Exploring Spiking Neural Networks in Single and Multi-agent RL Methods

M Saravanan
Ericsson Research
Ericsson India Global Services Pvt. Ltd
Chennai, India
m.saravanan@ericsson.com

P Satheesh Kumar
Ericsson Research
Ericsson India Global Services Pvt. Ltd
Chennai, India
perepu.satheesh.kumar@ericsson.com

Kaushik Dey
Ericsson Research
Ericsson India Global Services Pvt. Ltd
Chennai, India
deykaushik@ericsson.com

Sreeja Gaddamidi
Indian Institute of Information Technology
SriCity, Chittoor
Andhra Pradesh, India
sreejagaddamidi99@gmail.com

Adhesh Reghu Kumar
Indian Institute of Information Technology, Design & Manufacturing
Kancheepuram
Chennai, India
coe18b001@iiitdm.ac.in

2021 International Conference on Rebooting Computing (ICRC) | 978-1-6654-2332-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICRC53822.2021.00023

***Abstract*—Reinforcement Learning (RL) techniques can be used effectively to solve a class of optimization problems that require the trajectory of the solution rather than a single-point solution. In deep RL, traditional neural networks are used to model the agent's value function which can be used to obtain the optimal policy. However, traditional neural networks require more data and will take more time to train the network, especially in offline policy training. This paper investigates the effectiveness of implementing deep RL with spiking neural networks (SNNs) in single and multi-agent environments. The advantage of using SNNs is that we require fewer data to obtain good policy and also it is less time-consuming than the traditional neural networks. An important criterion to check for while using SNNs is proper hyperparameter tuning which controls the rate of convergence of SNNs. In this paper, we control the hyperparameter time-step (dt) which affects the spike train generation process in the SNN model. Results on both single-agent and multi-agent environments show that these SNN based models under different time-step (dt) require a lesser number of episodes training to achieve the higher average reward.**

***Index Terms*—RL, single-agent, multi-agent, spiking neural networks, neuromorphic computing**

I. Introduction

Reinforcement Learning involves mapping situations to actions with the goal to maximise a numerical reward signal [1]. A typical RL task usually consists of one or more learners/decision-makers called agents who interact with the system on which we need to perform decisions called environment. The environment provides a reward and a new state based on the decisions usually known as action(s) taken by the agent(s). The goal of RL is to teach the agent an optimal, or nearly optimal policy that maximizes the cumulative reward signal. RL can be used to solve a variety of interesting tasks from teaching robots in warehouses to deliver goods to teaching computers to play real-time strategy games [1]. The simplest implementation of RL is Q-learning where the agents learn the best actions to take in a given state from a Q-table approach [2]. However, the same approach is memory intensive and the complexity increases exponentially with the number of states and actions [2]. To overcome this limitation, in [3] another popular RL algorithm known as deep RL is proposed where neural networks are used in modeling the agent's value function to obtain an optimal policy. However the use of traditional neural networks may require extensive training time to achieve good results, and also have high power consumption. In this paper, we present the implementations of deep RL algorithms on several single and multi-agent environments using an alternate neural network model called the Spiking Neural Network (SNN).

Spiking Neural Networks (SNNs) are Artificial Neural Networks (ANNs) that closely mimic the functioning and structure of the human brain [4]. The typical SNN architecture consists of interconnected computing components called neurons. These neurons use spikes, short electrical pulses, to communicate with each other. SNN considers both the temporal and spatial aspects of the input data. SNNs thus have more computational power compared to the non-spiking networks due to their ability to incorporate temporal dimension in information representation. Implementing SNN models on dedicated neuromorphic hardware enables increased scalability, low power consumption, reduced footprint, and fault-tolerant mechanism. However, a problem with using SNN's directly is the selection of hyperparameter part which will impact the performance of the SNN models. Hence in this work, we propose detailed analysis on how the selection of hyperparameter affects the performance of RL methods when SNN is used instead of ANN in deep RL methods.

In this paper, we experiment on the single-agent environments of CartPole [5] and Mountain Car [6] from OpenAI Gym [7], and on multi-agent environments of Multi-Robot Warehouse (RWARE) [8], a cooperative, partially-observable environment with sparse rewards, and on StarCraft Multi-agent Challenge (SMAC) [9], an environment that simulates the battle scenarios of a real-time strategy game StarCraft II. RL

978-1-6654-2332-8/21/$31.00 ©2021 IEEE
DOI 10.1109/ICRC53822.2021.00023

for both the single-agent environments is implemented using Deep Q-Learning algorithm which uses Neural Networks as function approximators to estimate the Q-values of each state-action pair. In our paper, we use Spiking Neural Network models instead as the function approximators in the Q-Learning algorithm. RL for the RWARE and SMAC environment is implemented using a MARL algorithm known as value decomposition network (VDN). In our paper, we replace these neural network models with the corresponding spiking neural network models. We then compare the performance of these SNN based Deep-RL models with the traditional Deep-RL implementations using Artificial Neural Networks in terms of the mean reward of the system and convergence speed of the reward/loss function.

II. Related Works

In this section, we take a look at some of the works related to our field of study. Spiking Neural Network (SNN) is a growing field of study that has become popular over last few years with advances in hardware [10] and training methods [11], [12] which have helped SNN architectures achieve performance on par with traditional ANN methods for several tasks such as the image classification tasks [13].

RL on the other hand has remained a field of active research for the last several years with successes in solving several real-world tasks [14], [15] and advances in learning algorithms [16], [17]. One such advancement is the use of deep neural networks with RL giving rise to the field of Deep RL (DRL). DRL has been used in developing several successful RL algorithms [18], [19].

