Reinforcement Learning with Spiking Neural Networks

1st Yigit Burdurlu
Informatics Department
Technical University of Munich
Munich, Germany
yigit.burdurlu@tum.de

1st Hanady Gebran
Informatics Department
Technical University of Munich
Munich, Germany
hanady.gebran@tum.de

***Abstract*—In this paper, we present two spiking agents that autonomously learn to control a robotic arm with three degrees of freedom. Its objective is to provide joint commands that will move the effector in a desired spatial direction, given the joint configuration of the arm and the target position. The present work is based on supervised learning with spikes using back-propagation for training and using an architecture that starts with an encoding layer, then fully connected linear layers with possibly IF layers between them, and that ends with a decoding layer. The trained spiking neural network was successfully tested on a HoLLiE robotic arm run on the Neurorobotics Platform.**

***Index Terms*—spiking neural network, reinforcement learning, robotic arm, robotic reaching task, neurorobotics platform**

I. Introduction

Reinforcement learning (RL), one of the most popular research areas in the context of machine learning, effectively addresses various problems and challenges in artificial intelligence. It has led to a wide range of exciting advances in various fields, such as robot control, which has become one of the research hot spots in the world. In general, RL methods have made significant progress recently for robotic applications due to their computational power, state-of-the-art algorithms, and large-scale data sets. However, in robot research, data plays an important role in decision making and learning evaluation. The scale of the action and state space increases exponentially with the increase in the number of features for RL tasks, leading to a dimensional disaster. More data and computation are required to explore the states and actions, and thus the computational and energy consumption generated by the learning robot increases exponentially, which is not sustainable development. Therefore, bio-inspired learning is an interesting new perspective that promises to solve this specific challenge in robotic applications as the computation effort is reduced. [1]

Spiking neural networks (SNN) is a bio-inspired approach modelled after information processing, where asynchronous sparse binary signals are communicated and processed in a highly parallelized fashion. SNNs on neuromorphic hardware exhibit advantageous properties of low power consumption and event-driven information processing. This makes them appealing candidates for the cost-effective deployment of deep neural networks. SNNs applied on purely event-based tasks are not only inherently energy-efficient but could potentially harness the rich temporal dynamics of the real world better than frame-based approaches, in which time steps are artificially introduced through sensor or processing components. Thus, SNNs seem particularly suitable for robotic tasks. [2]

The rest of the article is organized as follows, in section II we present related work on the reaching task in NRP with SNN. Section III provides a brief overview of the necessary background and methodology, followed in section IV by a run-through of the experimental setup before reporting our results in section V, and we finally conclude in section VI with a few insights.

II. Related Work

The focus of this study is to tackle the reaching task in the NRP with the SNN. In this task, we ask the robot effector to move between two points in space to reach a target. The problem that arises, however, is that although the task is inherently modeled in Cartesian coordinates, the robot can only control its arm through the motors and perform the task in joint angles space. The computation of the joint coordinates that result in a desired spatial position of the end effector is called inverse kinematics: it is typically a non-trivial task that becomes even more difficult when challenged with many degrees of freedom. In this paper, we train a 6 DoF HoLLiE robotic arm only on 3DoFs, which is a common reduction in most papers on reaching tasks that employ SNN, such as in Carillo et al. [3] working only on 2 DoFs and in Bouganis and Shanahan [4] where the 7 DoF iCub arm is trained on only 4 DoFs.

A straightforward way to move the end effector to its target position can be obtained by considering each joint independently, and incrementing its angle accordingly towards its final value at the target arm configuration; thus we can directly output the joint angles that result in the desired spatial position of the end effector. In this paper, we are taking this approach and considering that giving the final joint angles as the output of the agent to achieve the task is enough; however, this view is not universal; for instance, another popular method [5] consists of not directly computing the joint angles that result in the desired spatial position of the end effector; but

TUM Cloud-Based Machine Learning in Robotics WS21/22

using small steps to the target position and computing the joint velocities at each step to move the end effector in the desired spatial direction.

