Hierarchical Temporal Memory as a reinforcement learning method

Author: Antonio Sánchez Gómez

Supervisor: Dr. Jonathan Shapiro

Project submitted for the degree of BSc of Computer Science with Business and Management, 3rd May 2016
HTM as a reinforcement learning method

Abstract:

In this document the author explores the performance of Hierarchical Temporal Memory (HTM) in a reinforcement learning system and benchmarks it with that of Q-Learning, a state of the art method. We compare both system in 3 experiments, a basic problem, a second order Markov problem and a delayed reinforcement problem. We find that HTM is capable of solving the basic and second order problems but incapable of delayed reinforcement learning. Q-Learning, in contrast, solves the basic problem 40 times faster and is capable of delayed reinforcement learning but not of solving the second order Markov problem.
Acknowledgments:

I wish to thank the University of Manchester staff for their boundless dedication and commitment to quality teaching and especially Jon Shapiro for his help, advice and support throughout this last year. I would also like to mention the communities behind NuPIC (Numenta, 2016) and Matplotlib (Hunter, 2007) whose libraries have greatly simplified the development of this project.
HTM as a reinforcement learning method

Table of contents:

• Introduction 6
• Reinforcement Learning 8
  o Reinforcement learning framework 8
    ▪ Task component 1: Environment and state 9
    ▪ Task component 2: Reward 10
    ▪ Agent component 1: Value function 10
    ▪ Agent component 2: Policy 11
  o Reinforcement learning methods 12
• Hierarchical Temporal Memory 15
  o Context 15
  o HTM regions 18
    ▪ Form a sparse distributed representation of the input 19
    ▪ Form a representation of the input in the context of previous inputs 20
    ▪ Form a prediction based on the context of previous inputs 21
  o HTM as a reinforcement learning system 24
• Results 27
  o Experiment 1 27
  o Experiment 2 29
  o Experiment 3 32
• Conclusions 34
  o Future work 34
  o Reflection 36
• References 38
• Appendix A: parameter values 40
List of figures:

• 2.1: Interactions in a reinforcement learning system 8
• 2.2: A slice of the space of reinforcement learning methods 13

• 3.1: Interactions between the old brain and the neocortex 16
• 3.2: structure of the neocortex 17
• 3.3: HTM structure 18
• 3.4: example of SDRs 20
• 3.5: pseudocode for putting cells in the predictive state 21
• 3.6: cells’ segments and synapses 22
• 3.7: HTM outputs and predictions for sequence A, B, A, B 23
• 3.8: simple reinforcement learning problem 24
• 3.9: HTM system example 1 24
• 3.10: HTM system example 2 25

• 4.1: Experiment 1 problem representation 28
• 4.2: evolution of accuracy of HTM predictions 29
• 4.3: Experiment 2 problem representation 30
• 4.4: accumulated reward of Q-Learning 31
• 4.5: Experiment 3 problem representation 32
• 4.6: performance of HTM system 33
Chapter 1: Introduction

Reinforcement learning is a field of Artificial Intelligence that studies learning by interacting with the environment. This differs from supervised learning where the objective is to develop an algorithm capable of learning from a set of examples. In contrast, here the aim is to optimize the interaction between an agent and its environment in order to achieve a predetermined goal.

In reinforcement learning, the agent takes actions, the environment reacts by changing its state and then, based on them, the agent receives rewards or penalties, which define its task. In order to achieve this task, the agent must balance exploratory behaviour, when it takes actions aimed at increasing its knowledge of the environment, and exploitative behaviour, when it takes the actions that -according to its current knowledge- provide the highest reward in relation to its goal.

For example, in a chess-playing agent the actions are the possible moves on the board; the environment is the position of all the pieces, which changes with the agent’s and its opponent’s moves; and the task of winning the match can be represented with a reward of 1 in case of a win, of -1 in case of a lose and of 0 in case of a draw.

According to Sutton and Barton (1998), reinforcement learning systems have 4 different parts:

- **A policy**: which defines the behaviour that the agent implements by mapping environment states to the probability of taking certain actions

- **A reward function**: that defines the goal of the agent by providing a numerical reward in response to the agent’s actions and the environment’s state.
HTM as a reinforcement learning method

• **A value function**: which defines the long-term value of each state. While the reward defines the immediate value of any given state, the value function returns an indication of the rewards that will be obtained from the future states that the agent will reach following its current policy.

• **A model of the environment**: which mimics the behaviour of the environment, predicts future states and rewards and can be used for planning. This part is optional and not all reinforcement learning systems use it.

On the other hand, Hierarchical Temporal Memory (HTM) is an online prediction algorithm that aims to replicate the way the human brain works. As reinforcement learning was originally inspired by psychological theories, like classical and operant conditioning (Sutton & Barto, 1981 and Sutton & Barto, 1990); it seems a natural step to apply HTM to reinforcement learning problems.

The resulting system might help overcome some of the limitations of reinforcement learning. In particular, it might support non-Markovian environments because HTM is capable of using higher order patterns to make predictions (Hawnkins, 2010). Non-Markovian environments are those where the next state and reward are determined by the history of states and actions rather than the immediately previous ones. The objectives of this project are to explore the performance of using HTM as the value function on a reinforcement learning system (i.e. use it to return an indication of future rewards) and to compare the performance of the resulting system with that of other approaches.
Chapter 2: Reinforcement learning

In this chapter we will develop the reinforcement learning framework that we briefly presented in the previous chapter as well as introduce 4 different types of approaches.