Our paper focuses on the intersection between these two fields. Some related research works in this particular sub-domain include [20] where RL algorithm was applied to a stochastic spike response model of spiking neuron to derive learning rules for reward modulated spike-timing dependent learning. [21] compares learning through reward-modulated spike-timing dependent plasticity (STDP) and reinforcement of stochastic synaptic transmission in the general-sum game of the Iterated Prisoner's Dilemma (IPD). In [22], a SNN model is shown to enforce the emergence of synchronized activities in ant colony optimization.

Other works on spike-based reinforcement learning include [23], [24] where a family of reward-modulated synaptic learning rules for spiking neurons is employed on a learning task in continuous space inspired by the Morris Water Maze.

Although there are previous works related to implementing RL using spiking neural networks, none of them focus on the impact of hyper-parameter selection on the performance of the model in different single and multi-agent environments. The contribution of this paper is to present a detailed analysis of the performance of SNN models under different hyper-parameters related to a Spiking Neural Network such as time-step (dt) in different single and multi-agent RL environments. We provide a detailed comparison between the SNN and corresponding ANN models in terms of their average reward and convergence speed. We also provide comparisons of the convergence speed of the SNN models under different hyper-parameters and discuss the impact of such hyper-parameters on the performance of the model.

III. Methods

In this section, we give a brief overview of the RL principles and algorithms (like Q-Learning, Deep Q-Learning, VDN, etc.), and Spiking Neural Networks that have been experimented with in our paper. Let us begin our discussion with Markov Decision Process, a mathematical framework for modelling environments in RL.

A. Markov Decision Process

An environment is said to satisfy *Markov property* if it's next state can be determined from its current state independently of its past states [1]. In other words, the current state successfully captures all relevant information of the past states and is sufficient to determine the future state.

Mathematically,

$$P\{S_{t+1}|S_t\} = P\{S_{t+1}|S_{-\infty}, ..., S_t\} \tag{1}$$

where S_t denotes the state at time *t* and S_{t+1} represents a state of the environment at time *t+1*.

An RL task satisfying the *Markov property* is called *Markov Decision Process*, or *MDP*. If the number of states and actions are finite, then the process is called a *finite Markov Decision Process*.

For a finite MDP, given the state *s* and action *a*, the probability of each possible pair of next state *s'* and reward *r*, is given by:

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\} \tag{2}$$

Further, given a state-action pair the expected reward can be computed:

$$r(s, a) = E[R_{t+1}|S_t = s, A_t = a] = \sum_{r \in R} r \sum_{s \in S} p(s', r|s, a) \tag{3}$$

B. Q-Learning

Now that we have given a mathematical basis for defining an environment in RL, let us take a look at some of the algorithms used. We begin with Q-Learning [2], a model-free RL algorithm, that is widely used in single-agent environments. It is an off-policy method where the target policy is different from the behaviour policy. The value function of Q-Learning is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma max Q(S_{t+1}, a') - Q(S_t, A_t)) \tag{4}$$

where α is the learning rate ($0 \leq \alpha$ 1), R_{t+1} is the reward received and γ is the discounting factor.

Q-Learning computes the Q-value for each state-action pair and continuously updates them using equation (4) until the Q-value converges.

C. Deep Q-Learning

Deep Q-Learning (DQL) [25] combines Neural Networks with Q-Learning. Deep Q-Learning employs neural networks as function approximators to compute and update the Q-values. However, the use of non-linear neural networks as value-approximators is highly unstable. The instability arises from the correlations in the sequence of observations. All states, actions, and rewards are affected by previous states. A technique called *experience replay* [26] stabilizes this. The experience replay stores the prior actions in a buffer and randomly sample them. This removes correlation in the observation sequence.

The above two algorithms explained enables obtaining an optimal policy in the environment. However, when we have multiple agents we need to use a collective approach between agents to design an optimal policy. Next, we explain one such algorithm known as value decomposition network also known as VDN.

D. VDN

Another algorithm used in this paper is Value Decomposition Networks (VDNs) [17], which learns a joint action-value function $Q_{tot}(h, a)$, where h is joint observation history and a is a joint action. Q_{tot} is given as a sum of individual value functions of each agent i, $Q_i(h_i, a_i; \theta_i)$, which are conditioned on individual action-observation histories.

$$Q_{tot}(h, a) = \sum_{i=1}^{N} Q_i(h_i, a_i; \theta_i) \tag{5}$$

Where N is the total number of agents. The loss function in VDN is similar to that in DQN but Q is replaced by Q_{tot}. A benefit of this representation is that each agent performs greedy action selection to its Q_i, which leads to a decentralized policy.

E. Spiking Neural Networks

Having defined the algorithms used in the experiments, we briefly describe Spiking Neural Network architecture, its general working and components. Spiking neural networks are a form of ANNs that mimic the functioning of the brain. SNNs constitute interconnected computing units called neurons which communicate with each other in the form of binary signals called spike trains. Spike, here refers to the 1s in the signal. The functioning of each of the neurons in an SNN model is similar to that of neurons in the brain. Each neuron is initially at resting potential. A typical SNN neuron receives inputs in the form of spike trains. Each spike in the signal increases the internal potential of the neuron. When the potential crosses a given threshold, the neuron emits a spike as the output. The potential of the neuron after a spike decays back to the resting potential. Based on the exact dynamics, there are multiple neuron models available such as *integrate-and-fire (IF)* model, *Leaky integrate-and-Fire (LIF)* model, *Spike Response Model*, etc [27].

Another component of SNN is information encoding [27]. SNN neurons require that the data be encoded in the form of Spatio-temporal pulses (spike-trains). There are several ways to encode the real-valued data into spike trains. Two of the popular methods are: (i) rate coding, which maps an input's intensity to its corresponding input neuron's firing rate, and (ii) temporal coding, which takes into account the exact timing of the spike.

Due to the consideration of the temporal aspects of data, SNN models also include time-related hyper-parameters such as time-interval (T) and time-step (dt). These hyper-parameters control the generation of the spike train outputs of the neurons in the SNN model. The hyper-parameter T determines the length of time for which a particular input is observed and hyper-parameter dt controls the temporal resolution of the simulation.

