DEVELOPMENT AND SIMULATION OF FOCUS OF ATTENTION USING REINFORCEMENT LEARNING AND FUNCTION APPROXIMATION

by

STEPHEN RATZ

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

MAY 2013

Acknowledgements

I would like to thank Dr. Huber for being my advisor. I would also like to thank him for his advice and challenging me during our many discussions and meetings.

I would like to thank Dr. Zaruba and Dr. Kamangar for being on my committee.

I would like to thank Rasool Fakoor, Tim Dockins, Vamsikrishna Gopikrishna, and Sudhamsh Reddy for their moral support and companionship in the Learn Lab.

I would also like to thank Tim Dockins for his assistance in developing proof of concepts of Function Approximation using tile coding.

I would like to thank the Mayorga and Aquayo families for their time and company during some of my more trying times whilst I coded and wrote this thesis project.

I would like to thank Gill Stevens and Jason Ostrom for their patience and fighting for me during my internship as Software Engineer for Sipera Systems and later Avaya, Inc.

I would like to thank the many mentors and professors I've had over the years for showing me that which I could not see on my own.

April 17, 2013

Abstract

DEVELOPMENT AND SIMULATION OF FOCUS OF ATTENTION USING REINFORCEMENT LEARNING AND FUNCTION APPROXIMATION

Stephen Ratz, CSE

The University of Texas at Arlington, 2013

Supervising Professor: Manfred Huber

Without short-term memory, humans would have little hope to learn and accomplish tasks. The same can be said for artificially intelligent agents. Often referred as Miller's Law, the number of working objects that a human can hold in working memory is around seven. For an AI agent, the cost of keeping additional memory blocks is exponential. Other issues to consider are what to keep in memory and for how long. Only a few of many of an agent's previous steps may be important to hold on to. This thesis project attempts to train an intelligent agent to learn what to hold onto in memory using Reinforcement Learning in order to accomplish a task. Function Approximation is used to mitigate the memory requirements of a task as simple as block copying. The concepts used in this thesis can be applied to any task that requires memory management.

Table of Contents

List of Illustrations

List of Tables

Chapter I

Introduction

1.1 Problem Description

Humans are habitually trying to create mirror images of themselves. From small figurines to paintings to puppets, humans instinctually try to create and understand what they themselves are. The advancement of technology and modern computing has allowed humans yet another avenue to pursue one of their most coveted activities.

Artificial Intelligence is a field in Computer Science whereby intelligent agents are developed and studied in relation to their environments. In one of the areas within this field, agents are tasked with comprehending their environment and making decisions to achieve a maximum outcome or the most reward. Agents in one way or another are given or develop a model (state) of the world. Based on what an agent perceives as the state of the world, an agent makes a decision (performs an action). Note that even no action is a perfectly valid action. Over time, an intelligent agent has the opportunity to develop a 'policy' of what actions are best based on the state of the world.

Memory and perception are intimately related. Humans have more or less five senses, depending on a person's physical capacity, which are: sight, hearing, touch, taste, and smell. Based on these external stimuli, humans develop an internal model of the world. These models are not guaranteed to be accurate or precise. One person's model of the world may be a completely different than someone else's. For a person that has not seen a lion for example, that creature does not exist in his/her model. This situation may or may not be a perfectly valid model. If a person is not in an environment where lions occur, then it is not necessary to model the creature. For environments where lions do occur, a person may suffer dire consequences for not having that in his/her model. In less severe circumstances, a person will have to re-evaluate his/her internal model of the world. Human memories can be thought of as a collection, often visualized of as a table, of recordings of models and of states of the world. This state table can become quite huge. Often a human's model of the world is not precise in relation to reality. A human must make conscious and/or un-conscience decisions about what to include in his/her model of the world. In some cases, information like sights or sounds may not be distinguishable to some humans. These same complexities are applicable to artificially intelligent agents. Design of AI agents involves much of the following: making decisions about what to keep in an agent's state information, how long to store state information, how to represent state, how to handle physically memory limitations, what external stimuli an agent needs, the cost-benefit of processing information, etc.

