university of groningen

faculty of science and engineering

University of Groningen

Interpretable Reinforcement Learning with the Regression Tsetlin Machine

Master's Thesis

To fulfill the requirements for the degree of
Master of Science in Artificial Intelligence
at University of Groningen under the supervision of
dr. S. Hamidreza Mohades Kasaei (Artificial Intelligence, University of Groningen)
and
Prof. dr. Ole-Christoffer Granmo (CAIR, University of Agder)

Varun Ravi Varma (s3893030)

In Memory Of:
Dr. Marco Alexander Weiring

January 16, 2022

Contents

Acknowledgments

First and foremost I would like to thank my first supervisor, Dr. Marco A. Wiering. Without his availability and guidance, this research would not have been accomplished. This research would have been incomplete without his insights into Deep Learning and Reinforcement Learning. The zeal and enthusiasm with which he treated his courses will forever be a cornerstone in guiding my foray into Artificial Intelligence.
I would also like to thank Prof. Dr. Ole-Christoffer Granmo for his valuable feedback and suggestions on the core technical components of this research. I would like to extend my heartfelt thanks for their support throughout the duration of this project.
I would also like to thank Dr. S. Hamidreza Mohades Kasaei, for assisting me with the final stages of this project and his feedback on the work.
I would also like to thank Mr Anil Varma and Mrs Preethi Varma for their continuous support in all my endeavours. I must also thank Ashwin Varma for his humor and affirmations in these trying times.
I must also thank Nicholas Kees Dupuis for the discussions and thought-bouncing sessions; my dear friend and colleague Magdalena Bilska for the emotional support and writers-block cures; and my friends across the world for reminding me that sometimes, we need to take a step back and breathe.
I would also like to express my gratitude to my Academic Advisor Dr. Rachel van der Kaaij for her consistent support and encouragement throughout my time as a student at the university.
I would also like to thank the Center for Information Technology of the University of Groningen for their infrastructural support and for providing access to the Peregrine high performance computing cluster. A special thanks to Dr.Remco Wouts for assisting me with navigating the Pallas AI systems at the Bernoulli Institute.

Abstract

As Artificial Intelligence (AI) methods are being widely adopted by industries in multiple domains, there is an increasing concern in the ethics and transparency of decisions made by AI tools. Although these black box agents improve the speed of decision making, we have seen quite a few scenarios where the agent displays clear biases in the decisions made. Hence, it has become increasingly important that the AI agent behaviour is transparent. Such transparency can be brought about either by making the agent's decision making process interpretable or by utilizing external tools to view the decision making process (white box equations or visualization tools).

Reinforcement Learning (RL) methods are widely used in problem spaces involving continuous control, such as games, self-driving cars and robotics. RL methods commonly involve an agent that learns to update its behaviour over time, through feedback generated from continuous interactions with an environment. Widely used RL agents utilize Deep Neural Networks (DNNs) to make decisions, and this makes the agent behaviour difficult to explain, since the complexity of DNNs increase with each layer.

The Tsetlin Machine is an alternative learning mechanism to deep neural networks, which has been shown to have comparable accuracy to state of the art machine learning algorithms with the additional advantages of interpretability and computational simplicity. In this project, we aim to implement an interpretable reinforcement learning framework utilizing Tsetlin Machines and Q-learning. We will then test this agent out on two classical control environments and try to understand how patterns in the input lead to decisions and also compare its performance with existing reinforcement learning methods.

Chapter 1

Introduction

Artificial Intelligence (AI) has been adopted for a wide variety of tasks varying from the detection of diseases to playing games, with reasonable success in understanding underlying pattern distributions and reducing the time taken to make a decision. However, underlying algorithms do not successfully explain how a decision was arrived at. Interpretability of algorithms would allow humans to identify how patterns contribute to a decision [1], address any inherent biases in the decision-making process and augment the models' decisions with additional data where available.

Reinforcement learning is a machine learning methodology wherein an agent learns to solve a problem through repeated interactions with a dynamic environment. Agents learn to choose optimal actions for the given environment through rewards and punishments without actually being taught how to solve the problem [2]. Reinforcement learning strategies have been applied in the fields of memory control in computers, simulating video game players, mastering strategy games, [3], cognitive robotics [4] and autonomous driving [5].

There exist two approaches to Reinforcement Learning [6]. In Model-based learning, the agent uses a model of the environment to predict outcomes of possible actions in a given scenario. The effectiveness of this approach is limited to the accuracy of the model of the environment. Model-free learning abstracts the agent behaviour into choosing an optimal policy, instead of trying to create an internal model of the environment [7].

The Tsetlin Machine [8] is an alternative learning mechanism to deep neural networks, which has been shown to have comparable accuracy to state-of-the-art machine learning algorithms with the additional advantages of interpretability and computational simplicity. In its most simplest form, the Tsetlin Machine is a pattern mining framework, utilizing binary inputs to decide the state of each automaton in the pool. These states form internal pattern representations, which are interpretable as propositions. Each proposition may be considered similar to a vote, cast by the automaton, on whether it finds a sub pattern in the input. The set of active propositions can then be used to perform classification [8, 9, 10] and regression [11, 12], depending on the function used to compute the total votes.

Applications of reinforcement learning in various continuous control situations require the reinforcement learning model or agent to make quick decisions in real-time. The implications of erroneous decisions may be catastrophic, as in the case of networked devices, autonomous vehicles and automated manufacturing lines, where reinforcement learning approaches are commonly used. There have been a few approaches to improve the interpretability of the decisions made by agents, such as interpretable Q-learning [13, 14] and interpretable deep reinforcement learning using symbolic planning [15]. These approaches utilize secondary frameworks to enforce interpretability [16], increasing the computational costs involved as well as the time taken to make decisions. Although an external framework may seem an ideal way to improve agent interpretability, it is not ideal in the case of problems involving continuous decision making. The Tsetlin Machine, with its interpretable clause based approach to learning, is thus an interesting alternative to neural networks for interpretable Reinforcement Learning.
In this thesis, we aim to implement a reinforcement learning framework utilizing Tsetlin Machines and Q-learning[17], a model-free reinforcement learning algorithm. We will also attempt to test the interpretability, trying to understand how patterns in the input lead to decisions and compare our automaton-based agent's performance with existing reinforcement learning methods. In the subsequent sections in this chapter, we will discuss the pertinent research questions, the significance of this study and conclude with a section presenting the layout of the chapters that follow.

1.1 Research Questions

The thesis attempts to answer the following research questions:

Q1. How can we design a Reinforcement Learning system utilizing Regression Tsetlin Machines?

Q2. How does the performance of the Tsetlin Machine Reinforcement Learning agent compare with an agent trained using a Deep Q Network?

Q3. How understandable is the decision making process of the Tsetlin Machine Reinforcement Learning agent?

1.2 Significance of the Study

Firstly, the study aims at creating an architecture facilitating Q-learning using the Regression Tsetlin Machine. The architecture is expected to be a first step in utilizing Tsetlin Machines for Reinforcement Learning problems, specifically in problems involving continuous input spaces.
Secondly, the new architecture is expected to preserve the interpretability of the Tsetlin Machine, enabling us to study how input patterns from the environment directly contribute to the computed Q-values for a state-action pair. As an extension, the interpretability would allow us to understand reinforcement learning problems better, translating to design of better algorithms that can generalize well over a wide range of problems.
Finally, interpretable reinforcement learning may assist industry in designing better input and feedback systems, particularly in domains involving automated control of agents such as robotics and self-driving cars.

1.3 Thesis Layout

The structure of the thesis is as follows. The first chapter introduced the research topic of the study and presented the research questions we aim to answer. The second and third chapters discuss the theoretical background for the study, introducing Tsetlin Machines in chapter two and a brief discussion of Reinforcement Learning methods in chapter three. In the fourth chapter, we introduce the approach to Reinforcement Learning using two variants of the Regression Tsetlin Machine for Q-learning. The fifth chapter describes the experiments used to evaluate the new approach. In the sixth chapter, we present the results from our experiments, leading to a discussion and possible future directions in the final chapter.

Chapter 2

Tsetlin Machine

Deep Networks using interconnected Artificial Neurons have become a fundamental building block of Artificial Intelligence systems, with the ability to perform well on supervised and unsupervised learning problems. Although they have very simple structures as neurons, the complexity of computations increases with the number of interconnected neurons in a network. The depth of the network and other architectural features such as the activation functions also decide the functions the network can express [18]. With this increase in complexity, and the layers of abstraction between the input and the output, decisions made by deep neural networks are difficult to interpret.
The Tsetlin Automaton, developed by M.L. Tsetlin in the 1960s, is a simpler alternative to the Artificial Neuron. A single Tsetlin Automaton (as in Figure 2.1), designed to solve the multi-armed bandit problem [19], is capable of learning optimal actions in an environment through reward mechanism triggered by changes in state. A simple two-action Tsetlin Automaton is defined by the quintuple [20]:

$$\{\underline{\Phi}, \underline{\alpha}, \underline{\beta}, F(\cdot,\cdot), G(\cdot)\}$$

- $\underline{\Phi}$: set of internal states ($\underline{\Phi} = \{\phi_1, \ldots, \phi_{2N}\}$)
- $\underline{\alpha}$: set of automaton actions (in Figure 2.1, $\underline{\alpha} = \{\alpha_1, \alpha_2\}$)
- $\underline{\beta}$: set of feedback given to the automaton, in terms of reward and penalty
- $F(\phi_t, \beta_v)$: transition function, determining the internal state of the automaton, based on state ϕ_t and the feedback β_v (v can be either a reward or a penalty)
- $G(\phi_t)$: output function, determining the action α_t performed by the automaton given the current state ϕ_t

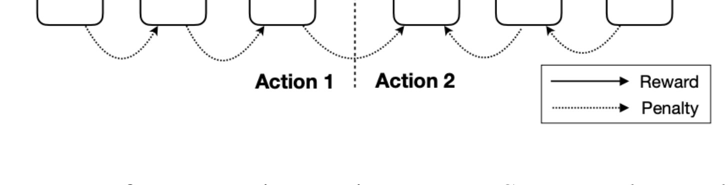


Figure 2.1: Tsetlin Automaton for two-action environments. Source: *The Tsetlin Machine – A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic* [8]

In its programmatic implementation, the Tsetlin Automaton only tracks a state index, the learning mechanism increments or decrements this state index based on the transition function $F(\phi_t, \beta_v)$. This makes the automaton much simpler than an artificial neuron, giving it a smaller memory footprint as well.
Similar to a deep neural network, the Tsetlin Machine is made up of teams of Tsetlin Automaton [8]. As in Figure 2.2, a an input vector of length o is passed through conjunctive clauses (Tsetlin Automaton) for evaluation. The sum of these votes from individual Tsetlin Automaton are then passed through a threshold function (a unit step function). The final output of the team of automaton is thus given by [8]:

$$\hat{y} = u\left(\sum_{j=1}^{n/2} C_j^1(X) - \sum_{j=1}^{n/2} C_j^0(X)\right) \tag{2.1}$$