Different strategies for learning deep SNNs have been developed in recent years: the present work is based on supervised learning with spikes using backpropagation to train deep SNNs. The emphasis in this work is not on biological plausibility. However, biological plausibility is at the core of other approaches using local learning. Again, in Bouganis and Shanahan [4], also addressing the reaching task, a feed-forward network of individual neurons using spike-timing-dependent plasticity (STDP) is architected, making the change in synaptic weights biologically plausible.

III. Background and Methodology

A. Reinforcement Learning and Algorithms

1) RL: A reinforcement learning agent aims to learn the optimal way to perform a task through repeated interactions with its environment [6]. To achieve this, the agent needs to assess the long-term value of the actions it undertakes.

RL algorithms can be categorized into several types, such as model-based algorithms or policy-gradient algorithms. In this paper, we focus on the Actor-Critic (AC) category which estimates the value function by solving $\theta_* = \underset{\theta}{\operatorname{argmin}}(v_\theta(s) - \hat{v}(s))^2$ to obtain a value function as close as possible to the actual value function, and use it to predict future rewards $r(s_t, a_{t,})$. It then takes the gradient of objective $\theta_* = \underset{\theta}{\operatorname{argmax}} E_{\tau\sim p_\theta(\tau)}\left[\sum_t \gamma^t r(s_t, a_t)\right]$ to improve the policy in order for the resulting policy to maximize the expected value of future rewards where the expectation is over the sequence τ of future actions and states, distributed according the probability $p_\theta(\tau)$).

2) Advantage Actor-Critic: A standard policy-based model-free method updates the parameters of policy function. One example of such a method is REINFORCE [12]. In order to reduce the high variance in policy-gradient, one of the methods used is subtracting the cumulative reward by a baseline function. Furthermore, an example of baseline function is advantage value which refers to comparing an action to the general average action [13]. Additionally, the A2C method is a synchronous version of A3C.

3) Twin Delayed DDPG: DDPG can be unstable and heavily reliant on hyperparameters because of the overestimation of Q values by the value network. [14]. The overestimation accumulates through the Bellman equation over timesteps and causes unavoidable errors. What TD3 brings are three essential steps; using two critic networks, delayed updates for the actor-network and action noise regularization. Using two critic networks avoids abnormally high action values. Delayed updates for the actor-network provides more stability on action output. Lastly, the additional noise regularization avoids high variance in target values when updating the critic networks.

B. Reaching task

The reaching task is an essential part of robotic manipulation that requires the end effector to reach the target while satisfying physical constraints. In this paper the reaching task is achieved successfully once the center of the end effector is reasonably close to a cylinder placed at a random position on a table. We can formulate the reaching task as a MDP process with the definitions of state, action, goal, and reward.

State: It contains the effector position (3 dimensions), the cylinder position (3 dimensions), and all joint angles (6 dimensions)

$$s = \left[ee_x, ee_y, ee_z, cyl_x, cyl_y, cyl_z, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\right],$$

Action: We consider the new joints angles as the action. In addition, the calculated final actions are limited by the joint limits. Within our configuration, we only use the first 3 joint angles, thus we set the upper and lower limits of the last 3 joint angles to 0.

$$\begin{aligned} A_1 = \{\ \text{low} &= \left[-\frac{\pi}{2}, -\frac{\pi}{2}, 0, 0, 0, 0\right] \\ \text{high} &= \left[\frac{\pi}{2}, 0, \pi, 0, 0, 0\right]\Big\} \end{aligned}$$

Goal: Based on these joint angles, defining the result of the forward kinematics (end effector position) in Cartesian space as the target can guarantee an attainable target position.

$$\text{goal} = [ee_x, ee_y, ee_z]$$

Reward: The design of the reward function is a critical component of successful RL. We trained implemented algorithms on two types of reward: dense and sparse reward functions, as both are commonly used for reaching tasks. Accordingly, we give the two definitions as follows:

For the dense reward, the reward is given as a function of the Euclidean distance between the end effector and the goal pose. If the position distance $\text{dist}(p)$ is less than the required accuracy ϵ, it is considered a successful action and the payoff is set to $a > 0$. Else, the reward is set to penalize the Euclidean distance between the end-effector and the target position

$$r = \begin{cases} a & \epsilon \leq \text{dist}(p) \\ -\text{dist}(ee,\ \text{goal}\ ) & \epsilon > \text{dist}(ee, goal) \end{cases}$$

For the sparse reward condition, a successful movement is rewarded with $a > 0$, otherwise, a punishment $p_1(< 0)$ is given for energy consumption. In addition, if the robot performs a physically impossible movement, such as crossing the table, it is also punished by $p_2(< 0)$.

$$r = \begin{cases} a & \epsilon \leq \text{dist}(ee, goal) \\ p_1 & \epsilon > \text{dist}(ee, goal) \\ p_2 & \text{constraint violated} \end{cases}$$

C. SNN, encoding and decoding [7]

1) SNN [8]: A neuron in a spiking neural network SNN can be viewed as a neuron in a recurrent neural network RNN: the input being the voltage increment, the hidden state bieng the membrane voltage, and the output being a spike. These spike neurons are Markovian: the output at the current time is

perfectly defined by the input at that time and by the state of the neuron itself.

WE can use three discrete equations −− Charge, Discharge, Reset −− to describe any discrete spiking neuron:

$$
\begin{aligned}
H(t) &= f(V(t-1), X(t)) \\
S(t) &= g\left(H(t) - V_{\text{threshold}}\right) = \Theta\left(H(t) - V_{\text{threshold}}\right) \\
V(t) &= H(t) \cdot (1 - S(t)) + V_{\text{reset}} \cdot S(t)
\end{aligned}
$$

where $V(t)$ is the membrane voltage of the neuron; $X(t)$ is the voltage increment; $H(t)$ is the hidden state of the neuron and $f(V(t-1), X(t))$ is the state update equation of the neuron.

The updating equation will vary for different neurons. For example, for a LIF neuron, the corresponding difference equations are:

$$
\begin{aligned}
\tau_m \frac{\mathrm{d}V(t)}{\mathrm{d}t} &= -\left(V(t) - V_{\text{reset}}\right) + X(t) \\
\tau_m(V(t) - V(t-1)) &= -\left(V(t-1) - V_{\text{reset}}\right) + X(t)
\end{aligned}
$$

The corresponding Charge equation is

$$
f(V(t-1), X(t)) = V(t-1) + \frac{1}{\tau_m}\left(-\left(V(t-1) - V_{res}\right) + X(t)\right)
$$

In the discharge equation, $S(t)$ is a spike triggered by a neuron. $g(x) = \Theta(x)$ is a step function called the spike function. The output of the spiking function is 0 if a spike is not triggered at the current time or 1 when a spike is fired, it is defined as follows:

$$
\Theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}
$$

Reset means the reset process of the voltage: when a spike is fired, the voltage is reset to V_{reset} ; If no spike is fired, the voltage remains unchanged.

The RNN uses differentiable nonlinear step functions, such as the tanh function. However, the SNN spiking function $g(x) = \Theta(x)$ is not differentiable; thus, contrary to RNN, SNN cannot be directly trained by gradient descent and back-propagation. To overcome this issue, we can use a function $\sigma(x)$ that is very similar to $g(x) = \Theta(x)$, but differentiable to replace it.

The core idea of this method is: when forwarding, use $g(x) = \Theta(x)$, the output of the neuron is 0 and 1 , and our network is still an SNN; when back-propagating, the gradient of the surrogate gradient function $g'(x) = \sigma'(x)$ is used to replace the gradient of the spiking function.

We clarify that we have 3 units of time in this work. An episode is a sequence of states, actions and rewards, which ends with a terminal state. An epoch is a forward and a backward passage of the agent. An SNNstep, which we will refer to as timestep, is a unit of time in which we can have up to one spike.