Focus of Attention involves the filtering of data into a more useable form. For an AI agent, and humans for that matter, too much information is not useful and is often associated with prohibitively high computation times and memory storage. Two main facets of Focus of Attention are the filtering and selection of information and also the memory management application of deciding and strategizing of what information to hold onto and when. This thesis is associated with the memory management facet of Focus of Attention and is related to and expanding the work of Srividhya [Rajendran 2003]. The work in this thesis uses Function Approximation with Reinforcement Learning to tackle memory and computation complexities of exponentially large state spaces. More specifically, this work can be thought of as a Partially Observable Markov Decision Process (POMDP) treated as a Markov Decision Process (MDP) that simulates short-term memory in humans. See chapter three in this thesis for background on POMDP and MDP. The intelligent agent is not in a fully observable environment but must act and make decisions as if it were. The agent's state information contains a perceptual memory block and a configurable number of auxiliary memory blocks. The perceptual memory block stores the agent's previous action and its result(s). The agent may choose to store what's in the perceptual memory block into an available auxiliary memory block. Different action schedules for the timing of this memory action are being exposed and evaluated here. Each memory block contains a tuple of state information: action, parameter(s), and result(s). Thus in the case of the blocks world domain used as an example domain in the experiments in this thesis, the agent's state information contains whether or not the agent is holding something and memory blocks (a sequence of tuples). Because the state information contains a sequence of tuples, the memory complexity is of the order of product of sequences, which is practically exponential. Since the agent is actively choosing whether or not to hold something in memory, the agent is learning memory management. The agent is also learning Focus of Attention, because it is learning which actions and features to pay attention to and when. The challenge for the agent is to perform and remember sequences of actions in order to accomplish a task. The task can be arbitrary, and therefore this research has applications in many other areas.

The experiments in this thesis focus on the task of block copying. There is a tower of blocks the agent must copy. There are also other blocks on the floor. The agent must stack the free blocks the same way as the tower of blocks it is trying to copy from. Since the agent is limited in memory, the agent must look back and forth between the stacked tower blocks and the blocks on the floor. Once an agent has learned the tower, it no longer needs to look at the tower and can proceed directly to stacking the blocks on the floor to get its reward. To prevent this, the tower is randomized after each episode of the simulation. In such a way, the agent is forced to learn memory management and Focus of Attention by having to observe each new stacking of the tower of blocks to be copied. In most representations of the simulation, the agent's actions have parameters. For example, 'move Y W' would mean 'move to the yellow block with the white background'. In such a scenario, the number of actions is a function of the number of block colors and background colors and the number of base actions ('move' or 'pick' for example). A memory block can contain a tuple of the fore-mentioned list plus an action's result(s).

The number of possible states and possible actions are related to the number of block colors and the number of auxiliary memory blocks an agent has available. Because the state space is a sequence of tuples, the memory cost complexity is exponential. Trying to store a table of state-action pairs, known as a Q-Table, is prohibitive for higher numbers of block colors and auxiliary memory blocks. Thus, Function Approximation is needed and used in this thesis project to mitigate the memory requirements needed to store state information.

1.2 Related Work

Sutton and Barto [Sutton and Barto 1998] have written an excellent book on Reinforcement Learning named, “Reinforcement Learning: An Introduction”. This book is often used when describing and learning Reinforcement Learning. The book covers topics such as Markov Decision Processes, Dynamic Programming, Monte Carlo Methods, Temporal Difference Methods, Eligibility Traces, and Function Approximation. This thesis uses methods and concepts from Reinforcement Learning to learn policies of Focus of Attention. Focus of Attention is only the application or avenue in which Reinforcement Learning is used.

Sutton [Sutton 1996] does a terrific job describing and explaining Function Approximation using coarse coding. Function Approximation, described in Sutton’s work in 1996 and in the book by Sutton and Barto [Sutton and Barto 1998], has been used explicitly in this thesis project to great success. The exponential state space this thesis copes with could only be managed by approximating state. The state space simply grows too quickly to be stored in a table.

Li et al. [Li et al. 2007] write directly on Focus of Attention. In their work, they introduce the concept of focused learning and the notion of state importance. They show experimentally that focusing learning on certain states can lead to better learning performance. They use state visitation probabilities and the concept of advantage introduced by [Baird 1993]. Their work, however, is more related to the selection facet of Focus of Attention. In their case, they are being selective of which states are most advantageous to learn on.

Daw [Daw 2005] uses neural networks in his coined approach of Attention-Gated Reinforcement Learning (AGREL). In his approach, he works to associate actions with input patterns. Although interesting and related to Focus of Attention, his approach is more related to the selection and exclusion of external stimuli and not memory management.