- $\hat{y}$ is the predicted output (in this case, $\hat{y} \in 0, 1)$
- X is the input vector of size o ($X \in$)
- $C_j^1(X)$ are the clauses assigned positive polarity
- $C_j^1(X)$ are the clauses assigned negative polarity
- u is the unit step threshold function

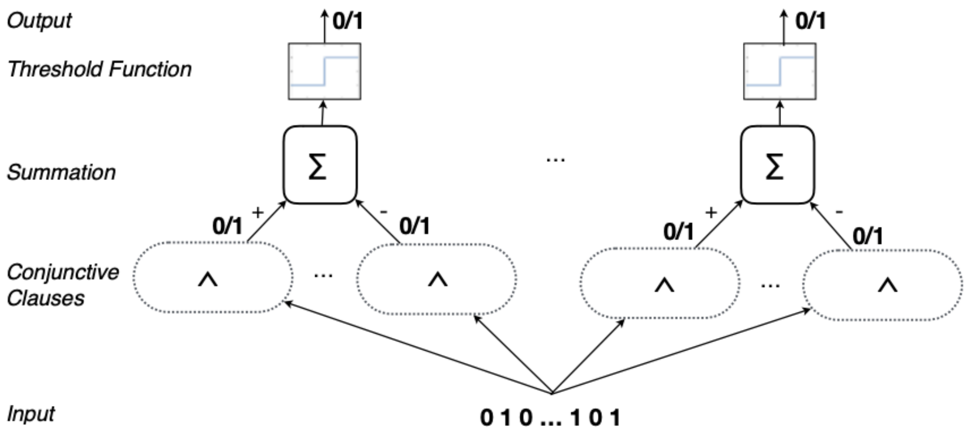


Figure 2.2: Inference structure of the Tsetlin Machine with clause polarity, vote calculation and a threshold function. Source: *The Tsetlin Machine – A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic* [8]

Unlike the neural network, which relies on back-propagated error to learn patterns, the Tsetlin Machine performs clause mining on binarized inputs and utilizes a feedback loop to update the states of the automaton to facilitate learning behaviour. The behaviour of each individual automaton is used to form clauses, generally represented in disjunctive normal form, allowing the Tsetlin Machine to represent complex non-linear patterns as a product of literals. The clauses may be treated as individual Tsetlin Machines voting on the presence of a sub-pattern. A two-step feedback system incorporated into the learning algorithm decides whether to include or exclude votes, based on the truth value of the clause and the truth value of the literal (Type I and Type II feedback)[8]. An overview of the transition probability definitions in the feedback mechanism is shown in Tables 2.1 and 2.2. The probability values are computed based on the specificity (s) of clauses mined (programmatically, this is implemented as a user defined parameter).

Table 2.1: Type I Feedback, as designed for the Classifying Tsetlin Machine Game[8]

	Truth value of Clause C_j^{i+}	1		0	
	Truth value of Clause l_k	1	0	1	0
Include Literal $l_{k_j}^{i+}$	P(Reward)	$\frac{s-1}{s}$	NA	0	0
	P(Inaction)	$\frac{1}{s}$	NA	$\frac{s-1}{s}$	$\frac{s-1}{s}$
	P(Penalty)	0	NA	$\frac{1}{s}$	$\frac{1}{s}$
Exclude Literal $l_{k_j}^{i+}$	P(Reward)	0	$\frac{1}{s}$	$\frac{1}{s}$	$\frac{1}{s}$
	P(Inaction)	$\frac{1}{s}$	$\frac{s-1}{s}$	$\frac{s-1}{s}$	$\frac{s-1}{s}$
	P(Penalty)	$\frac{s-1}{s}$	0	0	0

Table 2.2: Type II Feedback, as designed for the Classifying Tsetlin Machine Game[8]

	Truth value of Clause C_j^{i+}	1		0	
	Truth value of Clause l_k	1	0	1	0
Include Literal $l_{k_j}^{i+}$	P(Reward)	0	NA	0	0
	P(Inaction)	1	NA	1	1
	P(Penalty)	0	NA	0	0
Exclude Literal $l_{k_j}^{i+}$	P(Reward)	0	0	0	0
	P(Inaction)	1	0	1	1
	P(Penalty)	0	1	0	0

Since the initial advent of Tsetlin Machines in 2018, a few variants of the learning algorithm have been created, enabling the Tsetlin Machine to perform both regression as well as classification tasks. In the following sections, we will discuss three of these variants.

2.1 Classification Tsetlin Machine

The clauses mined by the Tsetlin Machine can be utilized for classification purposes, by utilizing an *argmax* function instead of the threshold. The output of the Multi-Class Tsetlin Machine (as in figure 2.3 is given by

$$\hat{y} = argmax_{i=1,\dots,m}\left(\sum_{j=1}^{n/2} C_j^{i+}(X) - \sum_{j=1}^{n/2} C_j^{i-}(X)\right) \tag{2.2}$$

where m is the number of distinct classes of objects specified in the problem and the $+$ and $-$ symbols indicate the clause polarity (include literals and exclude literals).

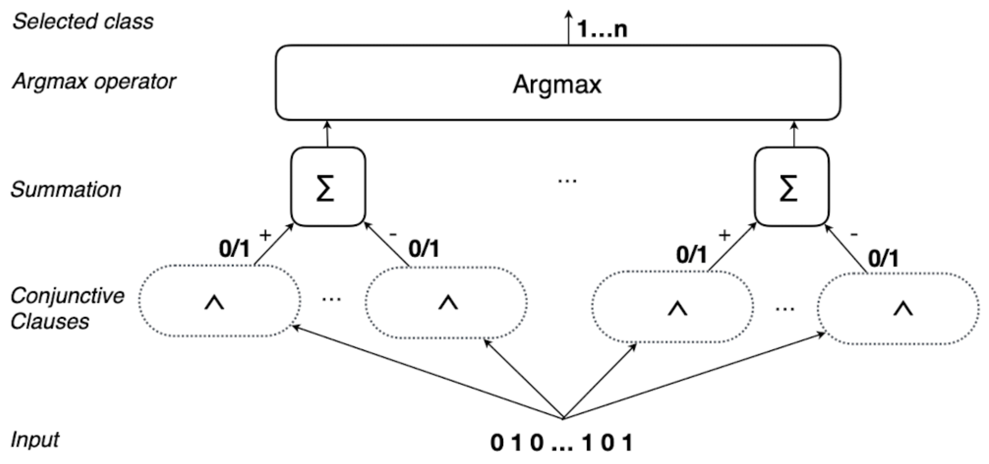


Figure 2.3: Multi-Class Tsetlin Machine. Source: *The Tsetlin Machine – A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic* [8]

The Classification Tsetlin Machine(CTM) has been shown to be effective compared to standard methods in the Iris classification [21] task, the MNIST hand-written digit classification [22] and the Binary Digits [21] tasks.

2.2 Regression Tsetlin Machine

In order to enable the Tsetlin Machine to compute real valued outputs, we remove the clause polarity that we were using to include and exclude literals[11]. We then utilize all the clauses to vote, based on the inputs. The votes are then mapped to a continuous output using Equation 2.3.

$$\hat{y_o} = \frac{\sum_{j=1}^{m} C_j(X_o) \times y_{max}}{T} \tag{2.3}$$

- $\hat{y_o}$ is the predicted output for the o^{th} input
- X_o is the io^{th} input
- $C_j(X_o)$ are the clauses mined for the o^{th} input
- T is voting threshold (programmatically, this is implemented as a user defined parameter)

In the CTM, learning was facilitated by increasing the number of votes for a class using Type I feedback when the number of clauses is less than the threshold necessary for classification, and conversely reducing the number of active clauses using Type II feedback when the predicted class is incorrect. We update the feedback mechanism to Equation 2.4 for the Regression Tsetlin Machine, thus allowing the clause updates to be performed for continuous outputs.

$$Feedback = \begin{cases} \text{Type I} & \text{if} \quad \hat{y_o} < y_o \\ \text{Type II} & \text{if} \quad \hat{y_o} > y_o \end{cases} \tag{2.4}$$

The clause activation probability is also updated, with the probability of feedback being directly proportional to the error[11], and the activation probability function P_{act} is defined as in Equation 2.5.

$$P_{act} = \frac{K \times |\hat{y_o} - y_o|}{y_{max}} \tag{2.5}$$

The constant K is a scaling factor which adjusts the magnitude of the activation function, preventing severe oscillation between predictions.
The RTM thus takes binarized inputs and converts them into continuous valued outputs using the voting mechanism and normalization equation shown above. This behaviour iis what we will utilize in our architecture for Reinforcement Learning using Q-value approximation.

2.3 Integer Weighted Tsetlin Machine

The CTM and RTM perform clause mining on binarized inputs. If the input space is large, the CTM and RTM require a large number of clauses to represent the inputs. This would result in slower computations and loss of interpretability. [12] introduced a novel approach to represent the patterns in a compact manner using weighted clauses. The Integer Weighted RTM (RTM-IW) uses stochastic searching on the line, in tandem with the feedback mechanism defined in Equation 2.4. This preserves the interpretability, since the weights increase when an automaton receives Type I feedback and decrease when the automaton receives Type II feedback, instead of activating more clauses. The compact representation also results in a lower memory footprint and faster computation times. The regression computation discussed in Equation 2.3 is updated to use the weighted sum, instead of the sum of clauses, as in Equation 2.6.

$$\hat{y_o} = \frac{1}{T} \sum_{j=1}^{m} w_j C_j(X_o) \tag{2.6}$$

In their research, [12] show that the RTM-IW performs well with artificial datasets, requiring fewer clauses than an RTM to learn representations on the same datasets.

In Chapter 3, we will discuss some literature on Reinforcement Learning, focusing on the methods we utilize in our research.

