, for each queen. Fitness is calculated by subtracting the number of collisions from 28, if no collisions are found the genotype is considered a solution. Using this environment, we set up an evolutionary benchmark. The population was evaluated on the MetaNQueens environment with 8 queens over 1000 trials, tracking the number of epochs until a solution is found. Each simulation has a maximum of 3000 epochs with 100 agents per. The algorithm was able to successfully find a solution 97.5% of the time, and on average, it was finished within 465 epochs. We considered this a successful validation of the base functionality of the evolutionary algorithm, that it is capable of effectively searching in large spaces.

Although the functionality of the evolutionary system has been validated with simple tasks and unit tests, see section 2, genetic operator and parameter choices still need to be assessed on the prime objective, a spiking neural network simulation. It's possible that the 8 Queens test is sufficient, but it is difficult to determine how similar the search space of that game is to that of a neural network. Thus for the sake of being thorough, we attempted to validate the evolutionary system on the EvolveNetwork environment. Previously while trying to recreate the rate-coded XOR experiments from [6], we were not able to successfully solve the problem without random fires and with weights generated over the whole range, as the original paper describes. Ideally, the MetaRL should find the flaw in our parametrization. In initial trials, an optimal genotype was found within the first epoch, solely out of random agents. To counteract this, we disabled random genotypes, shrunk the number of agents in the population and increased the size of the search space exorbitantly. The final configuration as follows, n_agents: 64, n_reruns: 5, survivor_rate: 10%, mutation_rate: 30%, crossover_rate: 50%, max_age: 5, mutate_eligible_pct: 50%. The fitness function was the accuracy if a rate from state 0, 0 or 1, 1 was less than the minimum average rate from 0, 1 and 1, 0, and if a rate from state 0, 1 or 1, 0 was greater than the maximum average rate from 0, 0 and 1, 1 are considered correct answers, ie $fitness = mean([(rate < min(mean\_rate(0,1), mean\_rate(1,0)))\ if\ state\ in\ [(0,0),(1,1)]\ else\ (rate > min(mean\_rate(0,0), mean\_rate(1,1)))\ for\ state,\ rate\ in\ responses])$. Effectively, this measures how distinguished the high and low output rates are. In ten epochs, the system was able to find a genotype with a fitness greater than 90%. Although the results were not crisp values as one might analytically generate, they are intuitive and can easily be cleaned up. Thus, we successfully validated that the evolutionary system will suffice the needs of the simulator.

4 Discussion

The framework detailed in this paper was built in an effort to make experimentation and novel research on reinforcement learning in spiking neural networks accessible. It is flexible enough to fit many potential objectives, allowing for the selective editing of any and all system components. As a whole, the framework consists of a generic STDP based spiking neural network simulator, a reinforcement learning environment scheme and corresponding analysis infrastructure. Optimistically, this project will evolve to accommodate a wider range of spiking neural network research in due time. For now, this package aims to save interested people a significant amount of time, allowing them to escape the need to build a custom simulator or attempt to retrofit a more rigid package

Spiking neural networks have been demonstrated capable of learning many control tasks with the help of a reinforcement signal. The existing literature details a plethora of network setups, learning mechanics and games. It is important that our simulator can accommodate the full spectrum of existing works and much more. The core spiking neural network simulator consists of interchangeable, customizable building blocks, that give users the ability to create arbitrary network dynamics and STDP rules. Similarly, the environment scheme allows for the construction of custom games ranging from simple logic gates to complex video games. All boilerplate code necessary to facilitate the interactions between these systems has been wrapped into the experiment object. Simulations can be run, rerun and logged simply by passing a set of configuration parameters into the experiment. This interface allows for the creation of high level analysis systems that are as independent as possible from the core simulator. Unit tests have been set up to quickly validate the structure of any existing or user defined piece. On top of this, visual analysis tools have been created to ensure the specific behavior of any component is as anticipated. The simulator as a whole has been substantiated by recreating relevant reinforcement learning experiments. As is, this simulator can be used to train a wide range of spiking neural network designs on many control tasks.

Evolutionary algorithms have been proven capable of finding solutions in very large search spaces. In this domain, analytical methods exist to derive hyper-parameters for many network setups, though these strategies do not necessarily generalize to all spiking neural networks.

This simulator provides a ready to use evolutionary system that will bypass the time barrier to calculate values for hyper-parameters and can help generate insights as to what sorts of configurations are useful. The population object provides a way to manage the evolution of the genotype pool, with easy to modify genetic operators. The MetaRL environment setup allows for users to perform arbitrary fitness calculations on any game. The baseline spiking neural network MetaRL environment makes use of the experiment object as a generic interface to the core of the simulator. This provides easy access to a plethora of already calculated and logged information that a potential user could use to score the network. The evolutionary system has been verified to consistently find solutions on the classic MetaRL benchmark, the 8 queens problem as well as a difficult task in spiking neural network parameter discovery. Ideally, this tool will allow users to run their own simulations quickly, not having to manually configure all its settings.

5 Acknowledgements

I would like to thank Dr. Taylor for his pertinent resources and insight that proved necessary to successfully build this simulator.

References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. Nature, 521(7553):436–444, 2015.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. ArXiv, abs/1312.5602, 2013.

[3] Wiebke Potjans, Abigail Morrison, and Markus Diesmann. A spiking neural network model of an actor-critic learning agent. Neural computation, 21(2):301–339, 2009.

[4] Nicolas Fremaux, Henning Sprekeler, and Wulfram Gerstner. Reinforcement learning using a continuous time actor-critic framework with spiking neurons. PLoS Computational Biology, 9(4):e1003024, April 2013.

[5] H Sebastian Seung. Learning in spiking neural networks by reinforcement of stochastic synaptic transmission. Neuron, 40(6):1063–1073, 2003.

[6] Razvan V. Florian. Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. Neural Computation, 19(6):1468–1502, 2007.

[7] Robert Legenstein, Dejan Pecevski, and Wolfgang Maass. A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback. PLoS Comput Biol, 4(10):e1000180, 2008.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.

[9] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 265–283, 2016.

[10] Alexander Russell, Garrick Orchard, Yi Dong, Ştefan Mihalaş, Ernst Niebur, Jonathan Tapson, and Ralph Etienne-Cummings. Optimization methods for spiking neurons and networks. IEEE, 21, 2010.

[11] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–.

[12] Eleni Vasilaki, Nicolas Frémaux, Robert Urbanczik, Walter Senn, and Wulfram Gerstner. Spike-based reinforcement learning in continuous state and action space: when policy gradient methods fail. PLoS Comput Biol, 5(12):e1000586–e1000586, 2009.

[13] Robert Urbanczik and Walter Senn. Reinforcement learning in populations of spiking neurons. Nature neuroscience, 12(3):250–252, 2009.

[14] A.E. Eiben and J.E. Smith. Introduction to Evolutionary Computing. Springer, 2015.