McCullum [McCullum 1994] tries to address the hidden state problem through an approach he coins as instance-based state identification. He recognizes hidden state as when “the agent’s state representation is non-Markovian with respect to actions and utility”. A problem is said to be Markovian when the next state only depends on the current state. If an agent’s history is relevant to determine the next state, then such information is needed in the current state’s information. Also known as the Markov Assumption, the current state’s information can be assumed to contain all previous states’ information in order to determine the next state.

> If the agent’s perceptual system produces the same outputs for two world states in which different actions are required, and if the agent’s state representation consists only of its percepts, then the agent will fail to choose correct actions. [McCullum 1994]

McCullum’s approach extends state information online in order to break state ambiguity. He does this to make the problem Markovian. If an action in a state has inconsistent or ambiguous utility, then the problem is considered to be Non-Markovian. McCullum augments his states with temporary or short-term memory to distinguish states. His work works toward establishing finer granularity of a state space. His work is similar to the work done in this thesis and works towards similar goals, but there are fine distinctions. His work assumes a fair amount of perceptual information and external stimuli, while the work done in this thesis work does not. This thesis work also does not attempt to add auxiliary memory blocks online. The reason for this is that the number of auxiliary blocks the agent has available affects both the action space and the state space. His approach uses memory to disambiguate states, while this thesis work works toward forcing the agent to learn memory management of available memory and what actions are important to keep in memory and when. The largest difference is, however, that in this work the state augmentation is treated as an active decision process where the agent has to learn to actively augment a state with past information rather than as a passive process that splits states by augmenting them with extensive, detailed information about past experiences which uses a more extensive state representation.

As mentioned in the introduction, this thesis is related to and expanding the work of Rajendran [Rajendran 2003] in relation to the block copying task. Besides using additional, different software and control architectures, this thesis work varies from hers in a number of ways. Her work uses a Q-Table and quickly runs into memory limitations for higher numbers of blocks and auxiliary memory blocks. This thesis work uses Function Approximation, which largely addresses the memory problem. Her work uses a Boltzmann Soft-Max Distribution for action selection, while this thesis uses an Epsilon-Greedy approach. As mentioned in Rajendran's listed future work, a later representation of experimentation (Representation 3) in this thesis successfully handles copying a block tower with duplicate block colors using counting and handles orientations in space of: up, down, left, and right. Great time and effort has been taken in this thesis to make the number of auxiliary memory blocks configurable. This allows an experimenter to change the number of auxiliary memory blocks used in a configuration file, so that the experimenter may see and compare the results of using different numbers auxiliary memory blocks while attempting to learn a specified block tower configuration. Because the number of auxiliary memory blocks and number of blocks can vary based on an experimenter's configuration, the implementation used in this thesis must dynamically deduce and handle the state and action spaces to be used for experimentation. Different decision protocols are used in this thesis that relate to how and when the agent performs memory actions. This thesis also implements three different representations of implementation. In these representations, the agent interacts with the environment in different ways. The state and action spaces are different and in one representation used a completely different way. The representations are explained in the next chapter.

Chapter II

Approach Taken

2.1 Learning Methodology

The approach taken in this thesis is to develop an agent with an adaptive internal state representation that allows it to learn to focus its attention through the learning of actions, perception, and memory management strategies. The latter two here allow the agent to actively augment its own state knowledge and with this the state used for decision-making. In particular, due to the agent's limited memory, the learned policy has to decide which of the percepts to remember (i.e. to include in the agent's state) and which ones are available. The overall learning algorithm the agent uses is Q(λ) with eligibility traces. The action selection method used is Epsilon-Greedy. The approach treats a POMPD problem like an MDP. Due to its perception and memory limitations, the agent is not in a fully observable environment but does act and make decisions as if it were in a fully observable environment where its internal state contains all relevant information. The agent's state is made up of what is in its memory and whatever its external stimuli are. In other words, the state of the agent is from the agent's perspective. Whereas in a fully observable state space for the block world domain used here, the agent would be aware of where all the blocks are on the table at each moment in time as well as where the agent is in the environment. The agent here is only aware of actively pursued and remembered percepts.

2.2 Actions

The agent's action set is comprised of physical actions and memory actions. Physical actions are actions that effect the environment and the agent's position in it as well as what aspects of the environment the agent perceives. Memory actions are actions that affect the agent's auxiliary memory blocks. Memory actions allow the agent to copy what's in its perceptual memory into a specified memory block it has available to it. 'Mem 0' is a No-Op, which means that agent chooses not to copy what's in perceptual memory into an auxiliary memory block. ‘Mem 1’ would mean the agent shall copy what’s in perceptual memory to the auxiliary memory block labeled ‘1’ or first auxiliary memory.