2) Encoding [9] [10]: The two most common approaches to encoding the input of spiking neural networks are both bio-inspired: rate encoding and time encoding, respectively called the ”stateless encoder” and the ”stateful encoder”. Stateless encoder because it has no hidden states within it, and the spikes $spike[t]$ are encoded just from the input data $x[t]$ at time step t, the higher the input, the larger the probability to have a spike ; and stateful encoder because it will encode the input sequence x containing the data of T time steps into a spike at the first time forward, and will output $spike[t\%T]$ at the t-th step call, we get only a single spike for the T time steps and the higher the input, the earlier the spike. Two stateless encoders have been implemented and fully tested, they are explained in IV-B.

3) Decoding [11]: Typically, in deep Q-learning, the neural network acts as the Q-function, whose output must be continuous values. Therefore, the last layer of the SNN cannot directly use the output spikes to represent the Q function as 0 and 1, otherwise it leads to really poor performance. There are several methods to make the outputs of SNNs continuous values and the one used is explained in IV-B.

IV. Experimental setup

A. Simulation Environment and Deployment

The RL algorithms are commonly trained in a simulation environment. In this regard, we used Neurorobotics Platform NRP [15] which provides scalable simulation platform with reproducibility and synchronization for models in a common simulation environment. Addtionally, In its backend, NRP uses Gazebo as the simulator and provides a control interface which benefits from Robot Operating System ROS. From

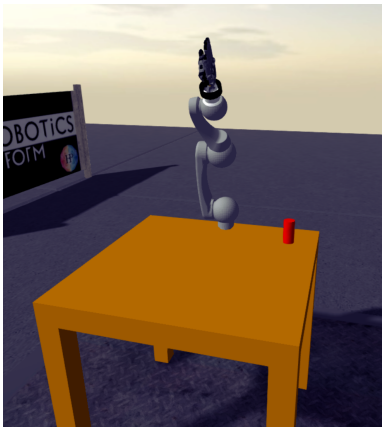

Fig. 1. Reaching task simulation model

variety of simulation models, we used 6 DoF HoLLiE robot arm model running on a NRP docker container. In order to give instructions to NRP simulation, we used Google Remote Procedure Call GRPC system between NRP backend and the application which runs an instance of training.

B. Architecture

All networks architecture follows the same pattern. We start with an encoding layer that transforms the input of size n into

a spike train of size n over T timesteps, this spike train can be seen as a vector of size n where each element is a vector of size T with elements either 0 or 1. The encoding layer is followed by fully connected linear layers that are regular feed forward networks with n neurons for the first linear layer and m neurons for the last one with possibly IF layers in between. Finally, these linear layers are followed by a decoding layer of size m which converts each vector of size T into a single number.

1) Encoding: We focused on 2 different encodings: Poisson encoder and linear+IF Node encoder.

The Poisson encoder is a stateless encoder. It converts the input data x into a spike train of the same shape, which conforms to a Poisson process, i.e., the number of spikes during a certain period follows a Poisson distribution. Consequently, in order to realize Poisson encoding, we normalize the input x element-wise to $[0, 1]$ and we set the firing probability on each dimention to the value of the normalized element x_i. The mean number of spikes during T timesteps is therefore $T \times x[i]$ for the ith element of the output of the encoding layer.

The linear encoder+IF node is more practical because the input does not need to be normalized. Indeed, we first feed the input x over T time steps to a linear layer, this layer is connected to an IF node of the same size; an IF node being a node with the same equation as LIF III-C1 but without the decay term. Since there is a spike as soon as the IF input is greater than 1, the linear layer modifies its weight so as to have an output with values less than 1 to send to the IF nodes. Indeed, for an input greater than or equal to 1, the IF node spikes at each time step and we obtain the vector $\begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}$ and a good encoding must lead to different results for different inputs of the agent. The IF node will then, over T time steps, transform the pseudo-normalized input into a spike train, with a higher spike rate for a higher input value.

2) Decoding: A traditional method is to make the final output of the network the firing rate of each neuron in the output layer. However, using the firing rate as a Q-function don't lead to satisfactory performance because it does not capture enough computational depth: indeed, after T simulation steps, the possible firing rates are [0, 1/T, 2/T, ... 1], which is not sufficient to represent a reliable Q-function. In this paper, we apply a novel scheme for SNNs to produce floating point numbers. Specifically, a LIF neuron's firing threshold is set to infinity, so that it does not fire at all, and we adopt its final membrane potential to represent the Q function.