IV. Experiments and Results

In this section, we first give a brief overview of the different environments implemented in our paper. We then discuss the implementation details of our experiments and the performance metrics used. We then present the results obtained by running the experiments on both the ANN and SNN Model. We also observe the results from the SNN model under different hyper-parameters.

A. RL Environments

In this section, we briefly explain the different single and multi-agent environments that were experimented with in this paper. We discuss the state, reward, and action spaces for each environment.

1) Single-agent: In a single-agent environment, an agent interacts with the environment performing actions. Each action provides the agent with a new state and a reward. The goal of the agent is to maximize the cumulative reward. The environment in a single-agent scenario is stationary concerning to the agent. In this paper, we have experimented on two single-agent environments:

- **Cart Pole** [5] The system consists of a pole attached to a cart by a pivot joint. The cart moves on a friction-less track along the horizontal axis. The goal of the problem is to balance the pole in an upright position by applying bi-directional force (+1 or -1) on the cart. The episode terminates when (1) pole is more than 15 degrees from vertical, or (2) cart position is more than 2.4 units from the center, or (3) episode length is greater than 200.

- **Mountain Car** [6] It consists of a car that rests on a one-dimensional track positioned between two "mountains". The goal of the car is to drive up the mountain on the right. However, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. The car begins each episode stationary at the bottom of the valley (at a position of approx -0.5),

and the episode terminates when (1) it reaches the flag (position > 0.5), or (2) episode length is greater than 200.

TABLE I: State, Action, Reward sets for the single-agent environments

Env.	State Space	Action Space	Reward
CartPole	Angular pos (pole), velocity (pole), linear position (cart), velocity (cart)	-1: left +1: right	+1 for each timestep
Mountain Car	position (car), velocity (car)	drive left drive right do nothing	10, pos. > 0.5 $(1 + pos)^2$, pos > 0.4

2) Multi-agent: The multi-agent environment consists of more than one agent interacting with the environment. The agents could either cooperate or compete with each other to accomplish a particular task. Unlike single-agent environments, the environment is non-stationary in the point of view of each agent. In this paper, we have experimented on the following two multi-agent environments:

- **Mult-agent Robotic Warehouse (RWARE)**
  RWARE [8] simulates a warehouse environment in which robots (agents) move around delivering requested goods. In the simulation, we consider a grid-world warehouse. The goal of the agent (robot) is to locate and deliver requested shelves and return them to empty shelf locations. The environment is considered to be partially observable since the agent can only observe 3×3 grid information surrounding agents and shelves. The RWARE environment can be configured to allow different sizes and the different number of agents. In our experiments, we have implemented a 2-agent RWARE environment. The State, Action, Reward for the 2-agent RWARE environment is as follows:
  - ***Observation Space***:
    self: location, carrying_shelf, direction, on_highway
    sensors: has_agent, direction, has_shelf, shelf_requested
  - ***Action Space***:
    left, right, up, down, no-op
  - ***Reward***:
    1 if shelf is delivered to goal
- **StarCraft Multi-agent Challenge (SMAC)**
  SMAC [9] simulates the battle scenarios of a popular real-time strategy game StarCraft II. SMAC focuses on the decentralized micromanagement challenges. In these challenges, a team of units, each controlled by an independent learning agent observes other units within a fixed radius and takes actions based on these local observations. These agents are trained to solve challenging combat scenarios.
  - ***Observation Space***:
    self: distance, relative x and y positions of the neighboring agents, the health of the agent, a shield of the agent, and unit_type
    Global: Relative x and y positions of all the agents
  - ***Action Space***:
    left, right, up, down, no-op, stop, attack[enemy_id]
  - ***Reward***:
    10 for killing enemy, 200 for winning episode and 0 for losing episode.

B. Implementation Details

In this section, we describe the implementation details of the ANN and SNN based Deep-RL methods used in our paper. Firstly, we discuss the methods employed in building the traditional Deep-RL algorithms (using ANN) for the different environments. Next, we discuss how these models are converted to Spiking Neuron models using ANN-to-SNN converters.

1) ANN Models: In this paper, we have employed Deep Q-Learning with experience replay algorithm to solve the tasks of both the single-agent environments of CartPole and Mountain Car. In this algorithm, we use a neural network model to approximate the Q-value of each possible action given a particular state. We then determine the optimal action as the action with the maximal output Q-value.

The architecture of the ANN models used in the Deep Q-Network of the *CartPole* and *Mountain Car* is a simple 2-layered Linear+ReLU model. Both these networks consider *mean-squared error* as the loss function and also use *Adam* optimizer [28].

For the multi-agent RWARE environment, we use the VDN algorithm. In this algorithm, the value functions of all agents are implemented using neural networks. These networks are updated based on collective rewards rather than individual rewards to improve collaboration. The ANN-based implementation of the algorithm for the multi-agent RWARE environment in our paper remains the same.

For the multi-agent StarCraft environment [9], we have used the PyMARL implementation of reinforcement algorithms like VDN. In this implementation, the agent network is implemented using neural networks.

2) SNN Models: In this paper, the SNN-based Deep-RL models for the different environments have been implemented by converting the corresponding ANN-based models into Spiking Neural Networks using ANN-to-SNN converters such as PyTorchSpiking [29] and KerasSpiking [30].

Frameworks such as PyTorchSpiking and KerasSpiking provide tools to train and run SNN models directly within the PyTorch and Keras frameworks respectively. We have used these frameworks to convert the activation functions in the ANN models into spiking equivalent ones in the SNN models. This introduces the temporal aspects of the data into the model. In the final layer, to get a single-dimensional output, we add a temporal average pool layer that takes the average across the time dimension. Such converted models use spiking activation functions in the forward pass and non-spiking activation functions in the backward pass. These models are trained and evaluated in the same manner as any other PyTorch or Keras

model.
Models for the Cart Pole, RWARE and Star Craft environments are implemented using PyTorch library and hence use PytorchSpiking framework for the ANN-to-SNN conversion, and model for the Mountain Car environment is implemented using Keras library and hence use KerasSpiking framework for the conversion.