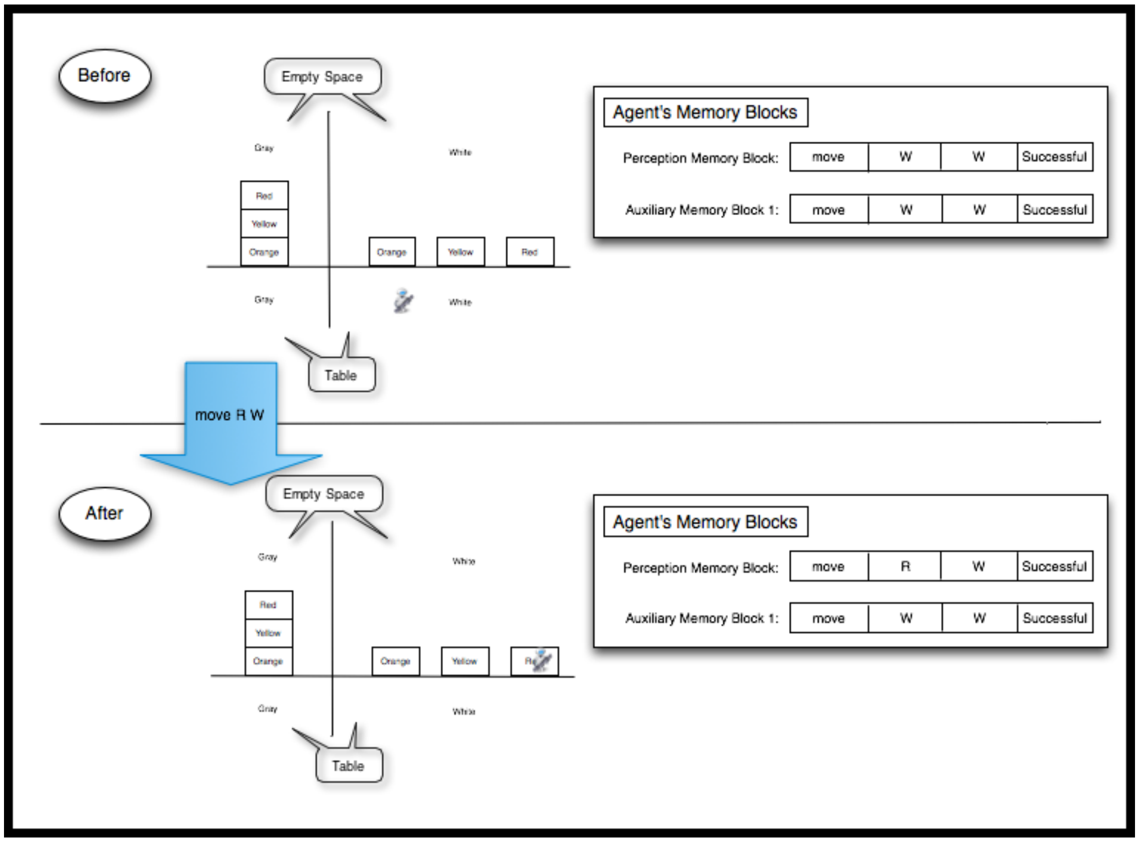


Figure 1-1: Agent Physical Action Example

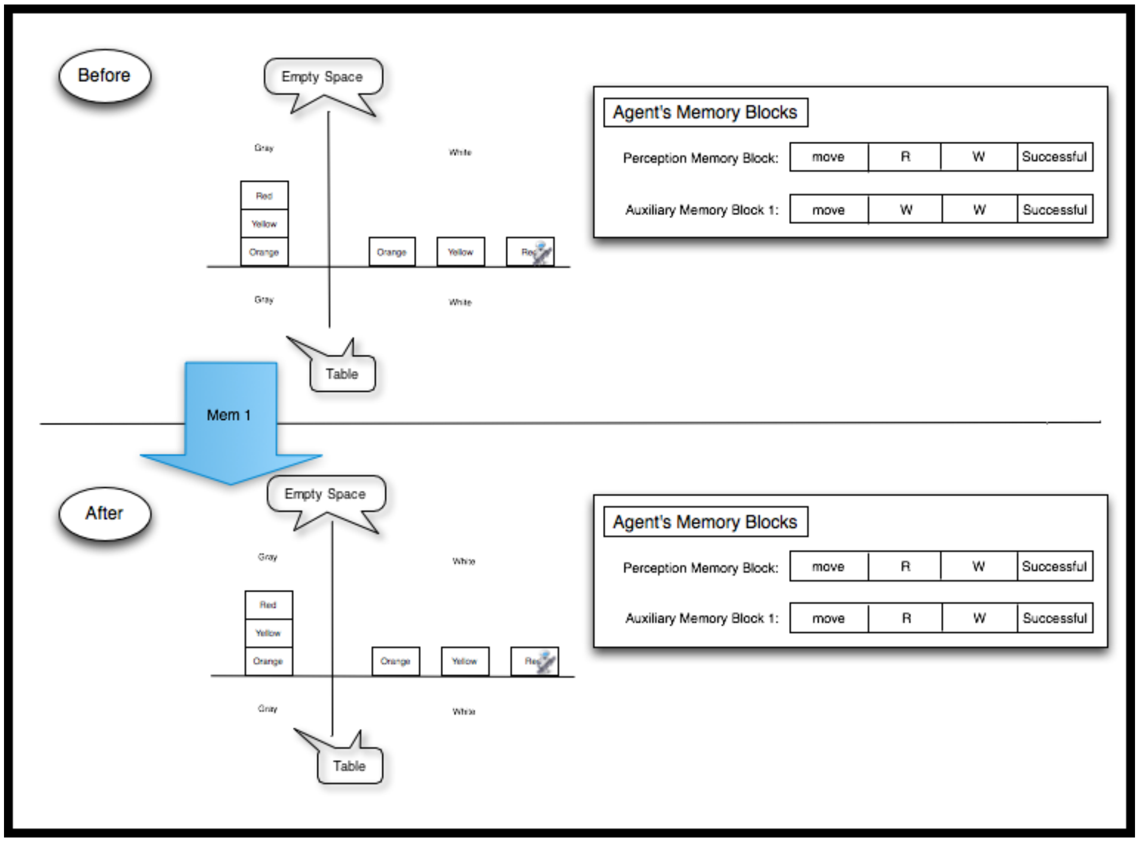


Figure 1-2: Agent Memory Action Example

Figure 1-1 and Figure 1-2 show example sequences of physical and memory actions.

2.3 Representations

To investigate the effect of different action and representation capabilities on learning, three different representations have been investigated here. The agent's exact state information varies and has different meaning depending on the representation of experimentation being explored. The action space the agent has available also varies depending on the representation of experimentation being explored. The representations used are detailed below in the context of the used block-copying task:

Representation 1:

- The agent knows whether it is holding a block or not.
- The agent knows its position.

- The agent may move one position away from its current position at a time.
- The agent must specify parameters of physical actions, which represent specific perceptual features that it is processing.
- Physical actions include the perceptions actions: top, bottom.
- Physical actions available: move, pick, top, bottom, and stop.
- Memory actions available: Mem0, . . ., Mem[num_auxiliary_mem_blocks].
- Physical actions only have one result: true or false.
- The absolute position of the agent is determined by the agent's 'atColor' and 'atBackColor'.

Representation 2:

- The agent knows whether it is holding a block or not.
- The agent does not know its position.
- The agent may move directly to a specified position at any time.
- The agent must specify parameters of physical actions, and thus which features are to be processed.
- Physical actions include the perceptions actions: 'top' and 'bottom', which allow the agent to see up and down a stack of blocks.
- Physical actions available: move, pick, top, bottom, and stop.
- Memory actions available: Mem0, . . ., Mem[num_auxiliary_mem_blocks].
- Physical actions have two results: 'physicalResult' and 'perceptionResult'.
- The absolute position of the agent is determined by the agent's 'atColor' and 'atBackColor'.

Representation 3:

- The agent knows whether it is holding a block or not.
- The agent does not know its position.
- The agent may move one position away at a time: up, down, left, right.