C. Parameters

In the implementation we are provided with several external parameters, the following are the most relevant and application specific ones.

1) Goal proximity: For robot reaching tasks, the different levels of goal proximity between the robot end effector and the target make the problem more or less difficult. The question is how to choose it: a poorly chosen goal can lead the learning process in the wrong direction and make the training inefficient. Therefore, in this study, the required distance threshold from the target is an important parameter: it should be suitably high to enable the robot to obtain rewards for acquiring the basic skills needed for the task, while being low enough to achieve a satisfactory result. An interesting way to do this is to start with a high threshold and automatically decrease it after reaching a certain accuracy.

2) Rewarding strategy: The agent's goal is to maximize its rewards and it learns by adjusting its policy based on them; thus, this is an important aspect for training, hence we test different a, p1 and p2 for the reward functions described in III-B. We also have an external parameter which is the goal reward threshold to finish model training.

3) Start timesteps: Reinforcement learning algorithms use replay buffers to store experience trajectories when executing a policy in an environment, we therefore have to choose the number of epochs to populate the replay buffer.

4) Max timesteps: The total number of epochs for training.

5) Tries per episode: The number of attempts allowed in an episode to reach the goal.

6) T: The number of timesteps used to encode the input, that is equivalent to the number of timesteps in an epoch.

V. Experiments and Results

The main purpose of the experiments is to see how SNNs perform in reaching task problems with different RL methods in a given environment. Additionally, we used two different base network architectures with a different number of layers to infer whether the size of the network has an importance on the given task. The different encoders at the beginning of the networks provided a variety of differences when they are referred again. the base networks design can be seen in figure 2.

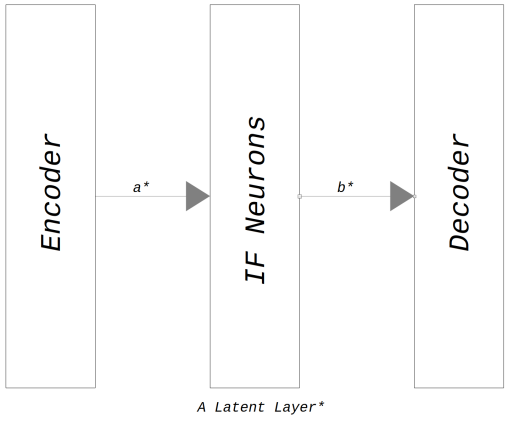

Fig. 2. General network architecture of implementations. (a) corresponds to a fully connected layer of size 15x128 for TD3 critic networks, 9x128 for both A2C networks and TD3 actor network. (b) corresponds to a fully connected layer of size 64x1 and 128x1 for larger and smaller architecture, respectively. The number of IF neurons in a layer is defined by the number of output size of its previous layer. Between two latent layers, there is another fully connected layer of size 128x128 is implemented.

We denote that a fully connected layer and a IF node layer form a spiking node layer. A general SNN contains spiking node layers as the latent layers, an encoder at the beginning

and an output layer (Decoder) at the end of the network. As the output layer, we used a non-spiking LIF node so the accumulated membrane potential after time-steps will give the network output.

The methods that we used require different network architectural designs such that TD3 contains two critic networks and the actor of A2C contains a standard deviation and a mean value output. The size of output and input tensors vary according to the algorithm of the methods. The only major difference is the actor of A2C splits into two parallel output layers as shown in the figure.

The provided robot arm model by simulation platform, give training possibility for six joints. However, we only trained three joints in our experiments.

The contrasting network architecture implementations have a different number of latent. In a larger architecture with a linear+IF node encoder, the network is defined with one latent layer. However, for a larger architecture with a Poisson encoder, the number of latent layers are two. The reason behind the scene is to have less number of non-linearity layers so the network does not become more complex. Accordingly, the number of latent layers for smaller network architecture is zero and one for a network with linear+IF node encoder and Poisson encoder, respectively.