C. Experimental Setup

In this section, we describe the setup of the experiments implemented in this paper. All models were run on a local CPU. The configuration of the CPU is as follows: 8 cores, with 2 threads per core. Each processor is a 1.80GHz Intel(R) Core(TM) i7-8550U CPU, and has a 64KB first-level cache and a 256KB second-level cache.

D. Performance Metrics

To compare the performance of the different models, we use mean reward and convergence speed. The formal definition of these metrics are as follows:

- ***Mean Reward***: It is the mean of the rewards earned by the combined system over the entire run of the environment. If r_i is the reward earned by the system at $episode\, i$, then the mean reward is given by:

$$mean\, reward = \frac{1}{N}\sum_{i}^{N} r_i, \tag{6}$$

  where N is the total number of episodes

- ***Convergence Speed***: It is the time taken for policy network or value network to converge. At the start of the experiment i.e. starting of the first episode we initialize the weights of the networks to a random value. Now, when we update the network for every B batch of episodes, the network will get updated. After some B batches, we can see there is no further change in weights of the network or very little change in the weights. We calculate the time taken for different cases of SNN and ANN until the network reaches convergence.

E. ANN vs SNN

In this section, we present a comparison between the ANN and SNN based Deep-RL methods when applied to the given single and multi-agent environments. The mean reward for the different models and environments is tabulated in Table 2. For single-agent environments, we have also plotted reward over episodes for the ANN and SNN models given in Fig. 1 and Fig. 2 respectively. We also compare the rate of convergence between the ANN and SNN models using these plots. For the multi-agent RWARE environment, we observe the policy loss plot (Fig. 3) for each agent to evaluate convergence. For the multi-agent StarCraft environment, we observe mean battles won (test) and mean episode length (test) plots, given in Fig. 4.

Fig. 1 and Fig. 2 shows the comparison between the ANN and SNN model for the single-agent environments of *CartPole* and *Mountain Car* in terms of reward over episodes plot. We observe that the SNN based Deep-RL models perform comparatively better than the corresponding ANN models. This can be observed in terms of both the average reward value and the convergence speed. SNN models achieve faster convergence to a high reward value than ANN models. This shows how SNNs perform comparatively better in single-agent environments than ANNs.

TABLE II: Mean Reward Comparison

Environment	ANN	SNN
CartPole	224.82	283.06
Mountain Car	56.64	59.67
RWARE	13.445	15.167
StarCraft	0.0714	0.14375

For the multi-agent *RWARE* environment, from Fig. 3 and Table. II, we observe that the SNN model performs comparatively better than the corresponding ANN model both in terms of mean reward and convergence speed.

For the multi-agent *StarCraft* environment, from Fig. 4, we can observe that the SNN models achieve a higher mean reward (in terms of battles won) in comparison to the ANN models.

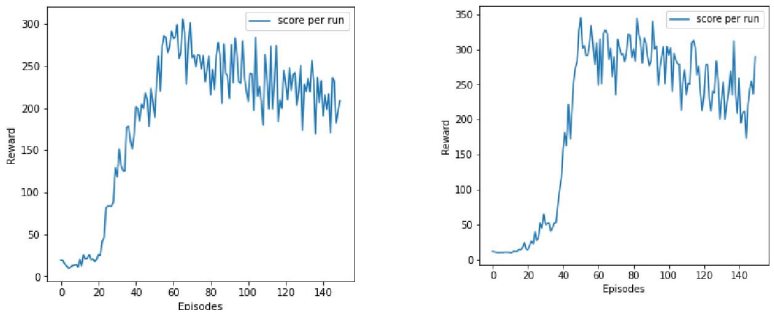
F. Hyper-parameter Tuning

In this section, we explore how the performance of SNN models varies under different hyper-parameters. Spiking neural networks consider temporal aspects of data, and hence hyper-parameters related to time such as time-duration (T), and time-step (dt) are also introduced within the model. Data input to a spiking neuron must also be encoded as signals of 1s and 0s. Hence an additional encoder layer is introduced in the model. In this paper, we control the hyper-parameter time-step (dt), which affects the spike generation process in the SNN model. Results of the experimentation under different time-steps on the different environments are given in Fig. 5, Fig. 6, Fig. 7 and Fig. 8.

Fig. 5 and Fig. 6 show the results for *CartPole* and *Mountain Car* environment respectively under different time-steps. We observe that the *CartPole* performs best at dt=5ms whereas the *Mountain Car* performs best at dt=1ms. Fig. 7 and Fig. 8 plot the results for the multi-agent environments of *RWARE* and *StarCraft* respectively. From these figures, we observe that *RWARE* performs best at dt=10ms and *StarCraft* performs best at dt=1ms.