- The agent does not specify parameters to physical actions, rather the action returns the relevant features found.
- The agent has in each auxiliary memory block a counter that can be incremented, reset, or left alone (no-op) during a memory action.
- Physical actions do not include perception actions.
- Physical actions available: move_up, move_down, move_left, move_right, pick, top, and stop.
- Memory actions available: Mem0, . . ., Mem[num_auxiliary_mem_blocks]_['MemCounter_NoOp'||' MemCounter_Increment'||' MemCounter_Reset'].
- Physical actions have two results: 'physicalResult' and 'perceptionResult'.
- The absolute position of the agent is determined by the agent's 'Position Number' and what block id the agent is at. If the agent is at the table, the block id the agent is at is set to -1.
- Duplicate Block Colors are allowed. For example, the blocks in the block tower to be copied can have two red blocks and there can be two red blocks on the floor.

There are a few notes on actions to mention. Actions are associated with a step cost. The agent is aware of the total reward acquired. The initial total reward acquired is set to zero at the beginning of each episode of experimentation. Episodes are explained in chapter four, Agent-Environment Architecture. For each action taken, the agent receives the step cost. In all experimentations done in this thesis, the step cost used is -1. The agent has the physical action 'stop' available. 'Stop' is used to let the agent give-up on the block-copying task. For agents that never see the final goal, the agent will learn to 'stop' immediately rather than continually acquiring more and more negative reward. To encourage the agent not to stop too early and seek goal of the simulations (resulting in a high positive reward), the agent's 'stop' action will not work early in episodes. Representations 1 and 2 include perception actions: top and bottom.

These actions are used to query the environment's block location. It is the same as asking, 'Is the blue block with the white background above the agent's current location?' or ' Is the blue block with the white background below the agent's current location?'. These perception actions provide two functions:

- Allow the agent to know whether a particular move will succeed or fail.
- Provide a means to store block location information in an agent's memory.

Note that the pick action will pick up a block if given the right parameters and the agent is not holding anything. A pick action will drop a block if it is holding a block and the parameters given are correct. In Representation 3, the agent does not use the counter in its perceptual memory block. In Representation 2 and Representation 3, physical actions have two results. This is to help the agent differentiate actions that fail because the given parameters are invalid from actions that fail because the action is not possible. A block can't move because another block is on top of it is a great example. The agent specifies the correct parameters, "pick B W", but the block is immoveable because another block is on top of it. This will result in the agent receivable a 'true' for the perception result and a 'false' for the physical action result.

2.4 Decision Protocols

For each representation, there are different decision protocols. These decision protocols affect how and when memory actions are performed. Decision protocols are as follows:

- simultaneous: The agent may take a memory action at the same time as a physical action.
- alternating: The agent may take a memory action only after a physical action.
- asynchronous: The agent may choose either a physical action or memory action at any time.

2.5 Memory Blocks

The content of what a memory block can hold has been eluded so far and not yet explained. As mentioned in the introduction of this thesis, a memory block can contain a tuple consisting of an action's name ('move' for example), a foreground color ('B' for blue for example), a background color ('W' for White or 'G' for Gray), plus an action's result(s). In Representation 1, there is only one action result stored. In Representations 2 and 3, there are two results stored (the perception result and the simulation result). In Representation 1 and 2, the foreground color and background colors are the colors the agent specified as parameters to a given action. For example, if agent wants to move to the blue block with the white background, the agent will attempt to 'move B W'. In the perceptual memory, the agent will now have a tuple of ('move', 'B', 'W', true) assuming the result is true and the decision protocol is Representation 1. Representation 3 behaves differently here. The agent does not specify parameters to physical actions. Recall that Representation 3's physical actions are: move_up, move_down, move_left, and move_right, and pick. What gets stored in the memory block in this case is the foreground and background colors the agent found performing the action. For example, the agent tries to move left with 'move_left'. What gets stored in the perceptual memory block is what the agent found moving left and its results. If the agent is trying to 'pick' up a block above, the perceptual memory block will store what the agent picked up. It should be noted that in all representations, what the agent is trying to pick up is always above itself. To pick up a block on the table, the agent must first move to the table ('move W W') and then do a 'pick B W' for example. To successfully place a block the agent is holding on top of another block, the agent must do a 'pick NOTHING W' for example while on the topmost block of a stack.