Reinforcement learning framework:

As it has already been mentioned, the aim of reinforcement learning is to optimize the interactions between an agent and its environment in order to achieve a certain goal. To formally define this process, we must first establish the boundary between the agent and the environment. For the purpose of this report, we will include in the agent everything that it can change at will. This does not include the reward function, because if it was under the agent’s control then it could be changed to give a reward even without completing the objective task.

Figure 2.1: interactions in a reinforcement learning system between an agent (brown box) and a task (green box).

Using this boundary we can define the agent to be composed of a policy and a value function, while the environment and a reward function define a particular task that the agent needs to carry out. As reflected in figure 2.1, at discrete timesteps $t=0,1,2…$ the agent receives a representation of the environment state $S_t \in S$, where $S$ is the set of
**HTM as a reinforcement learning method**

all possible states, and responds with an action \( A_t \in A(S_t) \) based on the information from the value function, where \( A(S_t) \) is the set of actions available from state \( S_t \). One time step later, the agent receives a reward \( R_{t+1} \in \mathbb{R} \) (the reward may be 0) and finds itself in a new state \( S_{t+1} \). Over the next pages we will aim to formally define each of these components.

**Task component 1: Environment and state**

The state encapsulates the information given by the environment that the agent uses to take decisions. The reinforcement learning systems described by Sutton & Barto assume that the state signal complies with the Markov property. That is, the current state in itself has all the relevant information contained in the history of the interactions:

\[
\Pr(R_{t+1} = r, S_{t+1} = s \mid S_0, A_0, R_0, S_1, A_1, R_1, \ldots, S_t, A_t, R_t) = \Pr(R_{t+1} = r, S_{t+1} = s \mid S_t, A_t).
\]

Any decision system that contains the property described above is considered a Markov Decision Process (MDP).

For simplicity, in this report we will only consider finite MDPs. Finite MDPs are those where the state and action spaces are finite and completely described by its state and action sets; its transition probabilities, \( p(s' \mid s, a) \); and the expected value of next rewards, \( r(s, a, s') \):

\[
p(s' \mid s, a) = \Pr(S_{t+1} = s' \mid S_t = s, A_t = a)
\]

\[
r(s, a, s') = E(R_{t+1} \mid S_t = s, A_t = a, S_{t+1} = s').
\]

Where \( \Pr \) denotes the probability that the next state will be \( s' \) and \( E \) the expected reward of \( s' \).
HTM as a reinforcement learning method

As we mentioned before, since HTM is capable of recognizing higher order patterns it may also be able to solve problems set in weaker Markovian environments. In particular, we will investigate second order MDPs, where the next state and reward are not just determined by the immediately previous interactions but also the ones before:

\[
\Pr(R_{t+1}=r, S_{t+1}=s \mid S_0, A_0, R_0, S_1, A_1, R_1, \ldots, S_t, A_t, R_t) = \\
= \Pr(R_{t+1}=r, S_{t+1}=s \mid S_{t-1}, A_{t-1}, S_t, A_t).
\]

**Task component 2: Reward**

The reward is a number, \( R_{t+1} \in \mathbb{R} \), given by a function of the current state, the action taken and the next state:

\[
r(s, a, s') = E(R_{t+1} \mid S_t = s, A_t = a, S_{t+1} = s').
\]

This reward function defines the task the agent needs to perform by associating desirable states with rewards and undesirable ones with penalties. The agent would thus seek to get to the desirable states that signify the completion of a task (like the state where you have won a chess game) and avoid undesirable ones (like the state of losing the game).

**Agent component 1: Value function**

The value function represents the agent’s knowledge of the environment. The agent goal is to maximise its accumulated reward and that is the meaning returned by the value function \( G_t \), the sum of the rewards that can be expected from future timesteps:

\[
G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{k=T-t-1} \gamma^k R_{t+k+1}.
\]
**HTM as a reinforcement learning method**

Where $0 \leq \gamma \leq 1$ is the discount rate that marks the present value of future rewards and $\text{T} = \infty$ in the case of continuing tasks or $\text{T} \in \mathbb{N}$ in the case of episodic tasks (i.e. tasks that have an end state).

However value functions are dependent on the policy being implemented. I.e the value of the state $s$ under policy $\pi$, denoted $v_{\pi}(s)$, is the accumulated reward expected when starting in state $s$ and following $\pi$ thereafter. For MDPs it can be formally defined as:

$$v_{\pi}(S_t) = \mathbb{E}_\pi \left[ \sum_{k=0}^{k=T-t-1} \gamma^k R_{t+k+1} \right].$$

Almost all reinforcement learning algorithms involve approximating the current estimated value function, denoted as $V$, to the real value function, denoted as $v_{\pi}$. This approximation is done by backing off, or, in other words, bringing information from a successor state back into a previous one in order to better approximate its value.

**Agent component 2: Policy**

The policy represents a mapping from environment states to the probability of taking an action at that time step:

$$\pi_t(a \mid s) = \Pr(A_t = a \mid S_t = s).$$

In order to achieve its goal in an unknown environment the agent will need to balance the exploration of new paths to better approximate the value function and the exploitation of its knowledge in order to achieve its task.