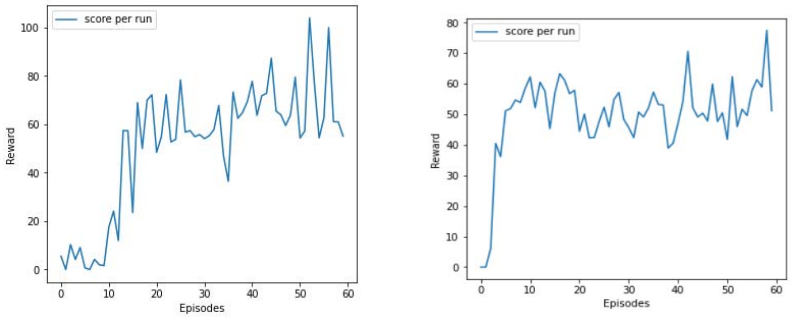
V. Discussion

In this section, we present discussions related to the results of the experiments run in Section IV. We begin with our observations on the results of the experiments comparing ANN and SNN Deep-RL Models. Results show that the SNN based models perform better in comparison to the corresponding ANN models under appropriate hyper-parameters for both single and multi-agent environments. This can be attributed to the fact that SNN models consider and encode data over



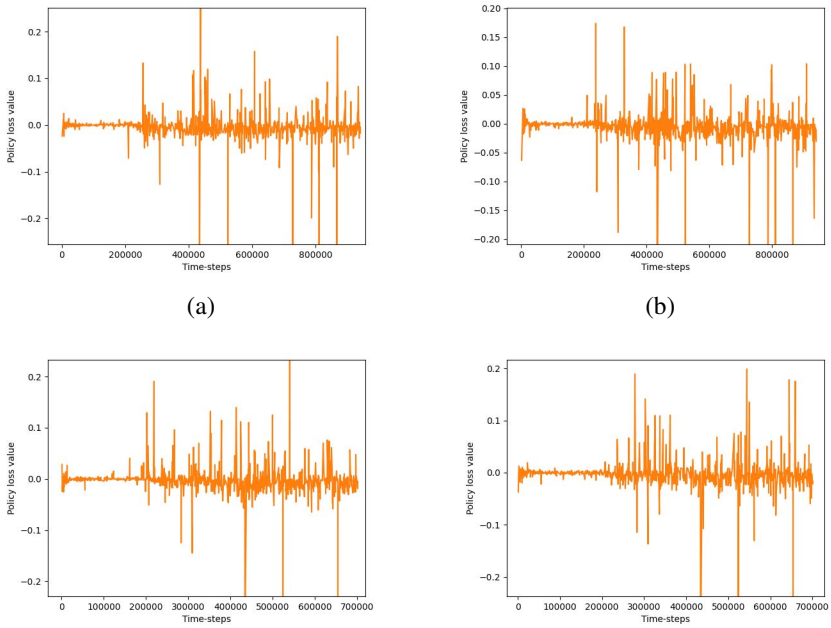
(a) ANN Model (b) SNN Model (dt = 5 ms)

Fig. 1: The reward vs episode plot for ANN and SNN based Deep-RL methods implemented on the *Cart Pole* environment.



(a) ANN Model (b) SNN Model (dt = 1 ms)

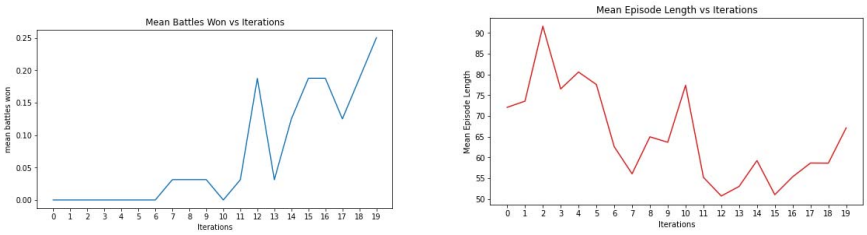
Fig. 2: The reward vs episode plot for ANN and SNN based Deep-RL methods implemented on the *Mountain-Car* environment.



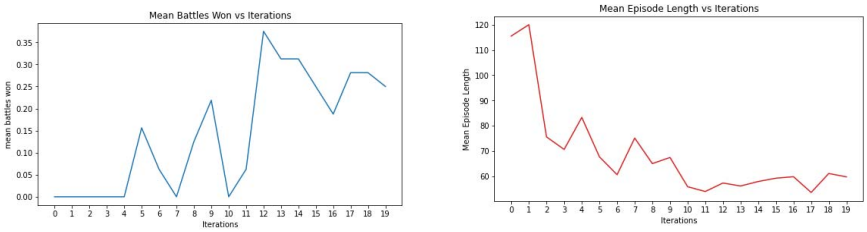
(a) (b)

(c) (d)

Fig. 3: Policy loss plots for the ANN and SNN models for the *RWARE* environment. (a) and (b) give the policy losses for agents 0 and 1 of the ANN model, whereas (c) and (d) are of the SNN model (dt = 10ms).

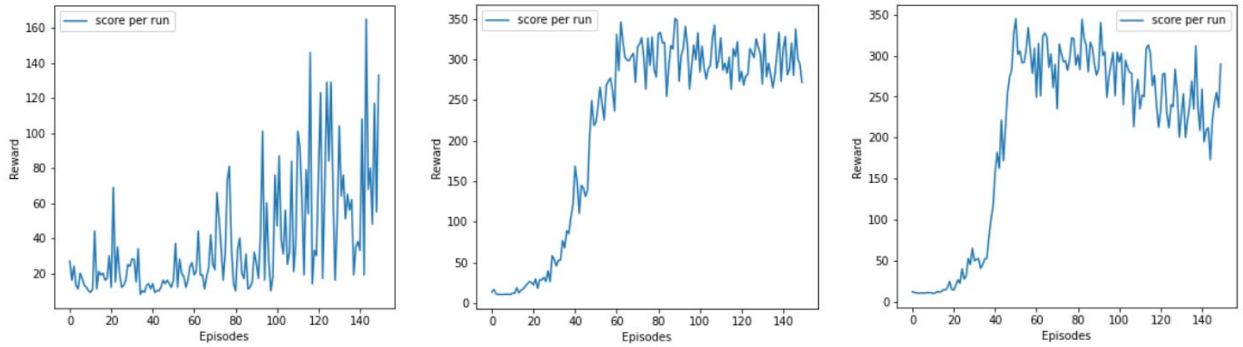


(a) (b)