Chapter 3

Reinforcement Learning

Reinforcement Learning (RL) refers to a set of problems and solution methods, in which agents learn to maximize a reward from interacting with the environment through the exploration of possibilities and exploitation of gained knowledge [6]. Generally, RL problems involve sequential decision making, such as in games, or the continuous control of robots, where decisions are made to solve a part of a problem at each time step, and each of these steps then contribute to solving a larger problem or reaching a predefined goal. Problems are formulated as a finite Markov Decision Process (MDP), such as the one shown in Figure 3.1:

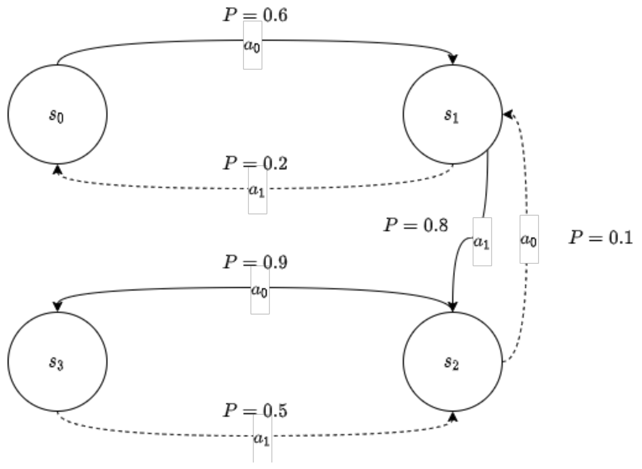


Figure 3.1: Simple 4 state, 2 action Markov Chain. The transition probabilities between connected states are given for each state-action pair. Dotted lines represent penalties, solid lines represent reward obtained from a transition.

- S: Set of states the agent receives as inputs
- A: The set of actions the agent can take in the environment
- $R(s,a)$: The reward function, reward is calculated given a state s and an action a taken

The agents learn to maximize obtained rewards over time through an RL algorithm and an efficient exploration strategy. The information obtained from the agent's interactions with the environment can be processed using one of the following approaches [2].

Model-based Learning

Model-based algorithms utilize planning methods to determine optimal actions given a model for a fixed window of time. The planning is facilitated by the knowledge of the dynamics of the RL problem, that is, the state transition probabilities and the associated reward function. Over many agent-environment interactions, the algorithm aims to reduce the errors in the estimation of future steps.

Model-free Learning

Model-free learning algorithms do not depend on knowledge of the dynamics of the problem, instead they approximate a policy which determines how the agent behaves given the current state. Model-free methods explore the environment until a preset end criteria is met. Repeated exploration allows the agent to refine the behavioural policy and maximize rewards obtained.
Existing model-free methods for solving RL problems utilize one of the following approaches to learning:

- Value-Function Methods:
  Assume an agent follows a control policy π in a reinforcement learning environment, the value function of a state s for such an agent is defined as the expected reward for the agent when it starts at s, and all actions are defined by the policy π for a fixed time frame.

  $$V^{\pi}(s) = E[R(s,a)|s,\pi] \tag{3.1}$$

  An optimal value function, V^* can thus be defined as follows:

  $$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{3.2}$$

  Value function methods maintain a table of the values of a state given a control policy π, generated by repeated action sampling over mini-batches of interactions with the environment. All optimal policies share the same optimal value function [6], thus, optimizing the value function should give us an optimal policy. Q-learning [17], SARSA [23] and Temporal Difference learning [24] algorithms utilize value function methods to determine an optimum control policy.

- Policy Search Methods:
  Let us assume an agent follows a control policy, π, which is a function mapping a state s to the probabilities of choosing an action a. An optimum policy is one which, given a state, takes an action that maximizes the reward obtained by the agent over a defined time frame.

  There are two popular approaches to compute an optimal policy:

  - *Policy Gradient Methods*:
    Policy gradient methods involve mapping finite-dimensional parameter vectors to the policy space, in a stochastic setting, followed by the use of gradient ascent methods on the noisy estimate of gradients in the parameter space. Such methods are highly popular in the context of robotics [25]. Examples of Policy-Gradient Methods include PEGASUS [26], Proximal Policy Optimization [27] and Vanilla Policy-Gradient Networks [28].

- *Gradient-Free Policy Search*:
  Direct policy search involves the use of metaheuristics to approximate the global optimum for an optimization problem, such as the optimum policy function. These methods iteratively update the policy based on samples and the value of a cost function that is specific to the problem and the metaheuristic used. Some implementations of gradient free methods are the Covariance Matrix Adaptation [29] and the Cross-Entropy Method [30].

For the scope of this research, we choose to focus on one specific model-free method, Q-Learning [17].

3.1 Q-learning

Q-Learning is an off-policy, value-based, model-free RL method. The algorithm is based on the concept of Temporal Difference (TD) learning, computing the best action to take in a given state and continuously updating estimates of the value-function based on the history of states visited and actions taken. Thus, the algorithm maps a state-action pair to a "quality" value in the real number space, as in 3.3.

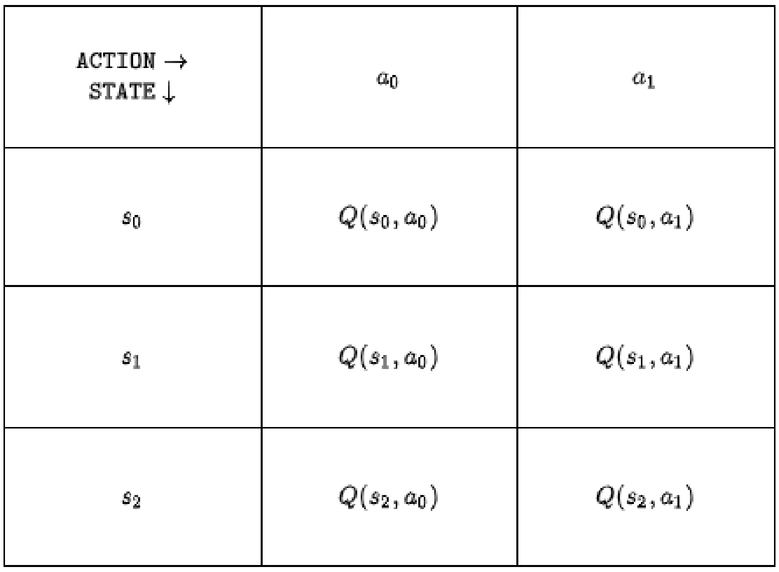
$$Q : S \times A \to \mathbf{R} \tag{3.3}$$

Specifically, an agent utilizing the Q-learning algorithm estimates the Q-value of a state using Equation 3.4.

$$Q^*(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{3.4}$$

- s_t is the state at time t
- a_t is the action taken at time t
- r_t is the reward obtained from performing action a_t on state s_t
- s_{t+1} is the state at time $t+1$, resulting from taking action a_t on state s_t
- Q^* is the updated Q-value
- α is the learning rate
- γ is the discount factor
- $\max_a(Q(s_{t+1}, a)$ is the estimate of the optimum future value

This approach differs from Monte Carlo methods in the frequency of updates to the estimates of the value function. The Monte Carlo method only updates the value function at the end of each episode, whereas in TD methods, the value function estimate is updated using known history at every step.



ACTION → STATE ↓	a_0	a_1
s_0	$Q(s_0, a_0)$	$Q(s_0, a_1)$
s_1	$Q(s_1, a_0)$	$Q(s_1, a_1)$
s_2	$Q(s_2, a_0)$	$Q(s_2, a_1)$

Figure 3.2: Sample Q-value table for a 3 state, 2 action environment.

3.1.1 Tabular Q-learning

Initial approaches to Q-learning utilised a Q-table, which mapped state-action pairs to Q-values. When the agent visited a state and performed an action, the Q-value for that pair was updated in the table. The simplest representation of a Q-value table is shown in Figure 3.2.
As is evident, this approach would not scale well for tasks with a large number of states and/or actions. Tabular approaches work well for limited state and action environments, like Tic-Tac-Toe or small scale maze problems. For problems on a larger scale, it would be more effective to construct a function that estimates the Q-value for each state-action pair without explicitly storing the values in memory, as we see in the following subsection.

3.1.2 Deep Q Networks

In their work on human level control for Atari game environments [31], the team at DeepMind utilized a convolutional neural network to estimate the Q-values for classic Atari games. With this architecture, an agent continuously updates a Q-value estimate over multiple episodes of the game, improving the estimation function using back-propagation. Deep Q-networks (DQNs) have since been the state-of-the-art in playing video games such as Atari Pong, Breakout and Space Invaders, inspiring many extensions, such as the double DQN [32], DQVN [33] and Actor-Critic Methods [34, 35].
Simpler approaches to DQN may utilize a Multi-Layer Perceptron (MLP) to compute the Q-values depending on the nature of the inputs presented [36]. For classical environments such as CartPole, a simple 3-layer MLP is sufficient to approximate the Q-values. The MLP only acts as an approximator, and is trained on the experience gained by the agent over multiple episodes of interacting with the environment.

3.2 Explore or Exploit: Epsilon-greedy methods

In a dynamic environment, it is important to make decisions using what we already know (*experience*) and knowing when we should update our knowledge (*exploration*). In terms of RL agents, knowing when to explore and when to rely on experience is important for convergence of the learning process and stability of agent behaviour. Classically, we use ε-greedy (Epsilon greedy) methods and the stochasticity of the system to decide when to explore and when to rely on experience. Algorithm 1 explains the ε-greedy method, specifically for Q-learning.

Algorithm 1 ε-greedy method applied to Q-learning

```
Require:
  $\varepsilon \in (0,1)$: Exploration Probability
  $Q(s,a)$: Q-value table
  while not terminal state do
  Generate random number $rand$
     if $\varepsilon \geq rand$ then
        Take a random action from action space $\mathbf{A}$          ▷ Explore possibilities
     else
        Take action based on last $Q(s,a)$ estimate          ▷ Exploit learned behaviour
     end if
  Update $Q(s,a)$ estimates
  end while
```

The ε-greedy approach works well for a small environment, but in a larger environment where we move from tabular methods to function approximation, we need to utilize more than just ε-greedy searches in order to converge to optimal agent behaviour.

$$\varepsilon_{n+1} = \varepsilon_{max} \cdot \Delta\varepsilon^{n} \tag{3.5}$$

$$\varepsilon_{n+1} = 1.1 - \frac{1}{cosh(e^{-\frac{n-\alpha\cdot n_{max}}{\beta\cdot n_{max}}}) + \frac{n\cdot\delta}{n_{max}}} \tag{3.6}$$

We tested two slightly different methods to compute the decay of ε with each episode. The simplest of these is the Exponential decay function shown in Equation 3.5. This simple function has a uniform decay rate which quickly reaches the minimum cap of 0.01 quite quickly, restricting the exploration capability of the agent. Hence we decided to utilize the Stretched Exponential Decay (SED) function for our agents, since it provides greater exploration over the same number of episodes as compared to the Exponential decay function. The SED function is as given in Equation 3.6. The parameters α, β and δ, all belonging to the interval $[0,1]$ control the following aspects of the decay curve:

- α controls the time between exploration and exploitation. Values above 0.5 force high exploration.
- β controls the slope of the transition between exploration and exploitation.
- δ controls the steepness of the left and right tails of the decay curve.

