RISHABH KATIYAR

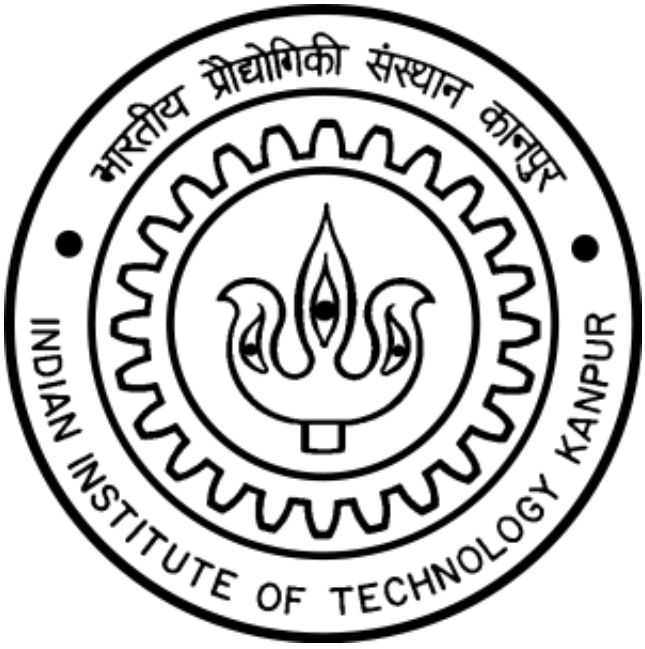
Indian Institute of Technology Kanpur

SURGE Project

Hardware Implementation of Reinfocement Learning Algorithms

Course Instructor: Dr. Shubham Sahay

December 24, 2022



Abstract

Standard computing systems are based on Von-Neumann Architecture, where CPU and memory are separate, and the data moves between them, thus increasing the latency. There is a memory wall due to the bottleneck of data to/from memory. With Moore's Law declining and the limit of scaling due to Heat Wall, we cannot increase the clock frequency much without damaging the chip. Neuromorphic Chips are suitable substitutes for the standard chips where memory, computation, and communication come all together similar to our brain.

Artificial Intelligence allows machines to depict human decisions, but sometimes the training data is difficult to obtain. Hence, Reinforcement Learning which allows machines or agents to simply learn from interacting with environments through rewards or punishments based on their actions, without the need for human intervention yields a robust machine-learning framework. Generally, these algorithms are implemented on conventional microprocessors or GPU, which store the majority of neural network parameters (for example, weights) in dynamic random-access memory (DRAM) thus moving data back-and-forth between the DRAM and the caches while performing computations leading to an increase in latency and power consumption.

However, Neuromorphic Hardware Systems that leverage memristors, the programmable and non-volatile two-terminal devices, as synaptic weights in hardware, neural networks could eventually be faster, more energy-efficient, and better adapted to environments than their software counterparts.

Various Reinforcement Learning Algorithms like Policy Gradient Algorithm, Q Learning, Deep Q Learning are being tested on the common benchmark problem which is the balancing of an inverted pendulum.

Contents

1 Reinforcement Learning **3**
- 1.1 The Cart-Pole Problem . . . 3
- 1.2 TD Learning . . . 5
- 1.3 Q Learning . . . 5
- 1.4 Policy Gradient . . . 6
- 1.5 Deep-Q Learning . . . 7

2 MEMRISTORS & HARDWARE IMPLEMENTATION **9**
- 2.1 Introduction . . . 9
- 2.2 Forming Process in RRAM's . . . 9
- 2.3 Switching in RRAM's . . . 9
- 2.4 Use of RRAM's as weights in Neural Network . . . 10
- 2.5 Hardware Implementation . . . 11
- 2.6 Implementation Details . . . 12

3 Results **13**

4 CONCLUSION **18**

Chapter 1

Reinforcement Learning

1.1 The Cart-Pole Problem

Many times it's difficult to obtain sufficient information about the environment that we are in and it's even more difficult to gather a sufficient amount of training data to train our supervised Machine Learning Model. In such scenarios or where the agent needs to consciously take actions adjusting to the environment in each step similar to a human brain, Reinforcement Learning is used. Reinforcement Learning contains five main parameters: Agent, Environment, Reward, Policy, and Value Function.

An agent interacts with the environment and based on the rewards it gets from the environment which is the only feedback signal for the agent to train, agent chooses the action to take in the next time step according to the policy and estimating the value of the state that the agent would end up in by taking that particular action by looking at the Value function V(St). All the Reinforcement Learning algorithms are designed to increase this value of each state as much as possible and thus the finding the optimal value function.

Policy pi(a—s) is map from state to action and is represented as: Deterministic Policy : a = pi(s) Stochastic Policy :

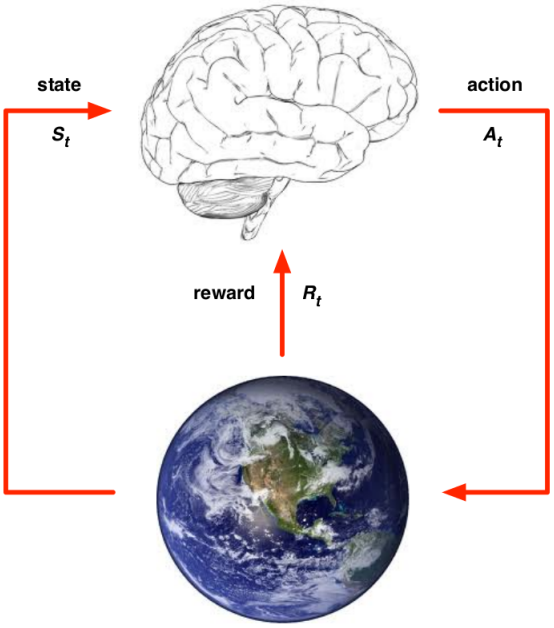
$$\pi(a|s) = P[A_t = a|S_t = s] \tag{1.1}$$

The Value Function is a prediction of future reward and is used to evaluate the goodness/badness of states and therefore to select between actions. It is represented as :

$$V_\pi(s) = E_\pi[R_{t+1} + \gamma * R_{t+2} + \gamma^2 * R_{t+3} + ...|S_t = s] \tag{1.2}$$

This represents the expected values of a state based on the future rewards that it will get. Here gamma is the discount factor.