As policies are not the focus of this document, in this report we will use the simple $\varepsilon$-greedy policy to balance exploration and exploitation. In an $\varepsilon$-greedy policy, we take the optimal action (i.e. we exploit our knowledge) with probability $1 - \varepsilon$ and take a random action (i.e. we explore new knowledge) with probability $\varepsilon$. 
**HTM as a reinforcement learning method**

Value functions also provide a partial ordering of policies:

\[ \pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in S. \]

and can be used to define the optimal one. That is, the policy that maximizes the expected accumulated reward for all the states. Once the agent has converged to the optimal policy, it can be said it has learned to perform its task and thus the reinforcement learning problem has been solved.

**Reinforcement learning methods:**

Sutton and Barto identify 4 broad categories of approaches based on two axes, plotted on figure 2.2, the width and the depth of the backing done (the transferring of information from successor states):

- **Exhaustive Search:** where all the states are systematically explored to construct the value function. This is unfeasible except for small problems.
- **Dynamic Programming (DP):** which backup values from all the immediate successor states. They do this by continuously sweeping through all the states and updating their value using the one from the immediate successor states and their probabilities of occurring:

\[
v(s) \leftarrow \sum_a \pi(a | s) \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v(s')].
\]

Dynamic Programming methods continue this process until the values of all the states converge and stop changing. Also, because of the need to know the transition probabilities and rewards of each state, Dynamic Programming methods require a perfect model of the environment.
**HTM as a reinforcement learning method**

- **Monte Carlo (MC):** which backups values from the successor states that have been sampled in a particular exploration. That is, they follow a particular policy until a final state has been reached and then update the value of each state with the accumulated reward of that sample:

  \[ v(s) \leftarrow v(s) + \alpha [ G_t - v(s)] \].

  Where \( \alpha \) is a step-size parameter and \( G_t \) is the averaged accumulated rewards of the sample as it was defined in the previous section.

- **Temporal Difference (TD):** which backup the value of the current state using its reward and the value of one successor state:

  \[ v(s) \leftarrow v(s) + \alpha [ R_{t+1} - \gamma v(S_{t+1}) - v(s) ] \].

**Figure 2.2:** a slice of the space of reinforcement learning methods (Sutton & Barto, 1998)

TD methods are the most widely used as they do not need a complete model of the environment, unlike DP, and can be used in an online fashion, unlike MC methods.
HTM as a reinforcement learning method

that need to wait until the end state to perform the update. Sutton and Barto describe two types of TD methods:

- **On policy (SARSA):** which uses the state visited by the current policy to perform the update:

  \[ v(s) \leftarrow v(s) + \alpha [ R_{t+1} - \gamma v(S_{t+1}) - v(s) ]. \]

- **Off policy (Q-Learning):** which performs the update based on the next most valuable state, irrespective of the actual action taken by the current policy:

  \[ v(s) \leftarrow v(s) + \alpha [ R_{t+1} - \gamma \max_s v(S_{t+1}) - v(s) ]. \]

Q-Learning is capable of overlooking some inefficacies caused by stochastic policies, like \( \epsilon \)-greed, that balance exploration and exploitation. For this reason, in this report we will benchmark the performance of HTM against that of Q-Learning, as a state-of-the-art algorithm from reinforcement learning.
**Chapter 3: Hierarchical Temporal Memory**

Hierarchical Temporal Memory (HTM) is an online prediction algorithm that implements the theory of Jeff Hawkins of how the brain works. He started by studying brain theory with the aim of understanding its principles in order to create a system capable of “true” artificial intelligence. In 2007 he published a book, “On Intelligence”, with his hypotheses on the principles and the structure behind the brain’s cognition. He also funded a company, Numenta, to drive this effort and its commercialization. In 2009, Numenta released its first product, GROK, and in 2013 it sponsored, and has since then supported, an open source community around Hawkins’ ideas: Numenta Platform for Intelligent Computing (NuPIC) (Hawkins, 2013).

In this section we will outline Hawkins’ theory of how the brain works, provide an overview of HTM as part of this wider theory and finish describing the approach we have taken to apply HTM to reinforcement learning problems.

**Context**

Hawkins (2007) subscribes to the idea that intelligence consists of making predictions. I.e. continuously using our memories to make predictions of how the environment will behave, and then use them to guide our behaviour.

He identifies two different parts of the brain:

- **The “primitive” cortex (or old brain):** that we share with reptiles and all other animals, and is in charge of implementing basic behaviour, like regulating breathing, temperature, reflexes, etc…

- **The neocortex:** a relatively late evolutionary addition, shared only by mammals, that is in charge of memory and prediction. The neo-cortex feeds its
predictions into the reptilian brain modifying its actions to avoid predicted dangers and look for expected rewards.

In order to do this, the neocortex output is connected to the input of the old brain and a copy of the output of the old brain is sent into the neocortex (as represented in figure 3.1). That is, the neocortex does not directly control the actuators, but rather, by linking its output with the responses of the old brain, the neocortex is then able to learn to control those responses and generate new behaviour.