From our initial set of experiments, we notice that the values $\alpha = 0.4$, $\beta = 0.3$ and $\delta = 0.3$ gave us a good trade-off between exploration and exploitation for our Tsetlin Machine agents.[1]

[1] A sample plot comparing the decay functions over 1000 episodes is shown in the Appendix.

3.3 Experience Replay

When using a non-linear approximator, such as a neural network, to estimate the Q-value, we notice the preservation of correlations of historic state-action pairs, that is, the network learns too much from its immediate history, resulting in errors in Q-value estimation. In order to tackle this, [31] suggests using a biologically inspired feedback system, which uses random subsets of the agent's history to update its expectations of Q-values. This technique, called Experience Replay, reduces the correlations between historic and current states, preventing large unnecessary updates to the estimation function.

3.3.1 Prioritized Experience Replay

Experience Replay classically uses randomized samples of the agent's history to update the agent's behaviour. This stochastic sampling may lead to certain important state-action pairs being skipped in a limited number of training cycles, affecting the net change in the agent behaviour and hence convergence. A variation of the Experience Replay method which uses utilizes a weighted sampling method was introduced in [37]. In Prioritized Experience Replay (PER), the learning efficiency is improved through repeatedly learning from the more common transitions, using the TD-error of a transition to measure its priority. In their work [37], the authors showcase this approach to be much more effective in the training of DQN agents on Atari games.

The above discussed methods form the base of the implementation we are building with the Regression Tsetlin Machine. The general pipeline for Q-learning remains the same, with the RTM being used as the Q-value approximator which decides agent behaviour instead of the neural network.
In Chapter 4, we discuss the implementation of a Regression Tsetlin Machine based Q-learning agent, utilizing classical Experience Replay as well as PER.

Chapter 4

Regression Tsetlin Machine for Q-Learning

In this chapter, we will lay out the architectural changes made to the RTM to facilitate Q-learning. As mentioned in Chapter 2, the Tsetlin Machine works on binarized inputs, hence we will also visit some input binarization schemes we tested.

4.1 Considerations while Utilizing the RTM for Reinforcement Learning

The architecture and normalization equation for the RTM have been discussed in Chapter 2. This architecture is suitable for supervised learning from a labelled dataset. Since the most common RL environments involve continuous inputs, such as sensor readings or vector values (in simulators), we first tested a few input binarization methods. We also had to update the normalization equation to utilize expected Q-value ranges as the limits. The expected Q-value ranges are environment specific, since they depend on the reward and the number of steps in an episode of learning. We will now discuss in detail the changes that were made to individual components.

4.1.1 Input discretization techniques

As our baseline environments, since this research aims to implement a simple RTM based RL agent, we chose the classic environments from OpenAI Gym[38], specifically CartPole and MountainCar, partly due to their popularity at being baseline environments and partly because of the simplicity of inputs provided. These environments represent states as a vector of continuous values (input length being 4 for CartPole and 2 for MountainCar). These inputs need to be discretized for the RTM. Since it is difficult to directly represent floating point numbers in binary format, we built a custom discretizer.
Based on our tests, we found that a simple unsigned binarization scheme that divided the input space into a fixed number of bins and assigned 1 to the bin to which the number belonged worked best, since it had lower noise compared to some of the other schemes we tested.[1]

[1]A complete list of the binarization algorithms we tested are given in the Appendix.

4.1.2 Q-value Range Computation

Utilizing the Bellman update equation (Equation 3.4), we can compute the range of Q-values that the agent needs to normalize the votes to. Applying the summation of infinite terms to this equation, we get the following:

$$Q \in \left[r, \alpha \cdot r \cdot \frac{(1-\gamma^n)}{(1-\gamma)} \right] \tag{4.1}$$

n is the maximum number of steps in an episode.
We will utilize this range when computing the Q-value using the RTM agent. In the case of certain games, the maximum and minimum values are interchanged, if the reward for each timestep is negative (such as for MountainCar, where the reward is -1 for each timestep).

4.1.3 Q-learning with the RTM

The RTM is modified and combined with the Q-learning framework. The salient points of our architecture are as follows:

- We have one RTM agent per action, each agent computes the Q-value for a given state and the action index that it is associated to.
- We utilize the experience-replay method to train the agents and update the ε value to control exploration.
- We introduce a binarization step before the agent is fed an input to compute the Q-value from.

Algorithm 2 Q-learning with Regression Tsetlin Machine

```
Require:
  $n\_actions$: Number of actions in the environment
  $bin\_scheme$: The input binarization scheme
  Initialize RTM agent with $n\_action$ sub-agents.
  Initialize replay memory $Mem$
  Initialize action-value function $Q$
  for episode $\in [0, ep_{max}]$ do
      Initialize input sequence $s_1 = x_1$
      Binarize state $\phi_1 = \phi(s_1)$
      Initialize step counter $t$
      while not terminal state do
          Generate random number $rand$
          if $\varepsilon \geq rand$ then
              Take a random action from action space A                         ▷ Explore possibilities
          else
              Take action based on $max(Q(\phi(s_t), a = 0), \dots, Q(s, a = n\_actions))$   ▷ Exploit agent
              Execute $a_t$, observe reward $r_t$ and state $s_{t+1}$
              Binarize and commit $(\phi(s_t), a_t, r_t, \phi(s_{t+1}))$ to memory $Mem$
          end if
          Increment $t$
      end while
      Sample random batch from memory
      Update $Q(s, a)$ estimates for all $n\_action$ agents from memory states
      Update $\varepsilon$ value
  end for
```

Chapter 5

Experimental Setup

As discussed in Chapter 4, we utilize two of the most popular classical mechanics environments from OpenAI Gym [38], CartPole and MountainCar. We will discuss the structure of these environments briefly and follow it up with our experimental structure and performance criteria.

5.1 Environments

5.1.1 CartPole

The CartPole environment, designed based on the work of Barto et al. [39], is a 2-action simple environment. The environment consists of a pole attached by a hinge to a cart moving on a horizontal track, as in figure 5.1. The agent controls the system by applying a force of ± 1 to the cart (symbol signifies direction of applied force), moving the cart right or left (2 actions). At the beginning of an episode, the pole is perfectly upright. The goal of the game is to prevent the pole from deviating more than 15 degrees from the vertical. The agent receives a reward of $+1$ for each timestep when the pole remains within ± 15 degrees of the vertical. In order to limit the length of an episode, we consider more than 195 timesteps without the pole toppling as a win.

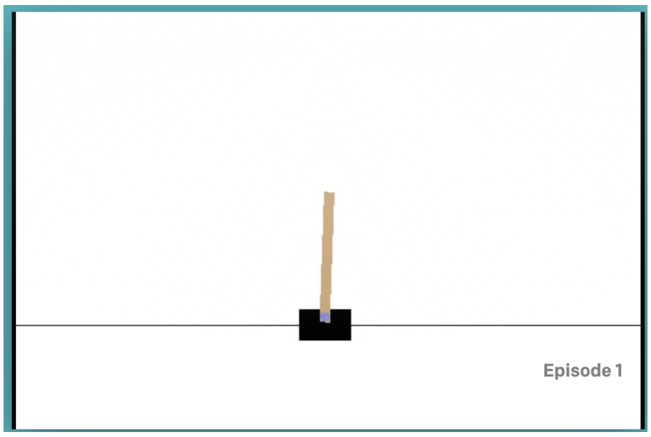


Figure 5.1: The CartPole environment as in OpenAI gym. [38]

As inputs, we are given 4 features as in Table 5.1. Over the course of our experiments, we noticed that the distribution of values during an actual run belong to a subset of the architectural range of these features as speciified in the package. Table 5.1 shows the values we used in order to restrict the bins for binarizing these features.

Feature	Architectural Range	Experimental Range
Cart Position	[-4.8, 4.8]	[-1.5, 1.5]
Cart Velocity	$(-\infty, \infty)$	[-100, 100]
Pole Angle	[-24, 24]	[-15, 15]
Pole Angular Velocity	(∞, ∞)	[-100, 100]

Table 5.1: Range of features for the CartPole environment. Architectural limit is the limit specified in [39]. Experimental range is the limited range of inputs based on experiments.

5.1.2 MountainCar

The MountainCar environment, designed by Moore [40], is a 2-action simple environment with negative rewards. The goal of the game is to move a car from a valley to the top of a mountain (denoted by the flag in figure 5.2). However, the car requires to build enough momentum to climb the mountain, through repeated oscillation between the left and right slopes of the mountain. The reward mechanism for this game is different from the previous environment, with the agent receiving higher rewards for achieving the c=goal in the least amount of steps.

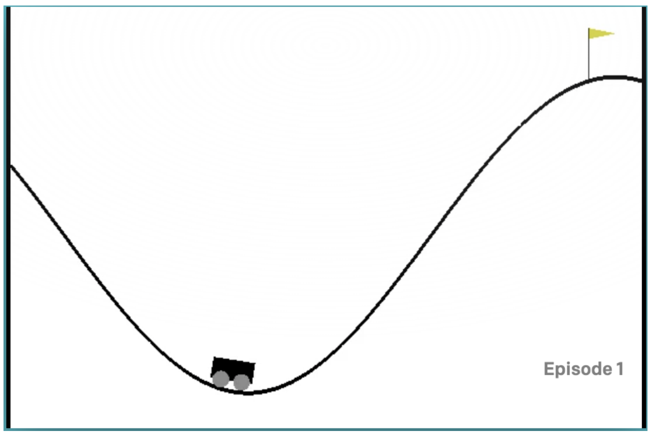


Figure 5.2: The MountainCar environment as in OpenAI gym. [38]

As inputs, we are given 2 features, as in Table 5.2. As with the CartPole environment, we noticed while experimenting that the distribution of values during an actual run belong to a subset of the architectural range of these features. Table 5.2 shows the values we used in order to restrict the bins for binarizing these features.

Feature	Architectural Range	Experimental Range
Car Position	[-1.2, 0.6]	[-1.2, 1.2]
Car Velocity	(-0.7, 0.7)	[-0.1, 0.1]

Table 5.2: Range of features for the MountainCar environment. Architectural limit is the limit specified in [40]. Experimental range is the limited range of inputs based on experiments.

5.2 Hyperparameters

In our initial experiments while constructing the algorithm, we tested out many combinations of the following hyperparameters necessary for the RTM agents:

- Specificity s: We tested values of specificity between 0.001 and 1000. From our initial experiments we noticed that $s = 5$ gave us reasonable results with respect to changes in Q-value over each iteration.
- Number of Clauses: We tested between 32 to 3000 clauses for our agents.
- Threshold T: We tested agents with $T =$ number of clauses$/2$ and $T = 1$.

Combinations of the above hyperparameters were tested within the experimental framework that we designed. The results and analysis are detailed in the subsequent chapter.