Generally the flow of information is as follows :



We have taken the common problem of balancing an inverted pendulum using Reinforcement Learning.

An un-actuated joint attaches a pole to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright, and a negative reward of -200 is given every time the pole falls. The episode ends when the pole is more than 15 degrees from vertical or the cart moves more than 2.4 units from the center.

We have considered the state to be a 4-dimensional vector containing
$\times$ position of the cart on the track
θ angle of the pole with the vertical
$\times$' cart velocity
θ' rate of change of the angle or the angular velocity

Based on these states and rewards, an action is taken in each time step to move the cart, either left or right, to balance the cart.

All of the Reinforcement Learning Algorithms boils down to the Bellman Generality Equation

$$V_\pi(s) = E_\pi[R_{t+1} + \gamma * V_\pi(s_{t+1})|S_t = s] \tag{1.3}$$

Or for the action-value function is this:

$$Q_\pi(s, a) = E_\pi[R_{t+1} + \gamma * Q_\pi(s_{t+1}, a_{t+1})|S_t = s, A_t = a] \tag{1.4}$$

The interpretation of this equation is as follows, Suppose I'm in a state s_t, and I take action a_t, then I would receive an immediate reward R_{t+1}, and I would end up in a state s_{t+1} based on the environment, and from there I would take the next action a_{t+1} based on my policy.

1.2 TD Learning

Whenever we leave our agent in the environment, it takes certain actions and ends up in certain states; this sequence of steps and actions is called an episode, and the episode ends when we reach the terminal state.

TD Learning is an algorithm to estimate the value of a state St means how good or how bad is a particular state. So naturally, the states near our goal state or a pendulum example, where our pendulum is held upright, would have a higher value, while the states where our pendulum falls will have a lower value.

We update the value of the state St based on the experience that the agent has collected so far while interacting with the environment.

$$V(s_t) \longleftarrow V(s_t) + \alpha * (R_{t+1} + \gamma * V(s_{t+1}) - V(s_t) \tag{1.5}$$

$$R_{t+1} + \gamma * V(s_{t+1}) \tag{1.6}$$

Equation 1.6 is called the TD target.

$$\delta_t = R_{t+1} + \gamma * V(S_{t+1}) - V(S_t) \tag{1.7}$$

Equation 1.7 is called the TD error.
Means by how much do we differ from the value of the state we expected it to be as V(S_t) and how much it actually is $R_{t+1} + \gamma * V(S_{t+1})$.

1.3 Q Learning

Q Learning can also be said as Temporal Difference Control Algorithm when we are trying to find the best policy or how our agent should behave in the environment optimally and the most optimal Action-Value function. Regarding the Inverted Pendulum example, the best policy determines whether we have to put a force of +1 or -1 means whether we have to push the pendulum left or right to keep the pendulum upright.

Q Learning Algo. makes use of two policies, one is the behavior policy, and one is the target policy. Behavior policy is the one according to which we choose our actions. We decide to optimize the Target policy by greedily selecting the action or by always taking that action from a particular state with the highest Q(s, a) value.

The Behavior Policy is epsilon-greedy with respect to the Q(s,a) function. There is an essential aspect in Reinforcement Learning which is Exploration vs. Exploitation. If we always choose the action that benefits us the most(Exploit our actions) then we will not be able to explore the whole state-space or w.r.t to the pendulum example we may not be able to explore every possible configuration in which the pendulum could be in. Thus exploring the State-space or choosing our action sub-optimally is important. Epsilon- greedy policy helps us to do just that. Its form is:

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & if a* = argmax_{a \epsilon A} Q(s,a) \\ \epsilon/m & otherwise \end{cases} \tag{1.8}$$

With probability (1-ϵ) we choose the greedy action and with probability ϵ, we choose a random action.
The target policy π is greedy with respect to Q(s, a)

$$\pi_{S_{t+1}} = argmax_{a'} Q(S_{t+1}, a') \tag{1.9}$$

The behavior policy μ is e.g. ϵ-greedy w.r.t. Q(s, a)

The Q Learning target then simplifies:

$$\begin{aligned} &= R_{t+1} + \gamma * Q(S_{t+1}, A') \\ &= R_{t+1} + \gamma * Q(S_{t+1}, argmax_{a'} Q(S_{t+1}, a')) \\ &= R_{t+1} + max_{a'} \gamma * Q(S_{t+1}, a') \end{aligned} \tag{1.10}$$

Thus it takes the final form

$$Q(S, A) \longleftarrow Q(S, A) + \alpha * (R + \gamma * max_{a'} Q(S', a') - Q(S, A)) \tag{1.11}$$

1.4 Policy Gradient

In Q Learning, a policy was generated directly from the value function e.g., using epsilon-greedy
In Policy Gradient, we will directly parametrize the policy, using a function approximator for the policy.
$\pi_\theta(s, a) = P[a|s, \theta]$
Here, θ is the parameter that we will tune to get the optimal policy. Policy-Based Reinforcement Learning is an optimization problem that solves the problem of finding θ to maximize J(θ).