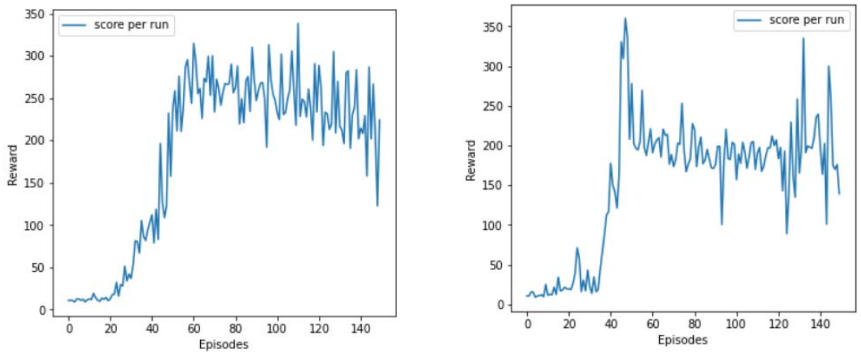


(c) (d)

Fig. 4: Mean Battles Won and Mean Episode Length for the ANN and SNN models for the *Star Craft* environment. (a) and (b) are the plots of the ANN model whereas (c) and (d) are of the SNN Model (dt = 1ms)

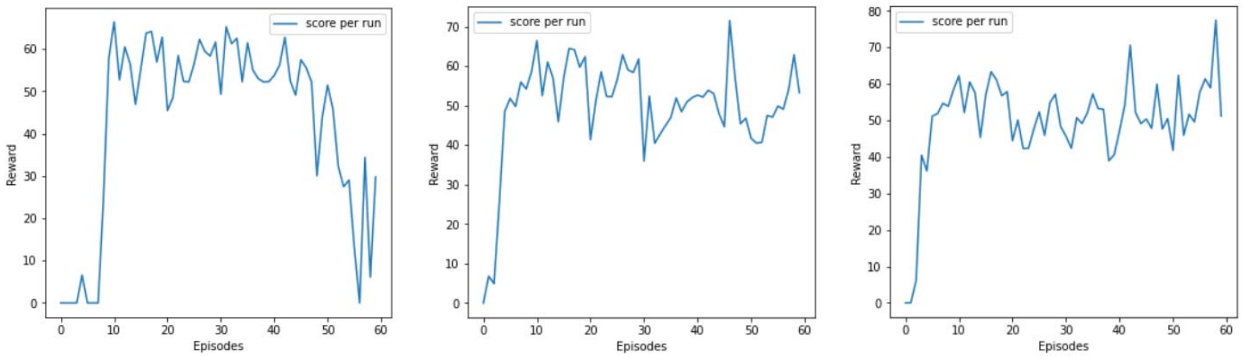


(a) dt = 0.1 ms (b) dt = 1 ms (c) dt = 5 ms

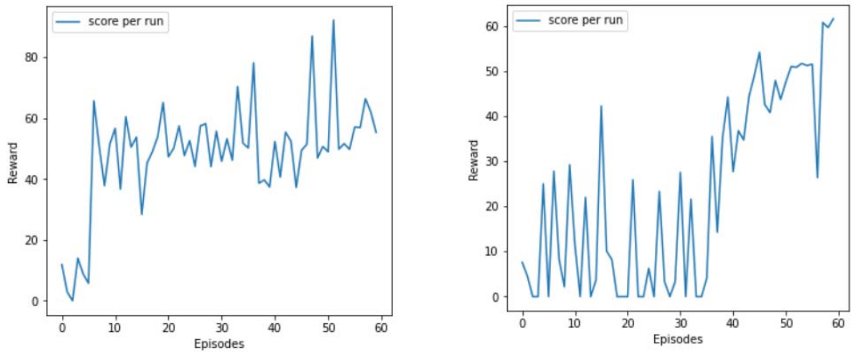


(d) dt = 10 ms (d) dt = 20 ms

Fig. 5: Hyper-parameter tuning on SNN Model implemented on the *CartPole* Environment.

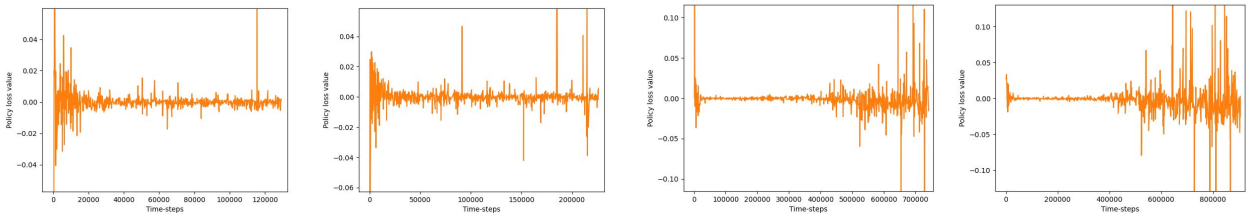


(a) dt = 0.1 ms (b) dt = 0.5 ms (c) dt = 1 ms

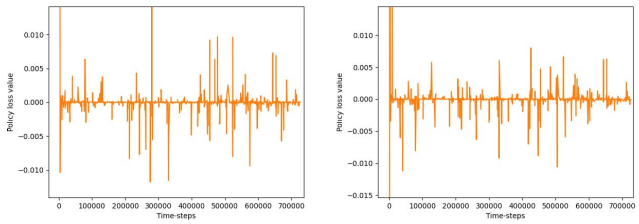


(d) dt = 3 ms (e) dt = 5 ms

Fig. 6: Hyper-parameter tuning on SNN Model implemented on the *Mountain Car* Environment.