5.2.1 Motivation for Threshold and Specificity Values

When we deconstruct the Regression Tsetlin Machine, we see that the Threshold value is a control on the number of votes that contribute to an output. We opted to test the default setting which is Number of clauses$/2$ and also 1, since setting the Threshold to 1 implies that all clause votes are utilized to compute the Q-value.
The Specificity controls the fineness of clauses learnt. Ideally, this would mean higher is better, hence we initially chose really high values of specificity. We then contrasted agent behaviour at low Specificity values, keeping all other hyperparameters constant. This allowed us to narrow down values of Specificity that enabled learning behaviour.

5.3 Performance Criteria

In order to measure individual agent performance in each game, we will utilize the metrics provided by OpenAI gym to compute the win rates of agents. The desired goal is for the agents to achieve similar performance as compared to a Multi-Layer Perceptron (MLP) agent in the same number of episodes.

We will also take a note of computation time necessary for the agents in order to compare the RTM approaches to the MLP agent. We would ideally like the RTM agents to take lesser time as compared to the MLP agent for the same number of episodes.

5.4 Experimental Configurations

We have a total of 5 algorithms that are each being tested on 2 environments. We consider the agents using the MLP to approximate Q-values as the baseline for both environments. We will then compare the Regression Tsetlin Machine agents (vanilla RTM, RTM with Prioritized Experience Replay, Integer Weighted RTM, Integer Weighted RTM with Prioritized Experience Replay) for a range of hyperparameters. We run five individual trials for each hyperparameter setting with different initial seed values for consistency. The results reported for each hyperparameter setting are average and deviation of the scores accumulated by the agents in the 5 trials.

5.5 Implementation

All the simulations for our experiments were done on Python3 [41]. Baseline experiments using a Multi-Layer Perceptron agent were programmed with Keras [42]. The Tsetlin Machine agents are slightly modified variants of the Regression Tsetlin Machine available on the CAIR GitHub repository [43, 12].
All experiments were carried out on the Peregrine HPC and Pallas AI systems at the University of Groningen. The experiments were exclusively programmed to compute on CPU.

Chapter 6

Results

After initial experimentation to build the agents and test a few hyperparameters, we created group experiments with 5 fixed seed values. We ran the RTM based agents on each environment with common hyperparameters. We also ran sets with MLP agents, in order to create a baseline. The most important results from each environment are discussed below, a list of the other experiments can be found in the Appendix.

6.1 Cartpole

The cartpole environments provides an input vector of four floating point numbers, which we converted into binarized form. We tested different binarization schemes, starting with 4 bins per input (an input length of 16 bits) to 16 bits per input (an input length of 64 bits). We noticed in doing so that the binarization range can be shortened to the Experimental ranges mentioned in Table 5.1. Fixing these values gave us better results as compared to an agent learning form the entire input space. Once the input ranges were fixed, we tested out different binarization schemes. The best performing scheme, Unsigned binarization (see Algorithm 3), was then utilised for our group experiments. We then tested out different clause lengths, Thresholds and Specificity values for each of the 4 RTM agents. For all experiments, we utilized binarized input vectors of length 32.

6.1.1 Comparison of Architectures

In our tests we noticed that the simple RTM agent required atleast 2000 propositional clauses to achieve high scores. We obtained the best results with an agent using 2000 clauses, with a Threshold of 1 and a Specificity of 5, as shown in Figure 6.1.

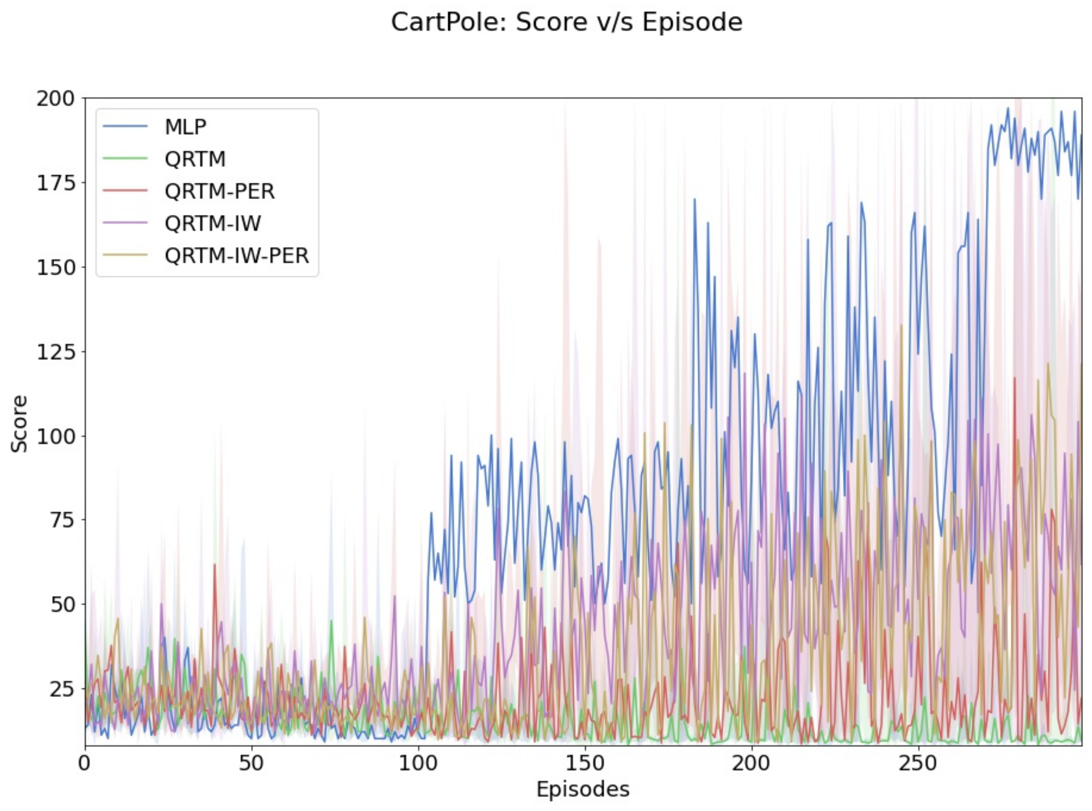


Figure 6.1: Performance of the best RTM agents as compared to the MLP on CartPole environment. Average and deviation of the scores depicted in graph for RTM agents. ($n\_clauses = 2000, T = 1, s = 5$)

The agent behaviour is not stable, and increasing the number of clauses did not improve agent behaviour. The value of Specificity controls the granularity of the clauses mined by the Tsetlin Automaton. Ideally, this would mean that the higher the value of s, the better the clause mining process, but in our experiments, we noticed that high values of s caused the learning to stagger. This could be attributed to the automaton resetting clauses repeatedly, whenever minor variations within the clauses are viewed while training.
Increasing the Threshold ($T = n\_clauses/2$) resulted in the limiting the agents learning, since that would be the same as using only half the total number of clauses to compute the same range of continuous values. For the best agents, we set the Threshold T to 1, utilizing the entire clause space to compute Q-values.
The Integer Weighted RTM agent required fewer number of clauses to achieve similar learning. We noticed peak performance at 1000 clauses, as in Figure 6.2. This is due to the condensed clause notation that the Integer Weighted RTM utilized. Increasing the number of clauses did not lead to any significant improvement for the Integer Weighted RTM agent, since some weights were seen to be set to zero after multiple episodes of the game.

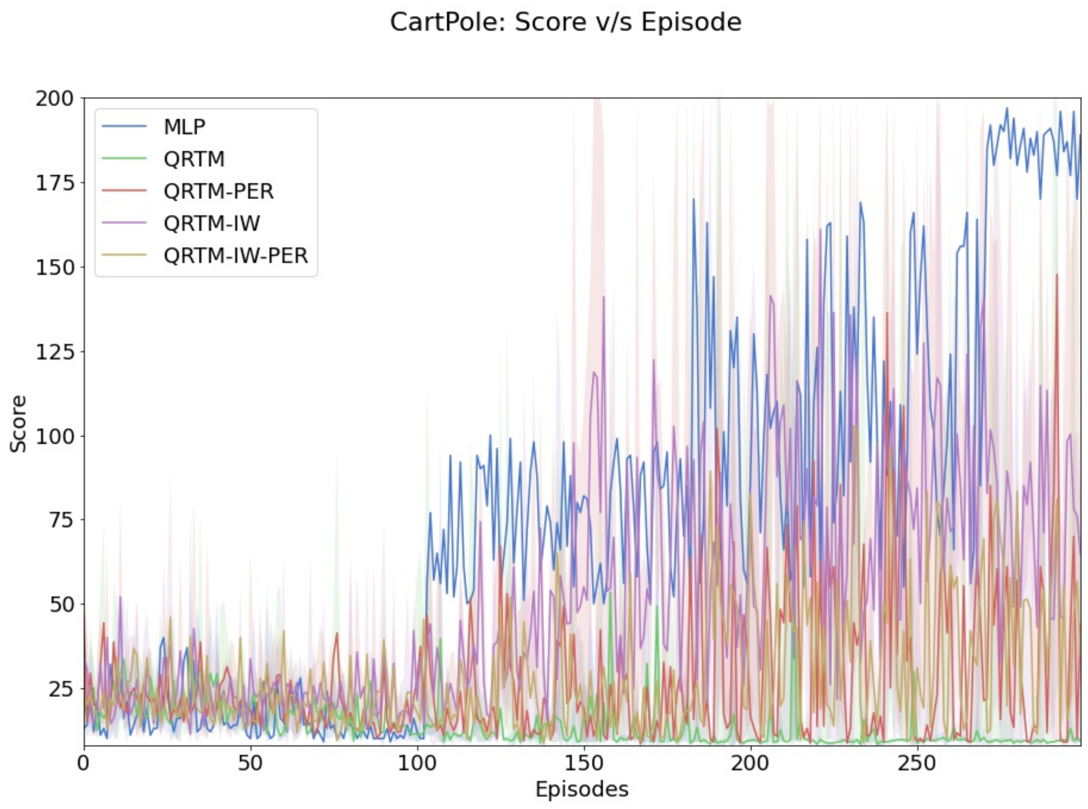


Figure 6.2: Performance of the RTM agents as compared to the MLP on CartPole environment. Average and deviation of the scores depicted in graph for RTM agents. ($n\_clauses = 1000, T = 1, s = 5$)

Agent Type	**Memory Type**	**Number of Clauses**	**Threshold**	**Specificity**	**Win Rate**
RTM	**ER**	2000	1	5	10%
RTM	**PER**	2000	1	5	12%
RTM	**ER**	2000	1000	5	0%
RTM	**PER**	2000	1000	5	0%
IW-RTM	**ER**	1000	1	5	20%
IW-RTM	**PER**	1000	1	5	20%
IW-RTM	**ER**	1000	500	5	2%
IW-RTM	**PER**	1000	500	5	2%

Table 6.1: Performance of selected RTM agents on the CartPole environment for different hyperparameter configurations.