Policy Gradient algorithms search for a local maximum in J(θ) by ascending the gradient of the policy w.r.t parameters θ.

$$\delta\theta = \alpha * \nabla_\theta(\theta)$$

We now compute the policy gradient analytically. Assume policy π_θ is differentiable whenever it is non-zero and we know the gradient $\nabla_\theta \pi_\theta(s, a)$

Likelihood ratios exploit the following identity:

$$\begin{aligned} \nabla_\theta \pi_\theta(s, a) &= \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} \\ &= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) \end{aligned}$$

The score function is $\nabla_\theta log_{\pi_\theta}(s, a)$.
Now using the score function, we can define $\nabla_\theta J(\theta)$ as

$$
\begin{aligned}
J(\theta) &= \mathbb{E}_{\pi_\theta}[r] \\
&= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s,a) \mathcal{R}_{s,a} \\
\nabla_\theta J(\theta) &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) \mathcal{R}_{s,a} \\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) r]
\end{aligned}
$$

The above equation tells us that if we want to increase our objective function, move in the (score function * reward) direction.
If you have a large negative reward, adjust the policy parameters such that you move in a direction away from it. If you have a large positive reward adjust the policy parameters to move in a direction towards it.

We can replace 'r' with our action-value function Q(s,a), thus leading to the policy gradient theorem, which states :
For any differentiable policy $\pi_\theta(s,a)$, for any of the policy objective functions J, the policy gradient is :

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log_{\pi_\theta}(s,a) * Q^{\pi_\theta}(s,a)]$$

Thus, we can get the gradients of the objective function with respect to θ; we can update the parameter values using any known optimization algorithms like Gradient descent, RMS Prop, or Adam.

1.5 Deep-Q Learning

Deep-Q Learning is almost the same as Q-Learning except for the fact that DQN(Deep-Q Network) uses experience replay and fixed Q-targets.
Here the action-value function Q is also parameterized using a function approximator. Like we could use an RBF(Radial Basis Function Approximator) for approximating Q or a Neural Network as a non-linear function approximator for Q

$$\hat{q}(\text{S,A,w}) = q_\pi(\text{S,A})$$

Experience replay means that we first let the agent roam in the environment with a random policy. The agent thus collects experience while randomly interacting with the environment in the form of $(s_n, a_n, r_{n+1}, s_{n+1})$ tuples and are stored in a replay memory. These stored observations also termed as 'experience', are used to train the agent for the decision-making in the future, which benefits the learning by minimizing correlations between samples.

Until now, the same Q-network was used for predicting the Q-value of the current state and next state. The Q-value of the next state is then used to calculate the ground truth. In other words:
We executed our optimization step to bring the prediction close to the ground truth but at the same time we are changing the weights of the network which gave us the ground truth. This causes instability in training.

The solution is to have another network called Target Network, an exact copy of the Main Network. This target network is used to generate target values or ground truth. The weights of this network

are held fixed for a fixed number of training steps after which these are updated with the weight of Main Network. In this way, the distribution of our target return is also held fixed for some fixed iterations, increasing training stability.
The Deep-Q Learning algorithm takes the following steps:

- Take action at according to the policy. The policy could be parameterized using a neural network or could be epsilon greedy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1}$) in replay memory D
- Sample random mini-batch of transitions (s, a, r , s') from D
- Compute Q-learning targets w.r.t. old, fixed parameters w
- Optimise the loss function between Q-network and Q-learning targets. Here the loss function chosen is the Mean Squared Error.
- Update the parameters w using any update scheme like the stochastic gradient descent

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i} \left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i) \right)^2 \right]$$

Chapter 2

MEMRISTORS & HARDWARE IMPLEMENTATION

2.1 Introduction

Memristors or Resistive RAM's RRAM's are programmable and nonvolatile two-terminal devices. Memristors are capable of storing the weights of the Neural Network (encoded) as conductances. We can change the conductance of memristors through the switching process.

2.2 Forming Process in RRAM's

Forming Process leads to soft breakdown. It is the process by which the High Resistance State of RRAM's is converted to the Low Resistance State. At high enough Electric fields, some oxygen atoms are knocked out of their lattice position and drift towards the anode thus leading to defect generation. The deficiency of atoms leads to the formation of conductive filament. And as soon as we remove the Electric field or the Voltage, the defects close to the electrodes get recovered during the HRS switching/RESET process.

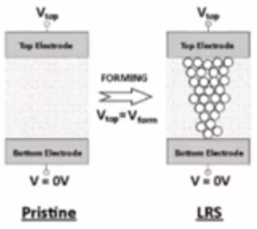


Figure 2.1: SET Process

2.3 Switching in RRAM's

Switching mechanism in RRAM's is not only a function of the oxide material but also depends on the electrodes. The conductance of the RRAM's can be changed by two processes SET and RESET.

SET Process: switching of device from high resistance state(HRS) to low resistance state(LRS). We apply stress or a high pulse on the RRAM's so that kind of forms a small breakdown and leading to the development of a filament connecting the Bottom electrode and the Upper Electrode thus

switching to LRS or high conductance state.

RESET Process :
As soon as the RESET Pulse is applied, oxygen ions close to the anode migrate back to the bulk. Thus they combine with the oxygen vacancies in the Conducting Filament and as a result the Conducting Filament dissolves thus leading to HRS or less conductance state.