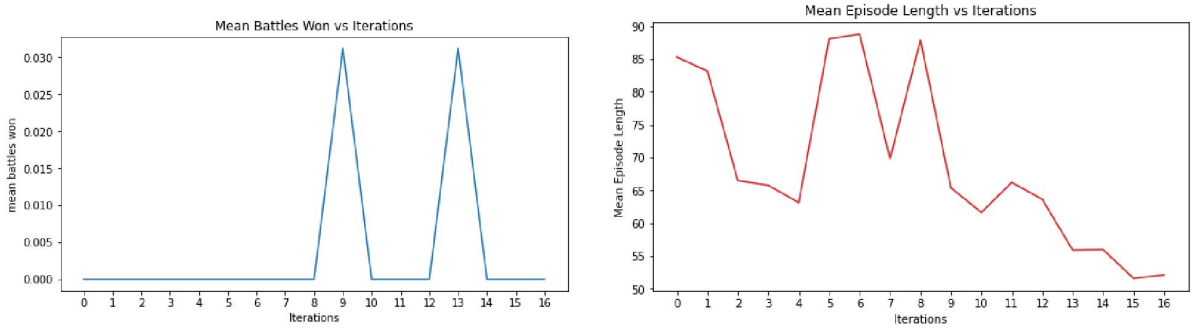


(a) dt = 25 ms (b) dt = 1 ms

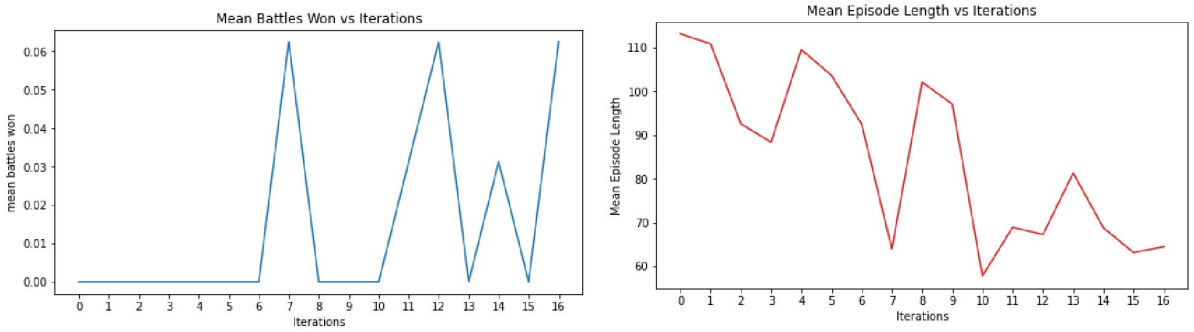


(c) dt = 0.1 ms

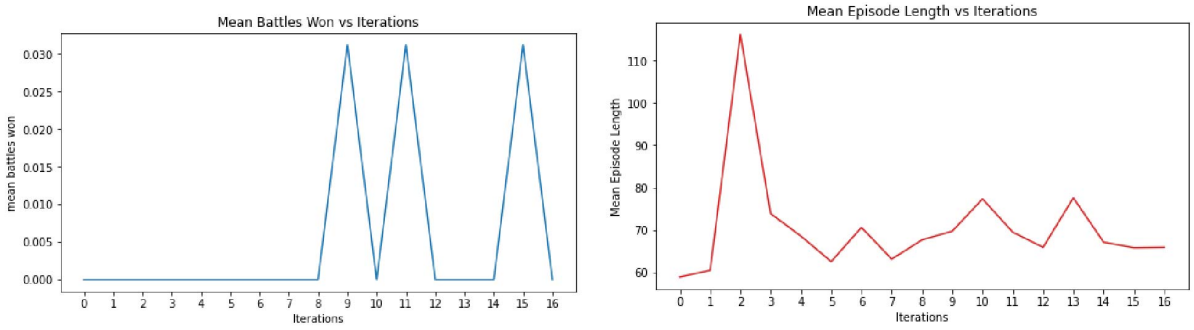
Fig. 7: Hyper-parameter tuning on SNN Model implemented on the *RWARE* Environment.



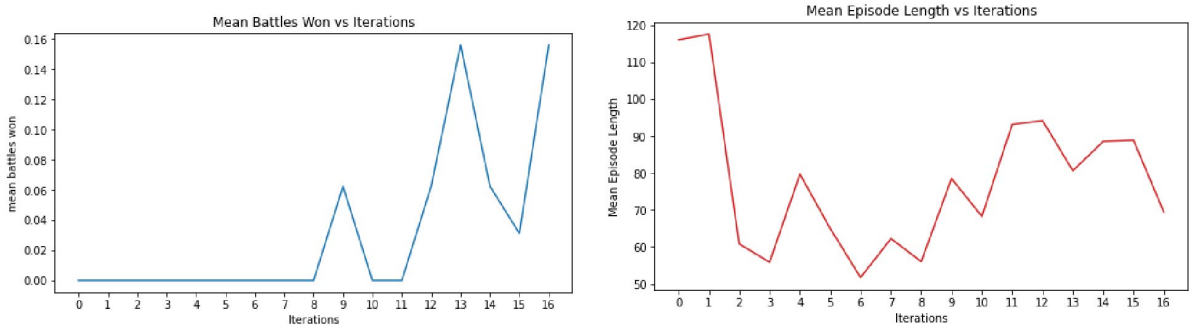
(a) dt = 0.1 ms



(b) dt = 3 ms



(c) dt = 5 ms



(d) dt = 10 ms

Fig. 8: Hyper-parameter tuning on SNN Model implemented on the *Star Craft* Environment.

time. This enables the SNN model to view the same input data as a sequence of spikes over time.

Now, we present our discussion related to the hyper-parameter tuning experiments run on SNN models. From all the experiments run in this regard for the different environments, we can observe that the hyper-parameters related to SNN models such as time-step (dt) play a key role in deciding the performance of the model. We can observe how tuning the time-step (dt) parameter changes the performance of the model. Hence, it becomes important in an SNN architecture to appropriately select the hyper-parameters of our model to get good results.

As for the selection of dt values for each model, we employed a grid-search based method to evaluate our model on a series of dt values, and then determine the optimal dt values that either performed best or helped convey significant information regarding the trend in performance.