Between the two memory methods, we notice that agents using Prioritized Experience Replay have much better win rates as compared to agents using simple Experience Replay, as can be seen in Table 6.1.

6.2 Mountain Car

The MountainCar environment is more complex than the CartPole environment due to the sparsity of rewards and the continuous cost of movement. As with the CartPole environment, we performed a set of initial experiments to narrow the input range for the binarizers as in Table 5.2. We utilise the same binarization scheme as we had for CartPole, with inputs ranging from 16 bits to 64 bits per state (each state is a vector of size 2). For all experiments, we used a binarized input vector of length 32. A complete set of the results with the different hyperparameter configurations tested is given in the Appendix (Table 3). The results for the MountainCar environment suggest that the current clause update mechanism is not ideal for extreme sparse reward environments.

6.2.1 Comparison of Architectures

With a simple MLP agent as baseline, we did not get the agent to learn the game. The behaviour of the RTM agents is similar to the baseline, with the agents not winning at this environment. A sample performance of the agents is as shown in Figure 6.3. The agents used 2000 clauses, with a Threshold of 1 and a Specificity of 5.

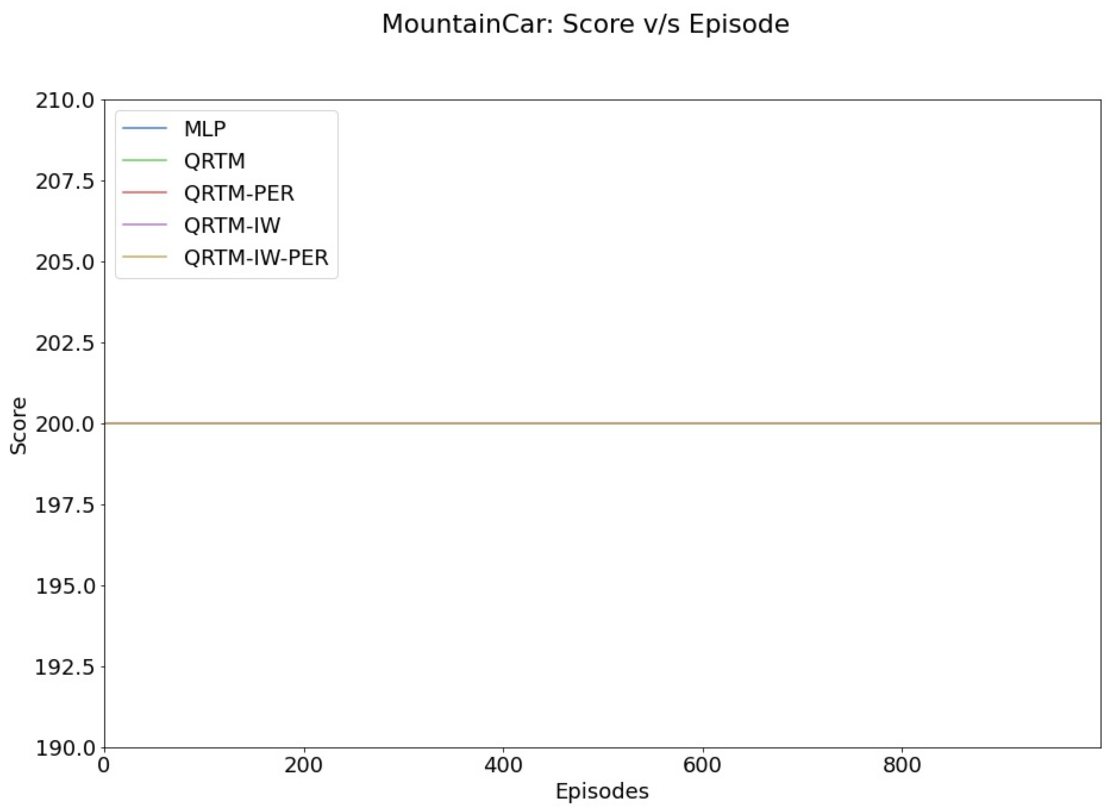


Figure 6.3: Performance of the best RTM agents as compared to the MLP on the MountainCar environment. Average and deviation of the scores depicted in graph for RTM agents. ($n\_clauses = 2000, T = 1, s = 5$)

We noticed that the agents always computed Q-values to be marginally different in the order of $1e-3$. The Q-values were almost always equal to the lowest reward possible from the game.

6.3 Clause Mining and Interpretability

Generally, Q-value approximation functions are non-linear. The complexity of the Q-value function is not easily translated into the domain of propositional clauses that the RTM utilizes to learn. With the approximation of complex functions requiring very large number of clauses, we do not achieve better interpretability of calculation of Q-values and hence the behaviour of agents. The Integer Weighted RTM agents are more promising in this aspect. However, we would need to address the issue with stability of agent behaviour before we can utilize the agents.
The clause mining behaviour is also not inherently interpretable. Initial analysis gives us the impression that the clause activation is random, and that the clause is just an internal representation that the RTM utilizes. However, this is not the case, since with a stable agent, we could link individual input features to clause activation.
In the next chapter, we will discuss the main contributions of our work, the shortfalls of the approach we took and some prospective directions for future research in the field.

Chapter 7

Conclusion

In this chapter, we will discuss the main points of our research. We will also discuss the reasons for some of the shortfalls that we see in the architectures we experimented with. We will conclude this section, and hence the report, with a discussion of possible future directions of research.

7.1 Summary of Main Contributions

How can we design a Reinforcement Learning system utilizing Regression Tsetlin Machines?

We started this research with an aim of building Regression Tsetlin Machine based Q-learning agents. We have successfully built 4 different RTM agents that are capable of playing games without supervision, learning solely from the signals received from the environments they interact with.

How does the performance of the Tsetlin Machine Reinforcement Learning agent compare with an agent trained using a Deep Q Network?

We tested the agents out on standard environments provided by OpenAI, CartPole and MountainCar. The CartPole agents show promise, with the agents able to win games, albeit not as continuously as Neural Network agents can. The sparsity of rewards in the MountainCar environment do not seem to enable learning behaviour in the RTM agents. In terms of computation, the RTM agents do take more time than their Multi Layer Perceptron counterparts. This is mostly because of the number of clauses required to approximate Q-values.

How understandable is the decision making process of the Tsetlin Machine Reinforcement Learning agent?

In terms of interpretability, we notice that our approach may not have been the best. As discussed earlier, representing complex functions using propositional clauses requires a large number of clauses, which detriments the interpretability of the agent.

7.2 Future Work

We shall now discuss in detail some of the issues we faced with the architecture of the RTM agents and how these issues may be addressed in future research.
The first issue is the binarization of continuous values to a discrete space, without losing the magnitude of values. Bin based binarization schemes are limited in the precision of magnitudes they may represent. We cannot utilize one-hot encoding style approaches directly in the case of unsupervised learning, since we do not have a dataset to learn from. The current approach to binarization introduces some noise to the system, which is detrimental to learning behaviour in such dynamic environments. The first step to utilizing the RTM would require better binarization schemes for dynamic environments, which are also interpretable in terms of mapping back to the continuous inputs provided by the environment.
The vanilla RTM agents require a lot of propositional clauses, which affect both the computation time as well as the interpretability of agent behaviour. Based on the limited results, we notice that this can be avoided by using a Integer Weighted RTM agent. However, we would also have to restrict the weight updates on the Integer Weighted RTM, since we noticed that in some cases, the weights were increased to the order of $1e4$. Also, we would need to perform more experiments which enable us to study the clause activation behaviour of RTMs in such environments, since without understanding this, we would not be able to improve the interpretability of the agents.
While analyzing the results, we noticed that the RTM needs a lot of clauses for learning non-linear functions like the Q-value approximation. We are also not sure how the propositional clauses map to the non-linear function. It would help to delve deeper into how these relate, since we could then understand the relation between the hyperparameters necessary for agent design. Being able to compute the minimum number of clauses necessary for approximating a function could allow for more robust agent design.
An alternative architecture, suggested by Glimsdal and Granmo [44], would allow us to combine commonly used clauses without replication, thus reducing the number of clauses necessary to represent a function. The Coalesced TM architecture would thus enable condensed representation of the propositional clauses, improving interpretability and agent robustness. A suggested approach would be to use the Coalesced TM to mine the most common clauses and fine tune the RTM architecture on top to have more precise computations of Q-values and hence more concise agent behaviour.
It is also suggestible to look into using alternate approaches to Reinforcement Learning, such as Actor-Critic agents with Classifier Tsetlin Machines, since the computation of Q-values may be avoided, removing a layer of abstraction. This may also perform better in complex environments such as MountainCar. The CTM does not abstract the inputs into a continuous value, instead choosing to convert the agent votes into a decision directly. The removal of one layer of abstraction could improve the overall interpretability of the system.
In our experiments, we have noticed a significant difference between the learning behaviour of the RTM agents on the MountainCar and CartPole environments. One of the reasons for this is the nature of rewards. While Cartpole generates a positive rewards for each timestep, the MountainCar environment is a punishment based environment, with a negative reward for each step. We tested out basic reward engineering approaches with a Markovian reward structure, but the punishment scheme seems to confuse the RTM agent, since it updates and resets the clauses frequently while training, losing patterns or learning erroneous patterns. Using a non-Markovian reward system could skew the learning behaviour of the automaton, since the approximation of the Q-function is different in case of non-Markovian rewards. We have tested out a wide range of hyperparameters for both environments, the complete results are tabularized in the Appendix. We also noticed that the simple neural network

agent we utilized in the MountainCar environments did not learn the game very well, which brings us to the conclusion that the function we approximate for MountainCar is more complex than the Q-value approximation function for CartPole. In extension, this means the RTM for MountainCar would need a very high number of clauses, which means very high computation costs and very low interpretability.
As a final note, with respect to the choice of hyperparameters, more experimentation is necessary. As mentioned earlier, we do not completely understand how many clauses are necessary to represent a function and what the relation between the number of clauses is to the length of the binarized input. The values of Specificity tested in our experiments were also limited, due to the high computation time for some of our environments.

7.3 Conclusion

In this thesis, we built a Reinforcement Learning agent utilizing Tsetlin Machines. We expected this method to be interpretable and less compute intensive as compared to neural networks, but the approach we utilized, combining the Regression Tsetlin Machine with Q-learning, did not give us expected results. However, we have seen that it is possible to represent the Q-value approximation functions using the RTM.
We have further explored the usage of different types of RTM agents on the two chosen environments and analyzed the performance of these agents. This has given us insight into the design of interpretable agents for the purpose of Reinforcement Learning. We also discuss some of the issues that have to be addressed in future iterations of research in order to improve agent stability and make the learning pipeline interpretable throughout.