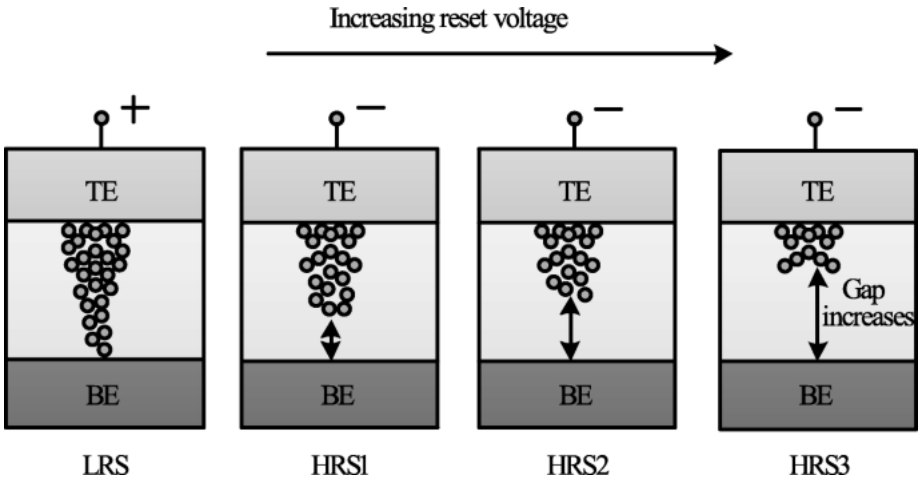
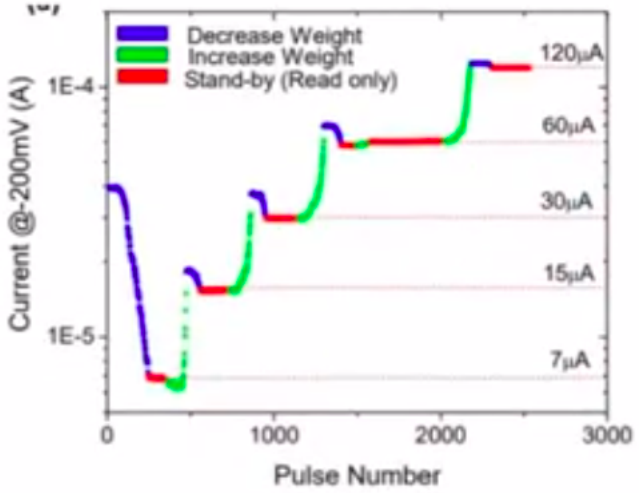


Figure 2.2: RESET Process

2.4 Use of RRAM's as weights in Neural Network

Memristors are capable of storing the weights of the Neural Networks encoded as their conductance states. There could be a linear mapping of the Neural Network's weights as the memristors' conductance. We can tune the weights of the NN by just changing the conductance states of memristors through update schemes.
One such update scheme is the Program Verify Algorithm.

Let's say if we are looking at conductance values as states, so for the first 7uA at a constant supply of 200 V (By Ohm's Law: Conductance = Current/Voltage), and we need to go to a higher conductance value at 15uA. Hence we apply a read signal to know our current conductance value. Since we have not reached our desired conductance value, we apply a SET voltage to increase the conductance, then again a read voltage, and then again set until we reach out required conductance. If we overshoot, we apply a RESET pulse then read, and we repeat this untill we converge to our desired conductance state.

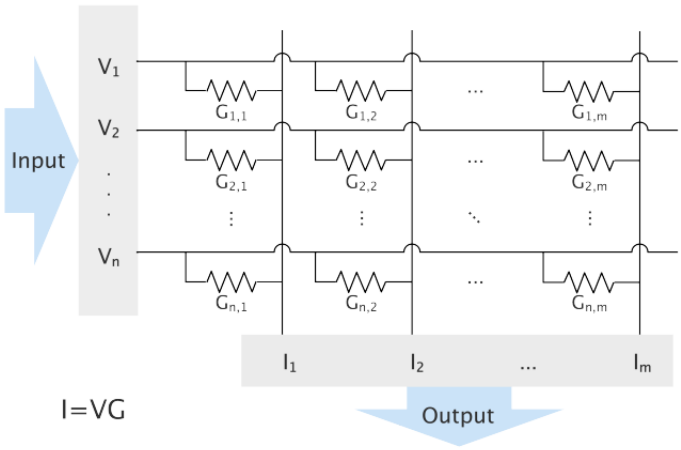


2.5 Hardware Implementation

With long retention, high integration density, low power consumption, analog switching, and high non-linearity of the current voltage curve, memristive devices are considered very promising candidates for implementing artificial synapses. Additionally, since memristors can integrate computation and memory in the same physical place, data movements between computing and memory elements can be significantly reduced compared to the software counterparts. With these characteristics, memristor-based crossbars are very efficient at computing analog vector-matrix multiplications in the locations where the matrices are stored, highlighting its potentials in neuromorphic computing.

There is a memristor cell in each intersection of the crossbar. An input vector V is applied to the rows and multiplied by the conductance matrix of memristors. The resulted currents are summed up by Kirchhoff's law in each column. The MAC operations are done in the form of I = VG where I is the current vector, V is the voltage vector, and G is the conductance vector. A large number of ADC's(Analog to Digital Converters) & DAC's(Digital to Analog Converters) are used in between the network. Therefore all the multiply-accumulate (MAC) operations can be done in-place.

A memristor array is being used where a pair of memristors represent one neuron of the network. For our purpose, the states of our RL environment are encoded as voltages and fed in the input layer of our memristor network, and the suitable number of actions that we want to determine are taken as output in the form of probabilities from the output layer of our network.