![Interaction between the old brain and the neocortex](image)

**Figure 3.1:** interaction between the old brain and the neocortex

The neocortex itself is divided into different regions organized hierarchically as represented in figure 3.2. Each hierarchical level operates at a higher level of abstraction, for example one region handles the visual information directly, another the auditory input and both may feed into a third region that connects the input of both senses.

Each region is divided into 4 to 6 layers, each implementing a variation of the same algorithm: HTM. Each layer has a role, like handling raw sensory input, integrating
HTM as a reinforcement learning method

the motor commands, handling feedback from upper regions, etc\(^1\). Together they build a more abstract and stable representation on the input that is then fed into upper layers as described before.

In other words, each layer uses the HTM algorithm to build temporal and spatial patterns of the input and convert the highly noisy and changing raw sensory information into a more stable representation at a higher level of abstraction. Presumably, by recursively repeating this process over several layers and hierarchical regions, the neocortex is able to build and use the complex models that we recognize as human intelligence.

In order to generate behaviour, according to Hawkins’ theory, the output of the top level is cascaded down the regions and fed into the old brain, which then generates the motor commands and sends them to the muscles and other actuators.

---

\(^1\) The exact roles of the different layers and the interactions between them are still unclear and are a subject of research and development (Hawkins, 2014).

\(^2\) Not to be confused with the neocortex regions that we have discussed so far. An
**HTM as a reinforcement learning method**

In this project we will use a single HTM layer (also called regions) as a value prediction function within a reinforcement learning framework, with the idea that the resulting system may be capable of generating intelligent behaviour and deal with non-Markovian environments thanks to HTM’s capabilities of pattern discovery.

**HTM regions**

According to Numenta’s whitepaper (Hawkins, 2010) HTM is formed by a set of cells grouped in columns as shown in Figure 3.3. The columns can be active or inactive and the cells can be in 3 different states: active, inactive or in the “predictive state”. The input to the region is given in the form of a fix-length vector of bits.

---

2 Not to be confused with the neocortex regions that we have discussed so far. An HTM region is the set of cells that make up a single generic layer of a neocortex region.
HTM as a reinforcement learning method

At initialization time, each column is permanently connected to a random subset of the bits in the input vector. These links, called synapses in HTM literature, will still point to the same position, even after new inputs come in and the bits in that position change. The synapses have a “permanence value”, \(0 \leq p_s \leq 1\), associated with them and a threshold parameter, \(0 \leq t \leq 1\), common to all the synapses.

A synapse will be considered active if its permanence value is above the set threshold: \(p_s > t\). These permanence values are initialized randomly and then, as we will see later, slowly changed to provide a more stable representation of the input, giving the first mechanism by which HTM regions learn.

At a high level, the algorithm relies on sparsity to form more abstract representations of the input, learn the connections among those inputs and use this more abstract representation together with the learned connections to make predictions. More explicitly an HTM region goes through the following steps, which we will explain in detail:

1) Form a sparse distributed representation of the input
2) Form a representation of the input in the context of previous inputs
3) Form a prediction based on the context of previous inputs

1) Form a sparse distributed representation of the input

The first step is to convert the sensory input, represented by numbers, symbols, words, etc. into a “Sparse Distributed Representation” (SDR). SDRs are highly sparse n-dimensional vectors of binary components, i.e. a vector of bits where a small percentage of the components are 1s. In SDRs every component has semantic meaning and, thus, similar input will have a higher number of active components overlapping.
**HTM as a reinforcement learning method**

For example, below are the SDRs for the scalars 1, 2 and 3. The given vectors have a relatively small percentage of active bits and the overlap between 1-2 and 2-3 is higher than the overlap between 1-3.

<table>
<thead>
<tr>
<th></th>
<th>0 1 1 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1 1 1 0 0 0 0 0 0]</td>
</tr>
<tr>
<td>2</td>
<td>[0 1 1 0 0 0 0 0 0]</td>
</tr>
<tr>
<td>3</td>
<td>[0 0 1 1 0 0 0 0 0]</td>
</tr>
</tbody>
</table>

Figure 3.4: example of SDRs

Numenta (Ahmad, 2014) claims that SDRs are the data structure of the brain and that they provide HTM regions with high capacity, noise resistance and the ability to represent several inputs. This last part is necessary, for example, when creating representations at a higher level of abstraction or making predictions of sequences that cover several steps (Ahmad, 2015).

Numenta (Nupic, 2016) has developed encoders for converting scalars, categories, dates or even words into SDRs. These are used in this project but for brevity we will not describe them.

**2) Form a representation of the input in the context of previous inputs**

Every time an input is received and converted into its SDR representation, an “overlap score” is computed for every column. The overlap score is the sum of all the synapses of a column that are active and its corresponding bit in the SDR vector is 1:

\[
\text{OverlapScore}(c) = \sum_{s \in c} b_s (1 \ if \ p_s > t \ else \ 0).
\]

Where s is a synapse associated with column c, and b_s and p_s are the bit component and the permanence value associated with synapse s.

The columns are then sorted in descending order according to their overlap score and only the top 2% of columns are set as “active”. This is done in order to preserve
HTM as a reinforcement learning method

sparsity and provide resistance against noise in the sensory input. If learning is active, the permanence value of the synapses of the columns that have been activated is changed. If a synapse was connecting to a bit that was active, its permanence value is increased, and if the bit was inactive, then the permanence value is decreased. This is done to ensure that similar SDR inputs lead to the same or very similar columns being activated, and thus form a more stable and abstract representation.