For the single-agent *CartPole* environment, we observe how increasing the time-step (dt) improves the model. As dt increases the model converges to a higher reward value. However, the improvement seems to stop after a limit. we can observe that dt=20ms performs worse than dt=10ms.

However, for the single agent *Mountain Car* environment, we observe how increasing the time-step (dt) worsens the model. As dt increases the model converges to a lower reward value. For the multi-agent *RWARE* environment, we can observe that low time-step values do not provide good results. The environment works best at time-steps close to dt=10ms. The performance of the model worsens as dt increases or decreases sufficiently. The multi-agent *Star Craft* environment seems to favor a small time-step close to 1ms. Other sufficiently lower or higher time-steps does not provide good results.

We understood that the environment complexity also plays a key role in the selection of hyperparameter (dt). We can see that the if the environment is complex i.e. if it requires precise and frequent actions, then we need to choose a larger dt value. This is since actions are so frequent, we need to ensure the spikes are measured at a larger time interval to the prevent noisy updates. On the other hand, if the environment is simpler i.e. we don't want to take frequent actions we can have smaller dt values as it won't contribute to the noise. In essence, the choice of dt depends on the time step we used in the environment.

In the *cartpole* environment, we can see that the time step of actions is 0.1 sec. Hence we need to choose larger time step here to remove the effect of the noise. For the case of *Mountainous Car* (time step is 5 sec) and *Starcraft* (time step is 3 sec) environments, we require only smaller dt as they do not require frequent actions. For RWARE environment, we find different effects of dt and hence we want to investigate it further. However, as said it is outside the scope of this paper.

A possible reason why SNN based models outperform ANN based models in terms of the convergence speed and mean reward could be due to the encoding method of SNN models. SNNs encode inputs over time, hence the model can view the same input represented multiple times over time. This provides the model more input representations to learn from.

The plots offer a clear understanding of the tuning effect and how controlling the hyper parameters can reduce the convergence time and can increase the reward which might make the SNN more convenient to apply in different systems.

In our experiments, we also measured the wall-clock time of execution of each model under different hyper-parameters. However, there were no significant difference in the execution times of the models under different *dt* values to warrant a detailed discussion on those lines.

VI. Conclusion

In this paper, we present a study on the performance of Deep RL algorithms implemented using Spiking Neural Networks on different single and multi-agent environments. We show how the SNN based models under appropriate hyper-parameters can perform better than the corresponding ANN implementations. We show the results of the experiments under different hyper-parameters to understand the importance of proper hyper-parameter tuning of the SNN models. From our experiments, we observe that SNN based implementations on single-agent environments provide much faster convergence to higher reward values in comparison to ANN implementations. A similar but limited improvement can also be observed from the SNN experiments on multi-agent environments. Finally, we conclude that the use of Spiking Neural Networks in the implementation of different RL methods improves the performance both in single and multi-agent scenarios. In the future, we plan to explore implementations of SNN-based models on other environments. We also intend to run the SNN implementations on neuromorphic hardware to observe the improvements in terms of time and energy.

References

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[3] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[4] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank. A survey of neuromorphic computing and neural networks in hardware. *CoRR*, abs/1705.06963, 2017.

[5] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[6] Andrew William Moore. Efficient memory-based learning for robot control. 1990.

[7] Gym. https://gym.openai.com/.

[8] Georgios Papoudakis, Filippos Christianos, Lukas Schäfer, and Stefano V Albrecht. Comparative evaluation of multi-agent deep reinforcement learning algorithms. *arXiv preprint arXiv:2006.07869*, 2020.

[9] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder De Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. The starcraft multi-agent challenge. *arXiv preprint arXiv:1902.04043*, 2019.

[10] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, 2019.

[11] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Learning to be efficient: Algorithms for training low-latency, low-compute deep spiking neural networks. In *Proceedings of the 31st annual ACM symposium on applied computing*, pages 293–298, 2016.
[12] Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, Yuan Xie, and Luping Shi. Direct training for spiking neural networks: Faster, larger, better. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1311–1318, 2019.
[13] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in neuroscience*, 13:95, 2019.
[14] N. Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. volume 3, pages 2619 – 2624 Vol.3, 01 2004.
[15] Satinder Singh, Diane Litman, Michael Kearns, and Marilyn Walker. Optimizing dialogue management with reinforcement learning: Experiments with the njfun system. *J. Artif. Int. Res.*, 16(1):105–133, February 2002.
[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 12 2013.
[17] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*, 2017.
[18] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.
[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.
[20] Răzvan V Florian. Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural computation*, 19(6):1468–1502, 2007.
[21] Chris Christodoulou and Aristodemos Cleanthous. Spiking neural networks with different reinforcement learning (rl) schemes in a multiagent setting. *Chinese Journal of Physiology*, 53(6):447–453, 2010.
[22] Sylvain Chevallier, Hélène Paugam-Moisy, and Michèle Sebag. Spikeants, a spiking neuron network modelling the emergence of organization in a complex system. In *NIPS'2010*, pages 379–387, 2010.
[23] 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 Computational Biology*, 5(12):1–17, 12 2009.
[24] Eleni Vasilaki, Robert Urbanczik, Walter Senn, and Wulfram Gerstner. Spike-based reinforcement learning of navigation. *BMC Neuroscience*, 9(1):P72, Jul 2008.
[25] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.
[26] Longxin Lin. Reinforcement learning for robots using neural networks. 1992.
[27] Nikola K. Kasabov. *Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence (Springer Series on Bio- and Neurosystems)*. Springer Publishing Company, Incorporated, 1st edition, 2018.
[28] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.
[29] Pytorchspiking — pytorchspiking 0.1.1.dev0 docs. https://www.nengo.ai/pytorch-spiking/.
[30] Kerasspiking — kerasspiking 0.3.0.dev0 docs. https://www.nengo.ai/keras-spiking/.