During a general training run, both implemented RL algorithms use a replay buffer. The replay buffer can be seen as a memory in which agents hold the previous transition data. For TD3 implementation, the replay buffer starts from a given number of random observations in a simulation environment without any training phase. Furthermore, the replay buffer is crucial for the training procedure as the agent needs to learn according to its previous transitions. For the TD3 algorithm, we fetch a random batch of transitions from the replay buffer. However, in A2C implementation, we used reversed sequential transitions which are observed in an episode to train the agent.

In the experiments, we used the same values for certain hyperparameters I. In this regard, we played with hyperparameters; the number of timesteps used to encode the input, the network architectures and encoders.

TABLE I
THE COMMON HYPERPARAMETERS OF EXPERIMENTS

Learning Rate	15e-5
Max Timestep	10000
Proximity	0.2
Discount	0.99
Tries per Episode	8
Start Time-step	2500
Batch Size	256

The table II show the success results and average distance at the end of each training phase with normalized state and action value inputs.

Addtionally, we used non-normalized state and action values on a number of experiments with TD3 implementation. The best success rate of the experiments was 0.7 and average distance was 1.7. The used parameters are; 8, 3e-3, 0.95,

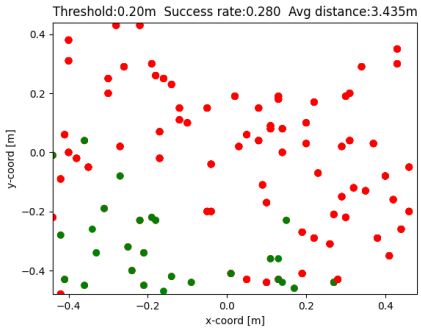

Fig. 3. TD3 Poisson Large Network Experiment

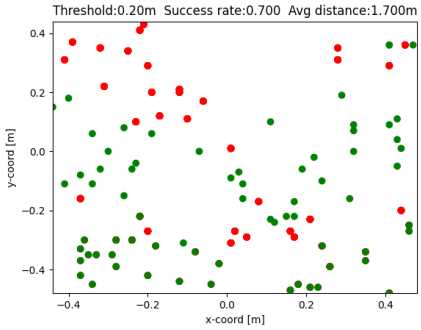

Fig. 4. TD3 Non-normalized Linear+If Experiment

Fig. 5. Graph of Successful Episodes

2 for T value, learning rate, discount and tries per-episode, respectively. This makes the best model that we had. In order to avoid collusion with the table surface, we added negative rewarding if the end-effector collides the table surface. The network use nearly the same architecture with general small network architecture that is defined above. The only different is the number of neurons in a layer is changed to 256 from 128 for latent layers.

TABLE II
THE EXPERIMENT RESULTS FOR A2C AND TD3 ALGORITHMS WITH DIFFERENT HYPERPARAMETERS

Method	Encoder	Network	T	Success	Avg Distance
TD3	Poisson	Small	8	0.1	3.952
TD3	Poisson	Small	16	0.15	3.736
TD3	Poisson	Large	8	0.2	3.979
TD3	Poisson	Large	16	0.28	3.435
TD3	Linear+IF	Small	4	0.04	4.002
TD3	Linear+IF	Small	8	0.0	4.656
TD3	Linear+IF	Large	4	0.17	4.329
TD3	Linear+IF	Large	8	0.2	3.902
A2C	Poisson	Small	8	0.24	3.494
A2C	Poisson	Small	16	0.04	6.562
A2C	Poisson	Large	8	0.0	NA
A2C	Poisson	Large	16	0.04	NA
A2C	Linear+IF	Small	4	0.0	NA
A2C	Linear+IF	Small	8	0.28	3.514
A2C	Linear+IF	Large	4	0.0	NA
A2C	Linear+IF	Large	8	0.0	NA