2.6 Implementation Details

I have tried different algorithms as explained in the first section using Q-Learning, Policy Gradient & Deep-Q Learning to balance the inverted pendulum environment. The cart-pole environment has been imported from the open-AI Gym and it consists of 4 states and 2 actions. By default, the cart pole environment has been set that if we get a reward of 200 is the maximum possible reward that we can achieve or if we get a total cumulative reward of 200, our agent is supposed to be trained.

Instead of the conventional gradient update rule where the gradients are calculated with respect to the lost function and then the weights and biases are updated according to the algorithm we have chosen like Gradient Descent or Adam, we have used the Manhattan update rule to apply the gradients.
In the Manhattan update rule instead, of updating the weights based on their actual gradient values we look at whether the gradient is positive or negative. If the gradient is positive we update(or increase) the corresponding weight/bias by a fixed amount and if the gradient is negative we decrease it by a fixed amount.
In conventional hardware, the update of weights and matrices would have been done by applying voltage pulses along the row and the column of the crossbar matrix. This would have done a one-shot update of the conductances(which are encoded as weights).

One problem we face is that after applying the SET/RESET voltage in a particular row or column, those conductances could also change, which we don't want to change. To avoid that or in order not to disturb half-selected devices Vx (voltage applied along the row) & Vy(voltage applied along with the columns) have to be below the corresponding effective switching thresholds, while Vx + Vy have to be larger than the threshold. Thus by doing this only those conductances change which we want to change.

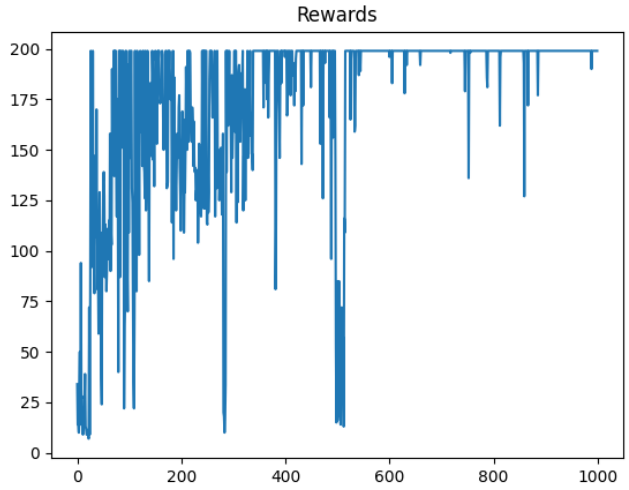
There is also a dependence for how much time we are applying the SET/RESET voltage which can be compared to the learning rate alpha in the standard weight update rules.

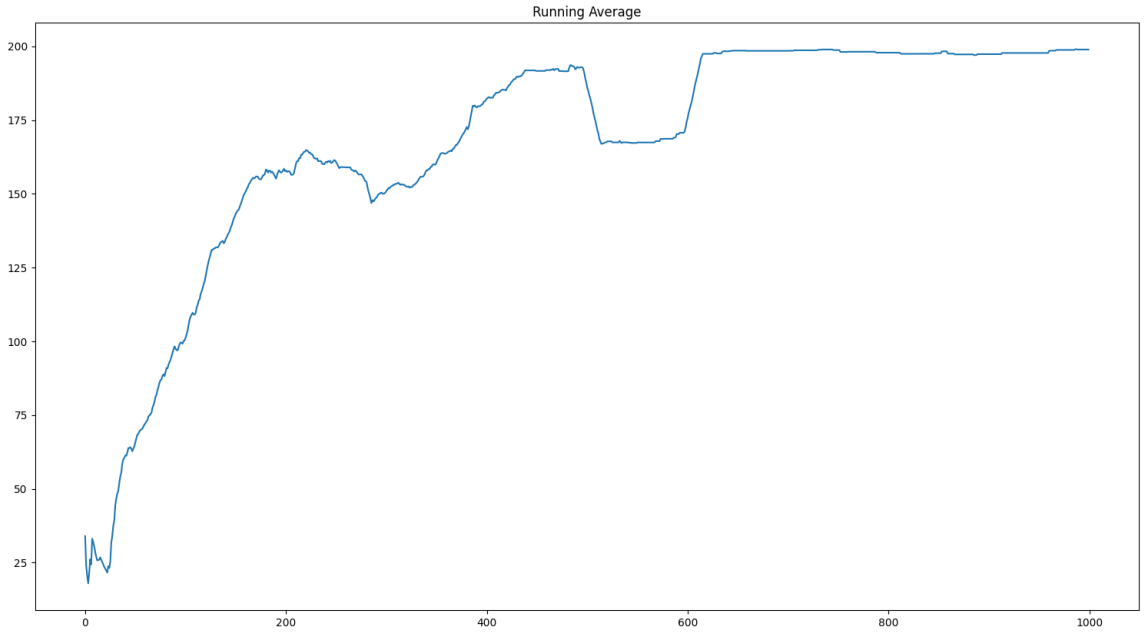
Since we are doing a simulation here, I took the individual gradients, checked out their values if they are positive or negative, clipped them to a particular fixed value according to their sign and then updated the weights through the normal optimization algorithm that I had used.