Bibliography

[1] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal, "Explaining explanations: An overview of interpretability of machine learning," in *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*, pp. 80–89, IEEE, 2018.

[2] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[3] A. G. Barto, P. Thomas, and R. S. Sutton, "Some recent applications of reinforcement learning," in *Proceedings of the Eighteenth Yale Workshop on Adaptive and Learning Systems*, 2017.

[4] L. Tai and M. Liu, "Towards cognitive exploration through deep reinforcement learning for mobile robots," *arXiv preprint arXiv:1610.01733*, 2016.

[5] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *arXiv preprint arXiv:1610.03295*, 2016.

[6] R. S. Sutton, A. G. Barto, *et al.*, *Introduction to reinforcement learning*, vol. 135. MIT press Cambridge, 1998.

[7] M. Lauer and M. Riedmiller, "An algorithm for distributed reinforcement learning in cooperative multi-agent systems," in *In Proceedings of the Seventeenth International Conference on Machine Learning*, Citeseer, 2000.

[8] O.-C. Granmo, "The tsetlin machine-a game theoretic bandit driven approach to optimal pattern recognition with propositional logic," *arXiv preprint arXiv:1804.01508*, 2018.

[9] O.-C. Granmo, S. Glimsdal, L. Jiao, M. Goodwin, C. W. Omlin, and G. T. Berge, "The convolutional tsetlin machine," *arXiv preprint arXiv:1905.09688*, 2019.

[10] G. T. Berge, O.-C. Granmo, T. O. Tveit, M. Goodwin, L. Jiao, and B. V. Matheussen, "Using the tsetlin machine to learn human-interpretable rules for high-accuracy text categorization with medical applications," *IEEE Access*, vol. 7, pp. 115134–115146, 2019.

[11] K. Darshana Abeyrathna, O.-C. Granmo, L. Jiao, and M. Goodwin, "The regression tsetlin machine: A tsetlin machine for continuous output problems," *arXiv preprint arXiv:1905.04206*, 2019.

[12] K. D. Abeyrathna, O.-C. Granmo, and M. Goodwin, "A regression tsetlin machine with integer weighted clauses for compact pattern representation," *arXiv preprint arXiv:2002.01245*, 2020.

[13] A. M. Roth, N. Topin, P. Jamshidi, and M. Veloso, "Conservative q-improvement: Reinforcement learning for an interpretable decision-tree policy," *arXiv preprint arXiv:1907.01180*, 2019.

[14] R. M. Annasamy and K. Sycara, “Towards better interpretability in deep q-networks,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4561–4569, 2019.

[15] D. Lyu, F. Yang, B. Liu, and S. Gustafson, “Sdrl: interpretable and data-efficient deep reinforcement learning leveraging symbolic planning,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 2970–2977, 2019.

[16] E. Dağlarli, “Explainable artificial intelligence (xai) approaches and deep meta-learning models,” *Advances and Applications in Deep Learning*, p. 79, 2020.

[17] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[18] R. Livni, S. Shalev-Shwartz, and O. Shamir, “On the computational efficiency of training neural networks,” *arXiv preprint arXiv:1410.1141*, 2014.

[19] J. C. Gittins, “Bandit processes and dynamic allocation indices,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 41, no. 2, pp. 148–164, 1979.

[20] K. S. Narendra and M. A. Thathachar, *Learning automata: an introduction*. Courier corporation, 2012.

[21] D. Dua and C. Graff, “UCI machine learning repository,” 2017.

[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[23] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, vol. 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[24] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[25] J. Peters, S. Vijayakumar, and S. Schaal, “Reinforcement learning for humanoid robotics,” in *Proceedings of the third IEEE-RAS international conference on humanoid robots*, pp. 1–20, 2003.

[26] A. Y. Ng and M. I. Jordan, “Pegasus: A policy search method for large mdps and pomdps,” *arXiv preprint arXiv:1301.3878*, 2013.

[27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017.

[28] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” 2016.

[29] I. Szita and A. Lörincz, “Learning tetris using the noisy cross-entropy method,” *Neural computation*, vol. 18, no. 12, pp. 2936–2941, 2006.

[30] K. Wampler and Z. Popović, “Optimal gait and form for animal locomotion,” *ACM Transactions on Graphics (TOG)*, vol. 28, no. 3, pp. 1–8, 2009.

[31] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[32] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[33] M. Sabatelli, G. Louppe, P. Geurts, and M. A. Wiering, "Deep quality-value (dqv) learning," 2018.

[34] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *NIPS*, 1999.

[35] I. Grondman, L. Busoniu, G. A. Lopes, and R. Babuska, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1291–1307, 2012.

[36] S. Nagendra, N. Podila, R. Ugarakhod, and K. George, "Comparison of reinforcement learning algorithms applied to the cart-pole problem," in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 26–32, IEEE, 2017.

[37] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[38] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[39] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 834–846, 1983.

[40] A. W. Moore, "Efficient memory-based learning for robot control," 1990.

[41] G. Van Rossum and F. L. Drake, *The python language reference manual.* Network Theory Ltd., 2011.

[42] F. Chollet *et al.*, "Keras: The python deep learning library," *Astrophysics Source Code Library*, pp. ascl–1806, 2018.

[43] Cair, "cair/regression-tsetlin-machine: Implementation of the regression tsetlin machine."

[44] S. Glimsdal and O.-C. Granmo, "Coalesced multi-output tsetlin machines with clause sharing," 2021.

[45] K. D. Abeyrathna, O.-C. Granmo, X. Zhang, and M. Goodwin, "A scheme for continuous input to the tsetlin machine with applications to forecasting disease outbreaks," 2019.

Appendices

A Epsilon Decay Algorithms: Comparison

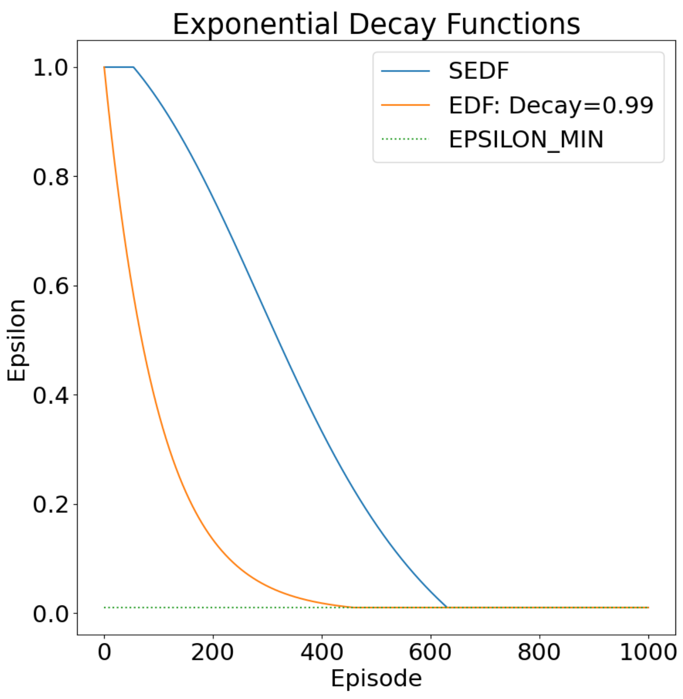


Figure 1: Comparison of epsilon decay functions over 1000 episodes.

Figure compares the Exponential Decay function (ED) with $\Delta\varepsilon = 0.99$ and the Stretched Exponential Epsilon Decay function (SED) with $\alpha = 0.4$, $\beta = 0.3$ and $\delta = 0.3$. We notice that the ED decays to ε_{min} in under 40% of the total number of episodes, leaving a majority of episodes to utilize exploitation. The SED decays slowly, providing more exploration, ensuring that the agent continues to learn from experiences until late into the game.

B Input Binarization Algorithms

As part of our initial experiments, we tested out 4 different binarization techniques:

- Simple Binning Binarizer with Sign Bit
- Unsigned Binarizer
- Greater than Binarizer with Sign Bit
- Less than Binarizer [45]

The Less than and Greater than binarizers provide dense inputs, whereas the other schemes return sparse binarized inputs. In our experiments, we found that the Unsigned Binarizer performed best, hence we utilize the Unsigned binarizer in all reported results.

Algorithm 3 Unsigned Binning Binarizer

```
Require:
  $input$: Floating point feature
  $n\_bins$: Number of bins the input range is divided into
  $range\_max$: Maximum of input feature range
  $range\_min$: Minimum of input feature range
  Initialize $binary\_rep$ array of size $n\_bins$
  Compute $bin\_delta = (range\_max - range\_min)/n\_bins$
  for $i \in [0, n\_bins]$ do
  $min\_bin = range\_min + i \cdot bin\_delta$
  $max\_bin = range\_min + (i+1) \cdot bin\_delta$
      if $min\_bin \leq abs(input) \leq max\_bin$ then
          $binary\_rep[i] = 1$
      end if
  end for
  return $binary\_rep$
```

Algorithm 4 Simple Binning Binarizer with Sign Bit

```
Require:
    $input$: Floating point feature
    $n\_bins$: Number of bins the input range is divided into
    $range\_max$: Maximum of input feature range
    $range\_min$: Minimum of input feature range
    Initialize $binary\_rep$ array of size $n\_bins + 1$
    Compute $bin\_delta = (range\_max - range\_min)/n\_bins$
    if $input < 0$ then
        Set $binary\_rep[0] = 1$                                   ▷ Negative input
    else
        Set $binary\_rep[0] = 0$                                   ▷ Positive input
    end if
    for $i \in [1, n\_bins + 1]$ do
    $min\_bin = range\_min + (i - 1) \cdot bin\_delta$
    $max\_bin = range\_min + i \cdot bin\_delta$
        if $min\_bin \leq abs(input) \leq max\_bin$ then
            $binary\_rep[i] = 1$
        end if
    end for
    return $binary\_rep$
```

Algorithm 5 Greater Than Binarizer with Sign Bit

```
Require:
    $input$: Floating point feature
    $n\_bins$: Number of bins the input range is divided into
    $range\_max$: Maximum of input feature range
    $range\_min$: Minimum of input feature range
    Initialize $binary\_rep$ array of size $n\_bins + 1$
    Compute $bin\_delta = (range\_max - range\_min)/n\_bins$
    if $input < 0$ then
        Set $binary\_rep[0] = 1$                                   ▷ Negative input
    else
        Set $binary\_rep[0] = 0$                                   ▷ Positive input
    end if
    for $i \in [1, n\_bins + 1]$ do
    $min\_bin = range\_min + (i - 1) \cdot bin\_delta$
    $max\_bin = range\_min + i \cdot bin\_delta$
        if $abs(input) \geq min\_bin$ then
            $binary\_rep[i] = 1$
        end if
    end for
    return $binary\_rep$
```

Algorithm 6 Less Than Binarizer

```
Require:
  $input$: Floating point feature
  $n\_bins$: Number of bins the input range is divided into
  $range\_max$: Maximum of input feature range
  $range\_min$: Minimum of input feature range
  Initialize $binary\_rep$ array of size $n\_bins$
  Compute $bin\_delta = (range\_max - range\_min)/n\_bins$
  for $i \in [0, n\_bins]$ do
  $max\_bin = range\_min + i \cdot bin\_delta$
     if $input \leq max\_bin$ then
        $binary\_rep[i] = 1$
     end if
  end for
  return $binary\_rep$
```

C Tables of experimental results

Computation Time: CartPole

Agent Type	Number of Clauses	Approx. Computation Time
RTM	3000	$17hrs$
RTM	2000	$14hrs$
RTM	1000	$10hrs$
RTM	500	$4hrs$
IW-RTM	3000	$18hrs$
IW-RTM	2000	$14hrs$
IW-RTM	1000	$12hrs$
IW-RTM	500	$5hrs$

Table 1: Average computation time of RTM agents for CartPole environment, 300 episodes. The MLP agent took 8 hours on the same CPU.