VI. Discussion and conclusion

One of the main drawbacks of deep SNNs is that their accuracy on typical landmarks does not reach the same standards as their NN counterparts. In addition, the learning algorithms of SNNs make the direct application of established backpropagation techniques problematic. We believe that the results achieved are not yet adequate, one of the issues is probably that reinforcement learning already involves numerous parameters, yet the SNN introduces an additional layer of complexity. Indeed, the choice of the encoding and the number of steps to encode an input has a strong influence on the results and this influence differs between agents. For example, the highest scoring agent TD3 had a Poisson encoder and a large network, with an input encoded on 16 timesteps, while the highest scoring A2C agent had none of these features in common, it had a linear+IF encoder, a small architecture and an input encoded on 8 timesteps. The performance of SNNs should be considered as a proof of concept only [2]: Since SNNs are a biological model, we expect them to be optimized only for the most behaviorally relevant tasks, such as making decisions based on continuous input instead of reaching an object whose position changes randomly and without any rationale.

Acknowledgment

Most of the trainings were run on a cloud platform by the Leibniz Rechenzentrum LRZ. We are grateful to our tutors Florian Walter, Mahmoud Akl and Josip Josifovski for their guidance throughout the project. All resources are provided by the Technical University of Munich.

References

[1] SINGH, Bharat, KUMAR, Rajesh, et SINGH, Vinay Pratap. Reinforcement learning in robotic applications: a comprehensive survey. Artificial Intelligence Review, 2021, p. 1-46.

[2] PFEIFFER, Michael et PFEIL, Thomas. Deep learning with spiking neurons: opportunities and challenges. Frontiers in neuroscience, 2018, p. 774.

[3] Carrillo RR, Ros E, Boucheny C, Coenen OJ. A real-time spiking cerebellum model for learning robot control. Biosystems. 2008 Oct-Nov;94(1-2):18-27. doi: 10.1016/j.biosystems.2008.05.008. Epub 2008 Jun 20. PMID: 18616974.

[4] A. Bouganis and M. Shanahan, "Training a spiking neural network to control a 4-DoF robotic arm based on Spike Timing-Dependent Plasticity," The 2010 International Joint Conference on Neural Networks (IJCNN), 2010, pp. 1-8, doi: 10.1109/IJCNN.2010.5596525.

[5] Mistry M, Schaal S. Representation and Control of the Task Space in Humans and Humanoid Robots. In: Cheng G PhD, editor. Humanoid Robotics and Neuroscience: Science, Engineering and Society. Boca Raton (FL): CRC Press/Taylor and Francis; 2015. Chapter 6.

[6] Sutton, R. S., and Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

[7] Wei Fang, Yanqi Chen, Jianhao Ding, Ding Chen, Zhaofei Yu, Huihui Zhou, Yonghong Tian, and other contributors. Spikingjelly: https://github.com/fangwei123456/spikingjelly,2020.

[8] NEFTCI, Emre O., MOSTAFA, Hesham, et ZENKE, Friedemann. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. IEEE Signal Processing Magazine, 2019, vol. 36, no 6, p. 51-63.

[9] SCHUMAN, Catherine D., PLANK, James S., BRUER, Grant, et al. Non-traditional input encoding schemes for spiking neuromorphic systems. In : 2019 International Joint Conference on Neural Networks (IJCNN). IEEE, 2019. p. 1-10.

[10] AUGE, Daniel, HILLE, Julian, MUELLER, Etienne, et al. A Survey of Encoding Techniques for Signal Processing in Spiking Neural Networks. Neural Processing Letters, 2021, vol. 53, no 6, p. 4693-4710.

[11] SKATCHKOVSKY, Nicolas, SIMEONE, Osvaldo, et JANG, Hyeryung. Learning to Time-Decode in Spiking Neural Networks Through the Information Bottleneck. arXiv preprint arXiv:2106.01177, 2021.

[12] Williams, R.J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach Learn 8, 229256 (1992).

[13] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu Proceedings of The 33rd International Conference on Machine Learning, PMLR 48:1928-1937, 2016.

[14] Scott Fujimoto, Herke Hoof, David Meger Proceedings of the 35th International Conference on Machine Learning, PMLR 80:1587-1596, 2018.

[15] Bornet, Alban, et al. "Running large-scale simulations on the Neurorobotics Platform to understand visionthe case of visual crowding." Frontiers in neurorobotics (2019): 33.