Chapter 3

Results

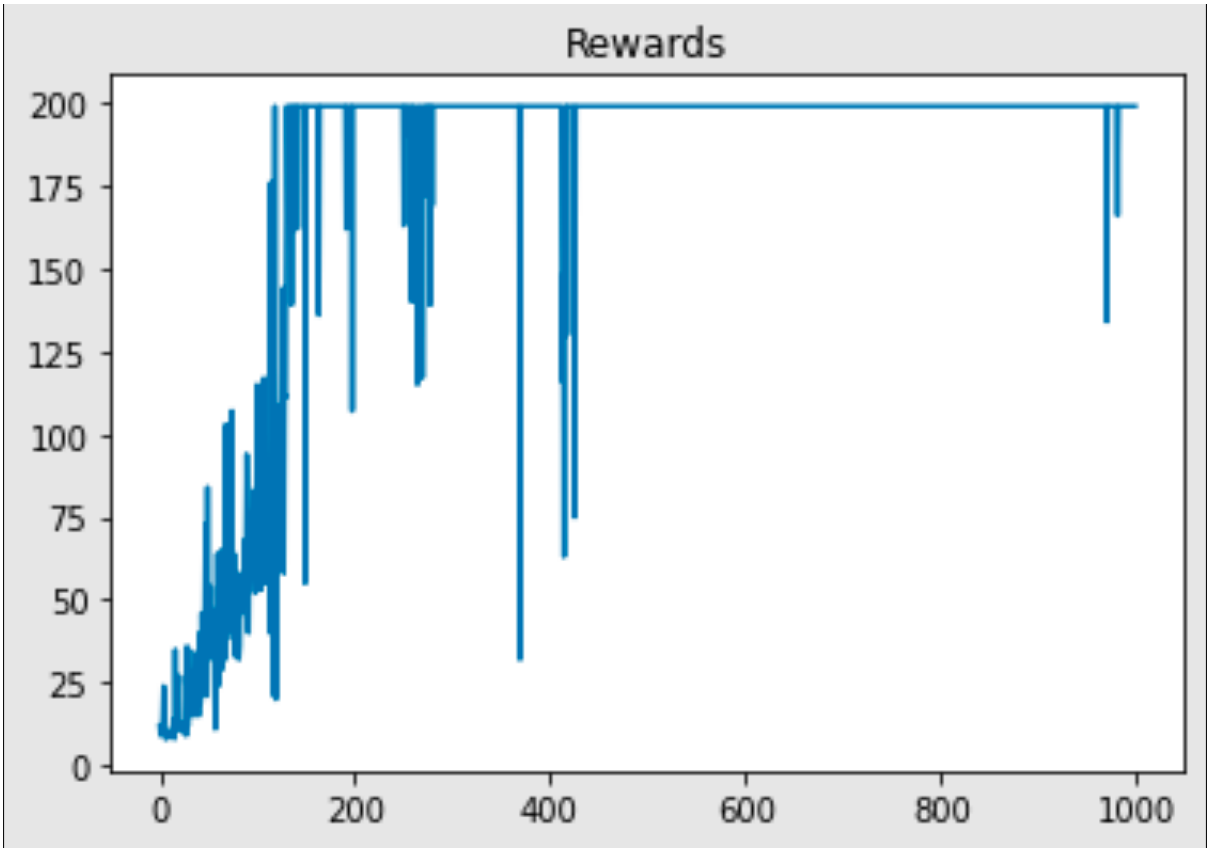
In the Q-Learning Algorithm, we used the RBF function as the linear function approximator for our policy and value function instead of a Neural Network. The parameter values are :
γ : 0.99
Learning rate (α) = 0.1
Epsilon: is a continuously decreasing function depending on the number of episodes. This conveys that initially for a lower value of epsilon we explore more and then after a sufficient number of episodes, we exploit our best action.
$\epsilon = 1/\sqrt{N+1}$; N = Number of episodes
Number of episodes = 1000
Max. number of time steps = 2000