The final step is to put the input, now represented by the active columns, in the context of previous inputs. In order to do this, for every column that was activated, the state of the cells it contains is changed. If any cell within the column was in a predictive state, then all the cells with that state will become active. If no cell was in a predictive state, then all the cells in the column will become active. This is called “bursting” and shows that input was unexpected and not predicted.

If (no cell in the column is in the predictive state):
Set as active (all the cells)
Else:
Set as active (the cells in the predictive state)

Figure 3.5: pseudocode for putting cells in the predictive state

3) Form a prediction based on the context of previous inputs

The output of HTM regions is the set of all the cells that are in the active or predictive state. Once the HTM region status represents the contextualized input, the next step is to calculate which cells go into the predictive state.

Each cell has a set of segments that are connected via synapses to a small subset of the neighbouring cells as represented in figure 3.6. These synapses, as the column ones, have a permanence value associated with them and become active if the permanence goes above a set threshold. A cell will change into the predictive state if
any of its segments become active; and a segment will become active if the number of active synapses that are connected to active cells goes above another threshold.

![Figure 3.6: cells’ segments and synapses](image)

This provides our second mechanism for learning. Whenever a segment becomes active, the permanence of each synapse that is connected to an active cell is increased and any that is connected to an inactive cell is decreased. Furthermore, in order to form predictions that cover several steps, whenever a cell becomes active from one segment, it will choose a second one and tune its synapses to match the cells that were active in the previous timestep. I.e. it will increase the permanence values of the synapses that are connected to a cell that was active in the previous timestep and decrease the ones that are connected to one that was inactive.

From this process we can get the set of cells that are in the active and predictive state, the output of the HTM region. However, as mentioned in the beginning, HTM is part of a wider theory and to be able to use it independently it becomes necessary to “translate” the output of the cell states into a form that is readable and understandable.
**HTM as a reinforcement learning method**

by other components. This is done by using a classifier to link the outcome of the region to the input at the next time step.

For example, the figure below represents HTM being fed the sequence A, B, A, B, … It lists the input at each timestep, the output of the HTM region, were the 1s represent cells that are in the active or predictive state, and the predictions from the classifier based on that output. As can be seen, in the first two instances, the classifier is not able to interpret the region’s output, but it links the [0, 1, 0, 1] output to the next input, B, and the [1, 0, 1, 0] output to the next input, A. After forming this association, it can then correctly “translate” the HTM output into manageable predictions.

<table>
<thead>
<tr>
<th>Input</th>
<th>Region output</th>
<th>Classifier prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>[0, 1, 0, 1]</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>[1, 0, 1, 0]</td>
<td>N/A</td>
</tr>
<tr>
<td>A</td>
<td>[0, 1, 0, 1]</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>[1, 0, 1, 0]</td>
<td>A</td>
</tr>
</tbody>
</table>

Figure 3.7: HTM output and predictions for sequence A, B, A, B

Following the algorithm that has been outlined, HTM is capable of forming stable representations of the input in the context of previous ones and learn the transitions between them to make predictions. The algorithm itself is very convoluted and, due to space constrains, the description here is not meant to be comprehensive, but rather to give the reader a feeling for how it operates. We thus omit several details and nuances that are necessary for an implementation. For more information, please refer to Numenta’s whitepaper.
**HTM as a reinforcement learning method**

**HTM as a reinforcement learning system**

In this section we describe how we will fit an HTM region into a reinforcement learning system and how the resulting system will work. As we described in the previous chapter the agent is composed of a policy and a value function. As policy we will use $\epsilon$-greedy, where with probability $\epsilon$ we will take a random action and with probability $1-\epsilon$ we will take the action that HTM predicts has the highest reward. At each timestep $t$, we will feed into the region a triple with the form $[S_t, R_t, S_{t+1}]$; the current state, the current reward and the next state. From this information we expect HTM to associate the history of states with the reward at the next timestep.

![Diagram](image)

**Figure 3.8:** A simple reinforcement learning problem where $S$ is the state name and [0] is the reward of the state.

For example, when going through the states $S \rightarrow C \rightarrow G \rightarrow E$ from the problem represented in figure 3.8, below we list the inputs the HTM region will receive and the predictions we expect:

<table>
<thead>
<tr>
<th>Timestep</th>
<th>Input</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$[S, 0, C]$</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$[C, 0, G]$</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>$[G, 1, E]$</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>Problem has ended</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.9:** HTM system example 1
HTM as a reinforcement learning method

However, to take decisions it will be necessary to obtain the predictions of going to multiple states from the same position. If we were to feed all the triples to the same region, this would break the relationship between \( S_{t+1} \) and \( R_t \) that we want the algorithm to learn. In order to prevent this, we will i) clone the region, ii) feed the clones the potential paths, iii) obtain their predictions, iv) use our policy to decide on an action based on these forecasts and v) feed the original region the action that was decided.