Average Wins: CartPole

Agent Type	Memory Type	Number of Clauses	Threshold	Specificity	Win Rate
RTM	**ER**	2000	1	5	10%
RTM	**PER**	2000	1	5	12%
RTM	**ER**	2000	1000	5	0%
RTM	**PER**	2000	1000	5	0%
RTM	**ER**	1000	1	5	10%
RTM	**PER**	1000	1	5	12%
RTM	**ER**	500	1	5	0%
RTM	**PER**	500	1	5	0%
IW-RTM	**ER**	1000	1	5	20%
IW-RTM	**PER**	1000	1	5	20%
IW-RTM	**ER**	1000	500	5	2%
IW-RTM	**PER**	1000	500	5	2%
IW-RTM	**ER**	500	1	5	0%
IW-RTM	**PER**	500	1	5	0%
IW-RTM	**ER**	500	250	5	0%
IW-RTM	**PER**	500	250	5	0%

Table 2: Performance of selected RTM agents on the CartPole environment for different hyperparameter configurations.

Average Wins: MountainCar - RTM

Agent Type	Memory Type	Number of Clauses	Threshold	Specificity	Win Rate
RTM	**ER**	2000	1	5	0%
RTM	**PER**	2000	1	5	0%
RTM	**ER**	2000	1000	5	0%
RTM	**PER**	2000	1000	5	0%
RTM	**ER**	1000	1	5	0%
RTM	**PER**	1000	1	5	0%
RTM	**ER**	500	1	5	0%
RTM	**PER**	500	1	5	0%
RTM	**ER**	2000	1	100	0%
RTM	**PER**	2000	1	100	0%
RTM	**ER**	2000	1000	100	0%
RTM	**PER**	2000	1000	100	0%
RTM	**ER**	1000	1	100	0%
RTM	**PER**	1000	1	100	0%
RTM	**ER**	500	1	100	0%
RTM	**PER**	500	1	100	0%

Table 3: Performance of selected RTM agents on the MountainCar environment for different hyper-parameter configurations.

Average Wins: MountainCar - IW-RTM

Agent Type	Memory Type	Number of Clauses	Threshold	Specificity	Win Rate
IW-RTM	**ER**	1000	1	5	0%
IW-RTM	**PER**	1000	1	5	0%
IW-RTM	**ER**	1000	500	5	0%
IW-RTM	**PER**	1000	500	5	0%
IW-RTM	**ER**	500	1	5	0%
IW-RTM	**PER**	500	1	5	0%
IW-RTM	**ER**	500	250	5	0%
IW-RTM	**PER**	500	250	5	0%
IW-RTM	**ER**	1000	1	100	0%
IW-RTM	**PER**	1000	1	100	0%
IW-RTM	**ER**	1000	500	100	0%
IW-RTM	**PER**	1000	500	100	0%
IW-RTM	**ER**	500	1	0.001	0%
IW-RTM	**PER**	500	1	0.001	0%
IW-RTM	**ER**	500	250	0.001	0%
IW-RTM	**PER**	500	250	0.001	0%

Table 4: Performance of selected IW-RTM agents on the MountainCar environment for different hyperparameter configurations.

D Graphs comparing methods

CartPole

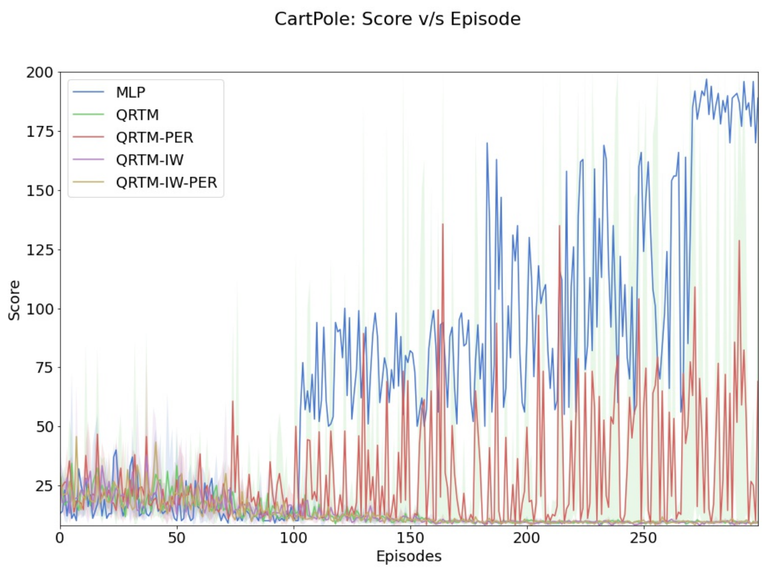


Figure 2: Performance of the RTM agents as compared to the MLP on CartPole environment. Average and deviation of the scores depicted in graph for RTM agents. ($n\_clauses = 500, T = 1, s = 5$)

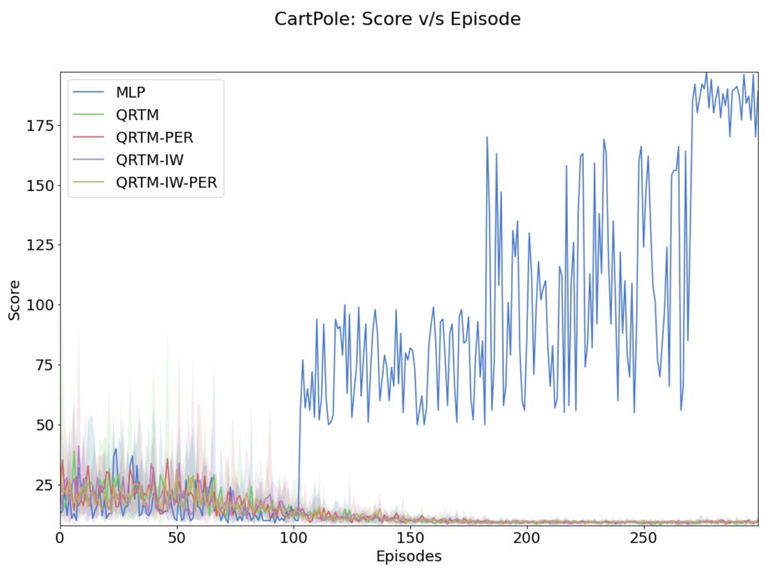


Figure 3: Performance of the RTM agents as compared to the MLP on CartPole environment. Average and deviation of the scores depicted in graph for RTM agents. ($n\_clauses = 200, T = 1, s = 5$)

MountainCar

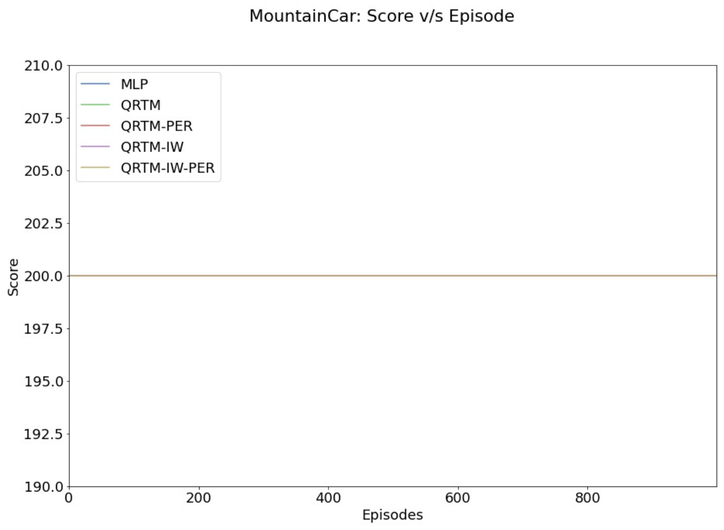


Figure 4: Performance of the RTM agents as compared to the MLP on MountainCar environment. Average scores depicted in graph for RTM agents. ($n\_clauses = 500, T = 1, s = 5$)

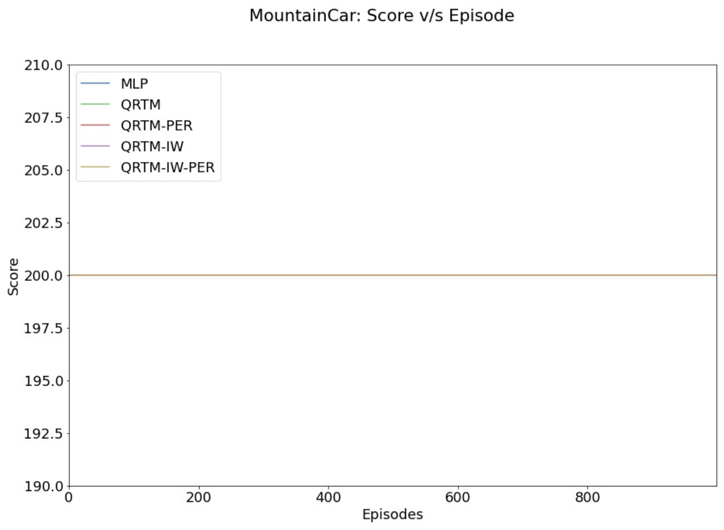


Figure 5: Performance of the RTM agents as compared to the MLP on MountainCar environment. Average scores depicted in graph for RTM agents. ($n\_clauses = 1000, T = 1, s = 5$)