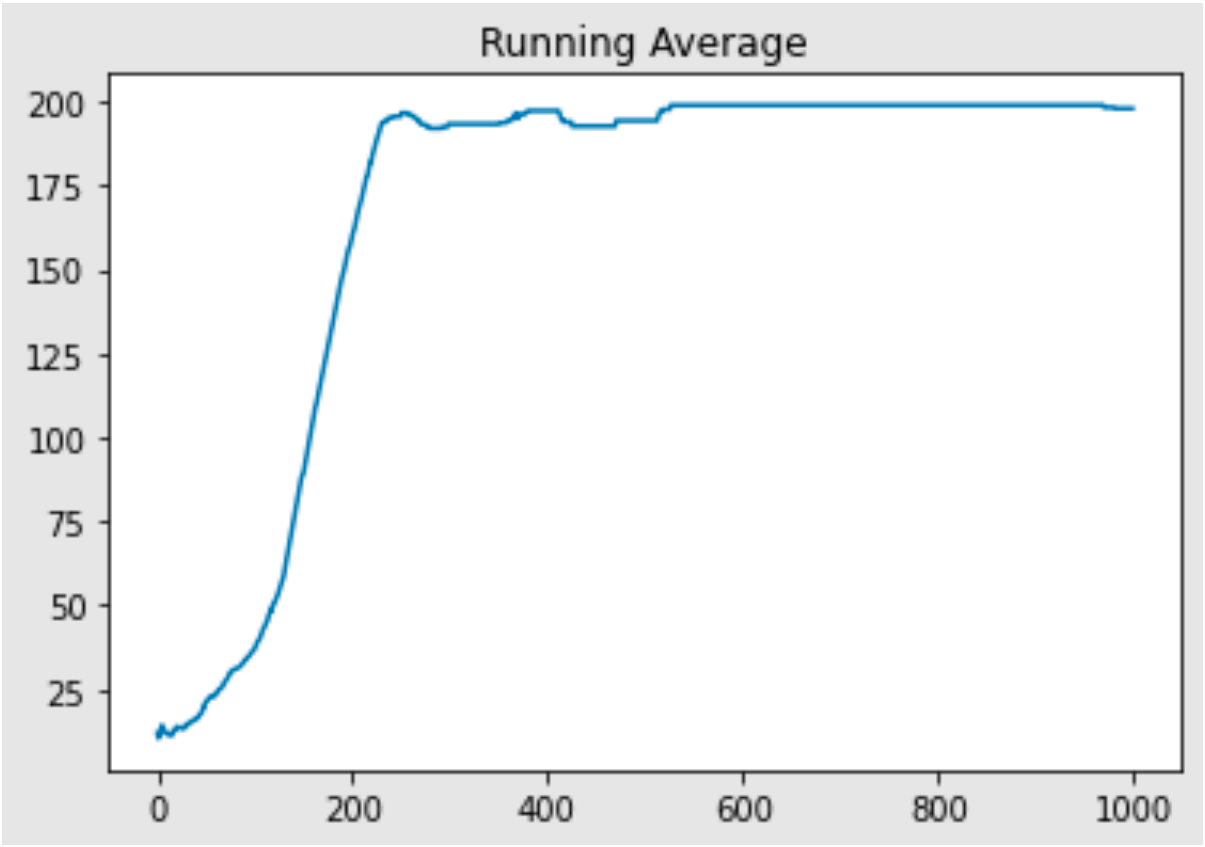




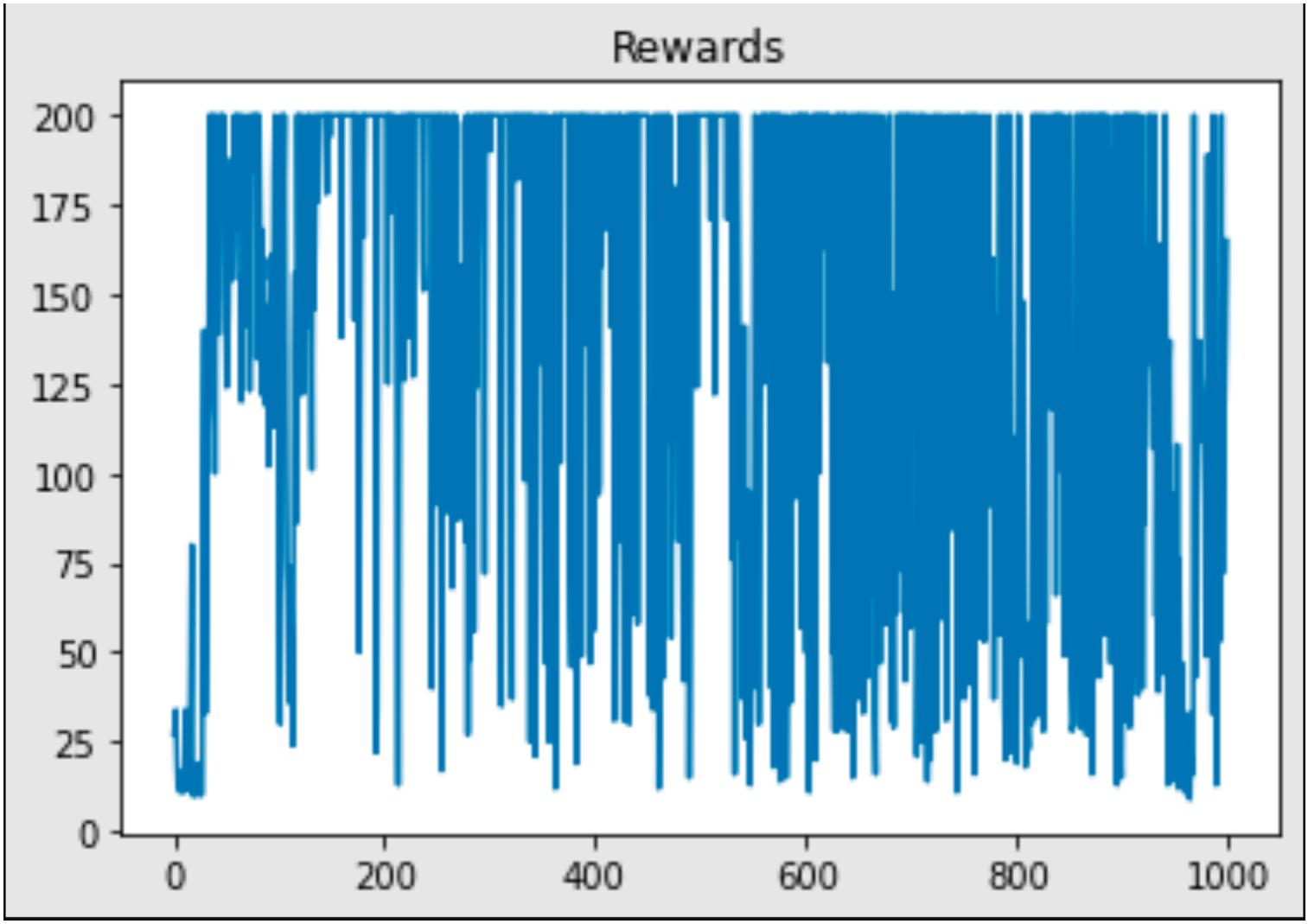
In the Policy Gradient Algorithm, the policy and the value function are both approximated by Neural Networks. The Policy function is a 0 layer hidden network with the input as states(number of states = 4) directly mapped to the actions(number of actions = 2). For the value function, we have taken a 2 layer hidden network with 10 & 8 neurons respectively. The number of neurons taken as input is 4 and the output layer has 2 neurons(for each action - left & right push).