So in the example before, at timestep 2 when we are in state C, we had to decide whether to go to K or G. In this instance we would actually create two clones, then feed each of them one triple that corresponds to going to K or G, and use the predictions returned by the clones in the \( \epsilon \)-greedy policy to finally, in this example, decide to go to G as it has the maximum predicted reward. We feed this decision into the original region and continue the process as before. This process is represented in the figure below:

<table>
<thead>
<tr>
<th>Timestep</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>([S, 0, C])</td>
<td><strong>Use ( \epsilon )-greedy to make a decision based on the predictions from the clones</strong></td>
<td>([C, 0, G])</td>
</tr>
<tr>
<td>Clone 1</td>
<td>([C, 0, K])</td>
<td>Predict 0</td>
<td></td>
</tr>
<tr>
<td>Clone 2</td>
<td>([C, 0, G])</td>
<td>Predict 1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.10: HTM system example 2

This system should be capable of solving second order problems, thanks to HTM’s ability of identifying high order patterns. However, the predictions returned by HTM represent the reward of the next state, rather than its value as we defined it in the previous chapter. Because of this difference, the algorithm cannot be expected to solve problems where the value of the next state, rather than its reward, is necessary.
HTM as a reinforcement learning method

That is the case, for example, of the problem we use in experiment 3. We will explore these qualities in further detail during the next chapter.
Chapter 4: Results

As it has been mentioned before, in this chapter we will compare the performance of Q-Learning and HTM in a reinforcement learning system. For doing this comparison we will run 3 experiments:

- **Experiment 1**: the first experiment is aimed at ensuring that the HTM system described in the previous chapter is feasible and at benchmarking its performance against Q-learning in a basic problem.

- **Experiment 2**: the second aims to explore whether HTM is capable of solving problems that have weak Markovian properties. This is the main advantage that can be foreseen from using HTM in reinforcement learning.

- **Experiment 3**: the third aims to expose the drawback that we mentioned in the previous chapter. I.e. because HTM predicts the rewards of a state rather than its value, it is not capable of solving problems were the distance between the state where a choice has to be made and the reward information to correctly make the choice is of multiple states.

Unless otherwise specified, the parameters used for conducting these experiments are listed in Appendix A.

**Experiment 1**

For this experiment we will use the simple problem outlined in Figure 4.1. The agent will start in state C and move within the constraints of the problem until G (the “Goal state”) is found, point at which the problem is considered finished. This problem has only one choice at point C (the “Choice state”) and its optimal solution is the path C ➔ G with a length of 2 steps. No matter what path is taken the accumulated reward
HTM as a reinforcement learning method

will always be 1, thus to assess whether the optimal solution has been found we will only focus on the length of the path taken.

Figure 4.1: Experiment 1 problem representation.

As we have mentioned a couple times before, to balance exploration and exploitation we will use an $\varepsilon$-greedy policy. At the beginning the algorithms will need a higher rate of exploration whereas towards the end, the randomness from the exploration face will interfere with the performance assessment. To avoid this we will use a decreasing epsilon with $\varepsilon = \frac{1}{n' \text{of iterations}}$.

After running it over 300 iterations (i.e. trying to solve the problem 300 times) and averaging over 14 trials, the results indicate that HTM converges to the correct solution after 286 iterations, with an average path of length 2.0 and a standard deviation of 0.0. Q-Learning in contrast, after being run for 20 iterations and 10,000 trials, converges much faster and after 7 iterations it solves the problem in 2.0 steps with a standard deviation of 0.0.

To try to understand the performance of the HTM system, we run the problem with $\varepsilon = 1$ and analyse the accuracy of HTM predictions. Figure 4.2 shows the evolution of the accuracy of HTM when predicting the reward of the next step averaged over 100 trials. Three stages can be differentiated. During the first 15 iterations, the algorithm seems to lack enough information. Over the next 15, the accuracy grows quickly as
HTM as a reinforcement learning method

the algorithm learns that most states have a reward of 0. The remaining ones are used to learn that G has a reward of 1.

Figure 4.2: Evolution of the accuracy of HTM predictions averaged over 100 trials.

Experiment 2

For testing this experiment we will use the problem outlined in Figure 4.3, which is similar to the previous one but is transformed into a second order Markovian problem, where:

$$Pr(R_{t+1}=r, S_{t+1}=s \ | \ S_0, A_0, R_0, S_1, A_1, R_1, \ldots, S_t, A_t, R_t) =$$

$$= Pr(R_{t+1}=r, S_{t+1}=s \ | \ S_{t-1}, A_{t-1}, S_t, A_t)$$

In this problem, the agent also starts in C and its task is to first visit K (the “Key state”) and then finish in state G. To reflect this task, the reward in G will be -1 if K has not been visited and 1 if it has. The optimal path is C \(\rightarrow\) K \(\rightarrow\) C \(\rightarrow\) G which has
an accumulated reward of 1 and a length of 4. To assess whether the optimal path has been found we will track both, the accumulated reward and the number of visited states before reaching G.

Figure 4.3: Experiment 2 problem representation. The reward in G is 1 if K has been visited and -1 otherwise.