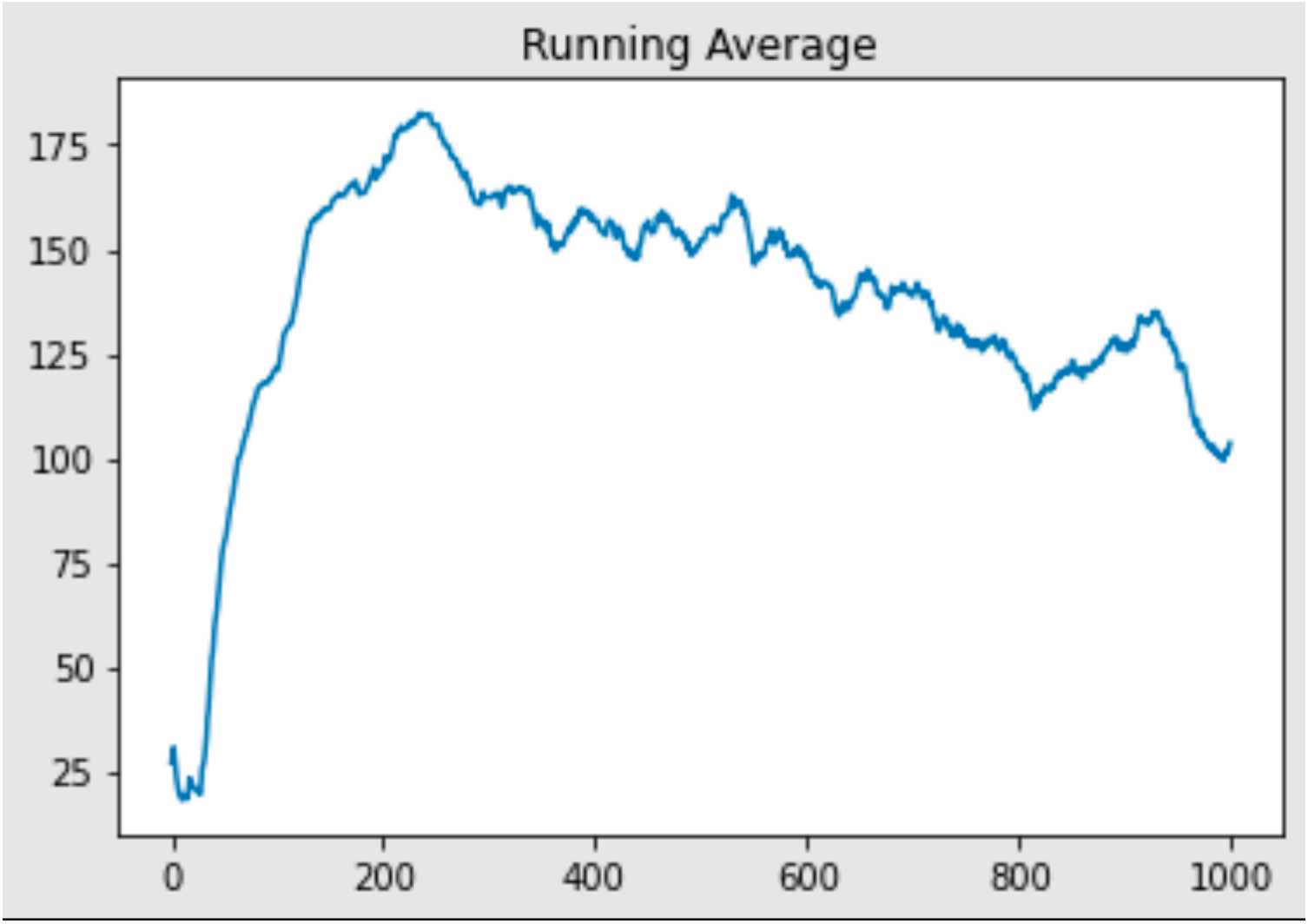
The parameter values used for training are:
γ : 0.99
For the Policy model :cost function is the score function as in section equation *1.12*
Optimizer : Adam Optimizer (α=0.1, β_1=0.9, β_2=0.999, ϵ=1e-07)
For the Value Model: cost function : Squared Error loss
Optimizer : Gradient Descent Optimizer (α = 0.0001)
Number of episodes = 1000
Max. number of time steps = 2000


Rewards
200
175
150
125
100
75
50
25
0
0
200
400
600
800
1000


Running Average
200
175
150
125
100
75
50
25
0
200
400
600
800
1000

In the Deep-Q network, the value function is approximated by a Neural Network of 2 hidden layers having 200 neurons each. We have chosen the epsilon greedy policy here. The parameter values used in training are:
γ : 0.99
Cost Function: Mean Squared Error
Optimizer: Adam Optimizer (α=0.01, β_1=0.9, β_2=0.999, ϵ=1e-07)
Experience Replay Memory Size: 100
Batch size for training: 32
Copy Period = 50 (for the target Q network to copy parameters from the parent network)




Running Average
175
150
125
100
75
50
25
0
200
400
600
800
1000

Chapter 4

CONCLUSION

We have therefore seen that the Policy Gradient model gave us the best results in terms of the total net reward, the average rewards as well as the number of episodes. It converges to the maximum reward in the least number of episodes.

Bibliography

[1] Zhongrui Wang, Can Li, Wenhao Song, Mingyi Rao, Daniel Belkin, Yunning Li, Peng Yan, et al.(2019). Reinforcement learning with analog memristor arrays. Nature Electronics, 2, 115–124. https://doi.org/10.1038/s41928-019-0221-6

[2] Nan Wu, Adrien Vincent, Dmitri Strukov, and Yuan Xie(2020). Memristor Hardware-Friendly Reinforcement Learning. arXiv:2001.06930

[3] Famood Merrikh-Bayat, Elham Zamanidoost and Dmitri Strukov(2015).Efficient Training Algorithms for Neural Networks Based on Memristive Crossbar Circuits.IEEE