For Q-Learning we once again will use a decaying $\epsilon$-greedy policy with $\epsilon = \frac{1}{\text{number of iterations}}$. However, if we do random walks with the HTM system, as in the previous section with $\epsilon = 1$ and for 1500 iterations and 100 trials, we can see that the region learns in 3 stages: first finding that most states have a reward of 0, then that K and G are associated with rewards of 1 and -1 and then that 1 is the reward for the K $\Rightarrow$ C $\Rightarrow$ G sequence and -1 for the S $\Rightarrow$ C $\Rightarrow$ G one. This process takes the whole 1500 iterations. To prevent infinite loops when the algorithm thinks that K has a reward of 1, for HTM we will use a decreasing epsilon where $\epsilon = \min (0.09, \frac{100}{\text{number of iterations}})$.

The results indicate that, when run for 1500 iterations and 10 trials, after 1496 iterations HTM finds a path with an average reward of 1.0 and standard deviation of 0.0 and length of 4.2 and standard deviation of 0.6. Q-Learning, as expected, does not converge to the correct solution but cycles between C $\Rightarrow$ K $\Rightarrow$ C $\Rightarrow$ G and C $\Rightarrow$ G.
Figure 4.4 shows the cycling behaviour of Q-Learning when run for 100 iterations and 10,000 trials.

Figure 4.4: accumulated reward of Q-Learning averaged over 10,000 trials.

This cycling behaviour can be explained with the evolution of the values of K and G. The algorithm normally favours \( C \rightarrow G \) as the value of G is higher than that of K. However, every time this sequence occurs the value of G is decreased because the reward it receives is -1. When it goes below that of K, the algorithm then chooses \( C \rightarrow K \rightarrow C \rightarrow G \). It does not visit K more than once because its value is then decreased due to the lack of reward in C. After this occurs the value of G is increased (but not that of K) and Q-Learning reverts back to the \( C \rightarrow G \) path. The cycles become more spaced as the value of K decreases.
Experiment 3

In this final experiment we expose the flaw that we theorised in the previous chapter; that the HTM system is not capable of solving problems where the distance between the state where a decision is made and the reward information for taking the action is of multiple steps. That is, when to make a decision is necessary the value of a state rather than its reward.

For example, in the problem represented in figure 4.5, which we will use in this experiment, the only choice-point is again C, but in order to decide between going to K or B, is necessary to compare the rewards of its successor states, C and G. This is because both K and B have a reward of 0 and we need to look further ahead to be able to differentiate between each path. This information of future rewards is encapsulated in the value of the states rather than in its rewards.

![Figure 4.5: Experiment 3 problem representation](image)

In this problem, the optimal path is \( C \rightarrow B \rightarrow G \), with 3 steps. As the reward will be the same no matter how the goal is found, we will use the number of steps to assess the performance of the algorithms. We will again use an epsilon-greedy policy with \( \varepsilon = \frac{1}{\text{n of iterations}} \) for both methods as there will be little risk of infinite loops.

As expected, the results indicate that HTM is not capable of solving this problem and chooses randomly between C and B. Figure 4.6 shows this random behaviour when
HTM as a reinforcement learning method

HTM is run for 500 iterations and 6 trials. In contrast, Q-Learning is capable of solving it and after running it for 10,000 trials and 30 iterations it solves it after 30 iterations in 3.04 steps with a standard deviation 0.27.

Figure 4.6: performance of HTM system averaged over 6 trials
Chapter 5: Conclusions

This project explored the idea of using HTM as a reinforcement learning method. The results clearly indicate that HTM is still in its infancy and its performance is far from that of state-of-the-art reinforcement learning methods like Q-Learning.

The resulting system is capable of solving basic problems, and even second order ones that methods like TD cannot solve without the intervention of a human. HTM however presents a most glaring shortcoming in that is not capable of solving delayed reinforcement problems, which are many of the applications of the real world (like games or robotics). Q-Learning in contrast solves the basic problem 40 times faster and is capable of delayed reinforcement learning but not of solving the second order one.

Whether this myopia characteristic can be overcome, together with having its performance improved, will decide whether the HTM system proposed here is worth of further development; and future steps should focus on these tasks. In the next section we outline some ideas of how these problems can be surmounted as well as list some other areas that can help to better understand the qualities of the system.

Future work

One way to overcome the myopia problem is to repeat the cloning process that we use to decide which state to go. I.e. to use a clone for each possible path the agent can take. But doing this would require a quite detailed model of the environment to know the successor states from the immediate ones and will only be feasible in small problems that have a finite number of paths, i.e. that do not have loops unlike the ones we used in this report.

---

3 We understand that the immediate states is information given to the agent by the environment in order for the former to know which actions it has available
HTM as a reinforcement learning method

However, HTM output is supposed to represent predictions that encompass several steps, i.e. in theory, the output of the region represents the value of the state rather than its reward. If a better way to “translate” the output is found, this drawback could be avoided. For example, we could use several classifiers to associate the predictions to the rewards at different steps in advance and then compute the value from those rewards. Or maybe find some mechanism to have the classifier associate the output with the value of the state rather than its reward.

In regards to its performance, the current HTM implementation is very convoluted and not well understood, for example, only very recently has a mathematical model for part of the algorithm been proposed (Mnatzaganian, 2016). Once the principles behind its behaviour are better understood and its operations simplified and optimized, the author believes its performance should increase significantly. Another approach to improve the results of the system is to connect multiple regions in a layered system that, according to Hawkins’ theory, would result in a system capable of coping with more complex patterns.

Finally, it will also be interesting to explore the behaviour of this method in a wider range of problems:

- **Larger ones**: the problems used in this report were extremely simple and were used to demonstrate some of the qualities of the proposed system. In the future it will be necessary to explore its performance in more complex environments.

- **Continuous tasks**: in this document, we have only explored episodic tasks with a clear start and end. Continuous tasks, like trying to maintain a pole balanced by moving its base, should be well suited to the online nature of HTM.
HTM as a reinforcement learning method

• **Weaker Markovian environments**: the strength of HTM is finding patterns in the input stream, which, together with its ability to combine multiple regions to build higher levels of abstractions, could be the key to overcoming one of the current limitations of reinforcement learning.

**Reflection**

In this last section, I reflect on the evolution of the project and the opportunities it has provided me to grow my technical and soft skills. On the technical front, the work was developed in Python, a language I was not familiar with and whose duck-typing paradigm is quite different from the strongly-typed languages I have programmed in until now. I chose Python because Nupic, a freely available implementation of HTM, was offered in that language.

Testing was done with PyUnit, a widely used unit-testing framework for Python, and its use has forced me to write cleaner code and explore the world of dependency injection. Finally, I have also employed Docker, a container technology that is meant to ease the deployment of programs and its dependencies. This technology was not necessary for this project, but I decided to use and learn the tool as I believe it would become an important part of the software development toolkit.

On the technical front, this project has also given me a deeper and broader understanding of reinforcement learning and HTM theory. Reinforcement learning is very briefly touched upon in the 3rd year course unit AI and Games, but thanks to this work I have obtained a more holistic view of the different methods and approaches, as well as a more intimate understanding of the reinforcement learning framework. HTM is a relatively new approach that seems to be gathering momentum, and working with
such a new algorithm has proven an interesting challenge, especially since I underestimated how much in its infancy it was.

On the soft skill front, during this last year I followed the agile practice of short iterations and aimed to complete a small milestone every week. This has helped me to continuously move forward and spread out the workload over the whole year. However, I have realized I need to improve my planning skills and focus more on the end goal. I believe this would have reflected in a more systematic approach, and a deeper and more comprehensive study of HTM and reinforcement learning being done at the begging. Overall, I feel this project has helped me grow and I believe it will have a lasting impact on my future career.
HTM as a reinforcement learning method

References


HTM as a reinforcement learning method


HTM as a reinforcement learning method

Appendix A: Parameter values

Here we list all the parameter values that we have used to configure our Q-Learning implementation as well as the values we have chosen to configure the Nupic library.

Q-Learning:

- Learning step: 0.1
- Discounting factor: 0.9
- Initial value: 1

Nupic:

- 'model': "CLA",
- 'version': 1,
- 'predictAheadTime': None,
- 'modelParams':
  - 'inferenceType': 'TemporalMultiStep',
  - 'sensorParams':
  - 'verbosity': 0,
  - 'encoders':
    - 'currentState':
      - 'fieldname': "currentState",
      - 'name': "currentState",
      - 'type': 'CategoryEncoder',
      - 'categoryList': ['B', 'S', 'C', 'K', 'G', 'E'],
      - 'w': 21
    - 'nextState':
      - 'fieldname': "nextState",
      - 'name': "nextState",
      - 'type': 'CategoryEncoder',
      - 'categoryList': ['B', 'S', 'C', 'K', 'G', 'E'],
      - 'w': 21
    - 'reward':

HTM as a reinforcement learning method

- 'fieldname':'reward',
- 'name':'reward',
- 'type':'ScalarEncoder',
- 'maxval':1,
- 'minval':-1,
- 'w':21,
- 'resolution':1,
  - 'sensorAutoReset': None,
- 'spEnable': True,
- 'spParams':
  - 'spVerbosity': 0,
  - 'spatialImp': 'cpp',
  - 'globalInhibition': 1,
  - 'columnCount': 2048,
  - 'inputWidth': 0,
  - 'numActiveColumnsPerInhArea': 40,
  - 'seed': 1956,
  - 'potentialPct': 0.85,
  - 'synPermConnected': 0.1,
  - 'synPermActiveInc': 0.04,
  - 'synPermInactiveDec': 0.005,
- 'tpEnable': True,
- 'tpParams':
  - 'verbosity': 0,
  - 'columnCount': 2048,
  - 'cellsPerColumn': 32,
  - 'inputWidth': 2048,
  - 'seed': 1960,
  - 'temporalImp': 'cpp',
  - 'newSynapseCount': 20,
  - 'maxSynapsesPerSegment': 32,
  - 'maxSegmentsPerCell': 128,
  - 'initialPerm': 0.21,
HTM as a reinforcement learning method

- 'permanenceInc': 0.1,
- 'permanenceDec': 0.1,
- 'globalDecay': 0.0,
- 'maxAge': 0,
- 'minThreshold': 12,
- 'activationThreshold': 16,
- 'outputType': 'normal',
- 'pamLength': 1,

- 'clParams':
  - 'regionName': 'CLAClassifierRegion',
  - 'clVerbosity': 0,
  - 'alpha': 0.0001,
  - 'steps': '1,5',
  - 'implementation': 'cpp',

- 'trainSPNetOnlyIfRequested': False,