2.6 Blocks in the Environment

The tower blocks will always have a background color of 'G' for Gray in all experimentations performed in this thesis. Similarly, stackable blocks will always have a 'W' for

White background color. In all representations, tower blocks (the blocks being copied) are immoveable. Blocks have properties. For the tower blocks, their moveable property is always set to false. For a stackable block that has a block placed on top of it, its moveable property is set to false. It is unset when the above block is picked up again. Make careful note that stackable blocks, the blocks having the White background, are not allowed to be moved to the Gray background. This is to prevent ambiguity and prevent the agent from confusing tower blocks with stack blocks. Representations 1 and 2 do not handle duplicate block colors. This allows those representations to have a unique mapping of blocks. For example, 'move Y W' would attempt to move the agent to the yellow block with the white background color. There is only one yellow block with background color of white. Because of this unique mapping, foreground and background colors have become the coordinate system for Representations 1 and 2. The following is a three-block setup diagram used in Representations 1 and 2:

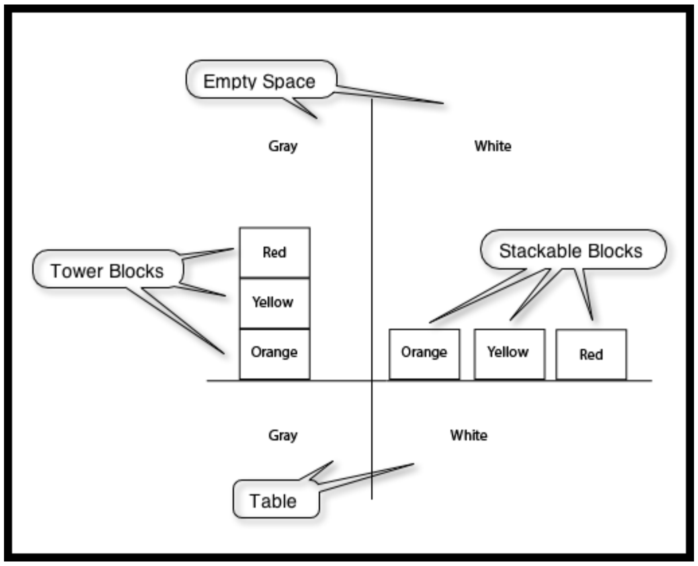


Figure 1-3: Representations 1 and 2 Three-Block Setup Example

To move to the table from a block at a bottom of the stack, the agent would do a 'move W W'. For the agent to move from one background color to another, the agent must be at the table ('W W' or 'G G'), not holding a block, and issue a 'move'. For example, to successfully move to the

Gray background color from the White background color, the agent must issue a ‘move G G’ while the agent is at ‘W W’.

Representation 3 has a slightly different coordinate system. Representation 3 allows duplicate block colors, so there is no longer a unique mapping of foreground and background colors to a block. To accommodate this, the environment must now use block ids to uniquely identify a block. The agent is completely unaware of the block ids. However, the environment ‘knows’ what block id the agent is at. If the agent is at the table, the environment sets the block id the agent is at to -1. Also, note that in Representation 3 the agent can no longer move from the table to a specified block color. ‘move O W’ is now illegal and not a recognized physical action. There are two main reasons for this. The first reason is that there can be ambiguity if there is more than one block of the same color next to the agent. The second reason is that orientation in space is desired for Representation 3. The agent now has the ability to move: up, down, left, and right. This forces the need for the environment to store in some way what blocks are left and right of each other. The term ‘position’ is introduced to facilitate this need. Because blocks in the block tower are always unmovable, all tower blocks are assigned the position of zero. Each stackable block is initialized to be on the ground and to it own position number. The maximum position an agent or block can be at is the total number of stackable blocks. For example, if there are three stable blocks, position three will be the last position available in the environment. It is important to note that the agent is not aware of these locations. Instead the agent has the ability to use its counters to keep track of its and the blocks’ positions. To illustrate this, see below diagram:

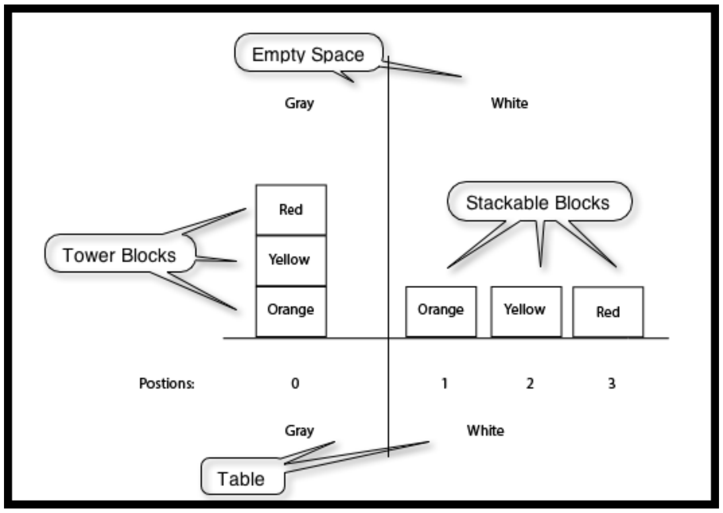
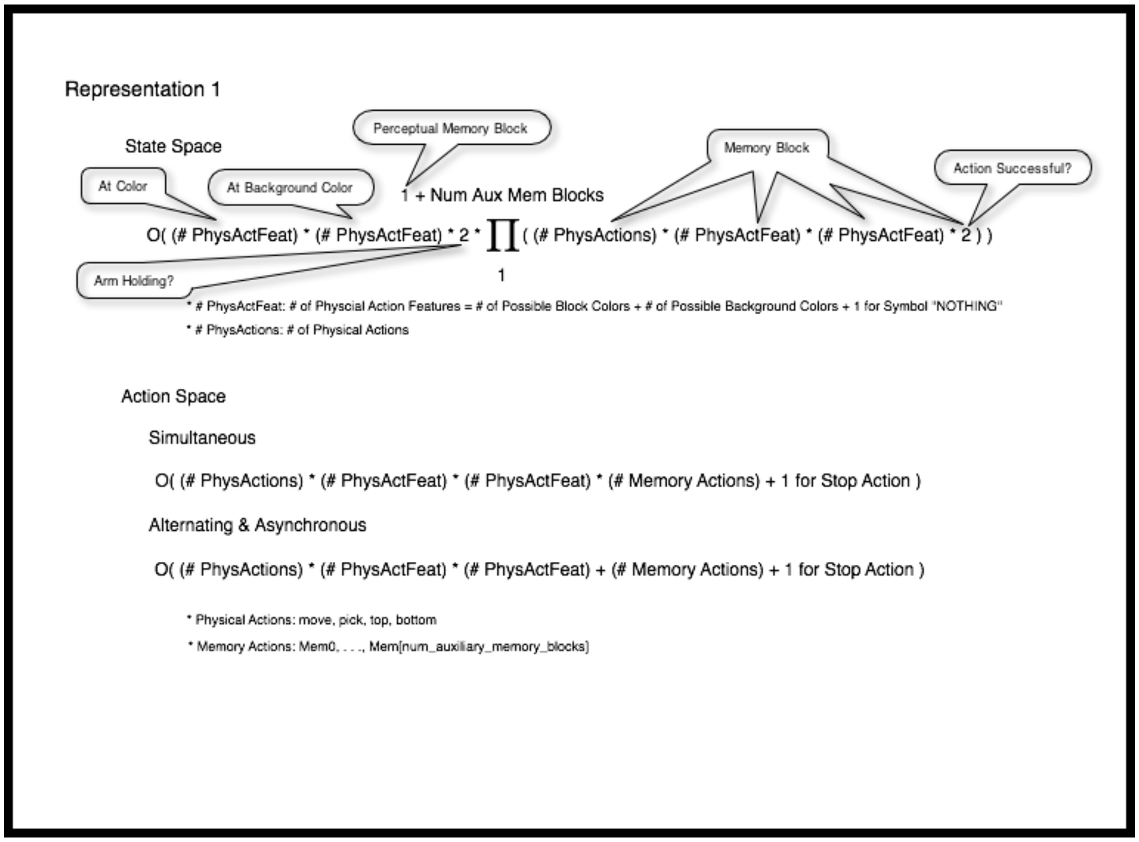


Figure 1-4: Representation 3 Three-Block Setup Example

2.7 State and Action Space Complexities

There is merit in noting the state and action space complexities for the different representations and decision protocol combinations. This information is relevant to explaining why an agent does or does not do well under a particular representation and decision protocol.



Representation 1

State Space

$$O\left((\text{\# PhysActFeat}) * (\text{\# PhysActFeat}) * 2 * \prod_{1}^{1 + \text{Num Aux Mem Blocks}} \left((\text{\# PhysActions}) * (\text{\# PhysActFeat}) * (\text{\# PhysActFeat}) * 2 \right) \right)$$

* # PhysActFeat: # of Physcial Action Features = # of Possible Block Colors + # of Possible Background Colors + 1 for Symbol "NOTHING"

* # PhysActions: # of Physical Actions

Action Space

Simultaneous

O((# PhysActions) * (# PhysActFeat) * (# PhysActFeat) * (# Memory Actions) + 1 for Stop Action)

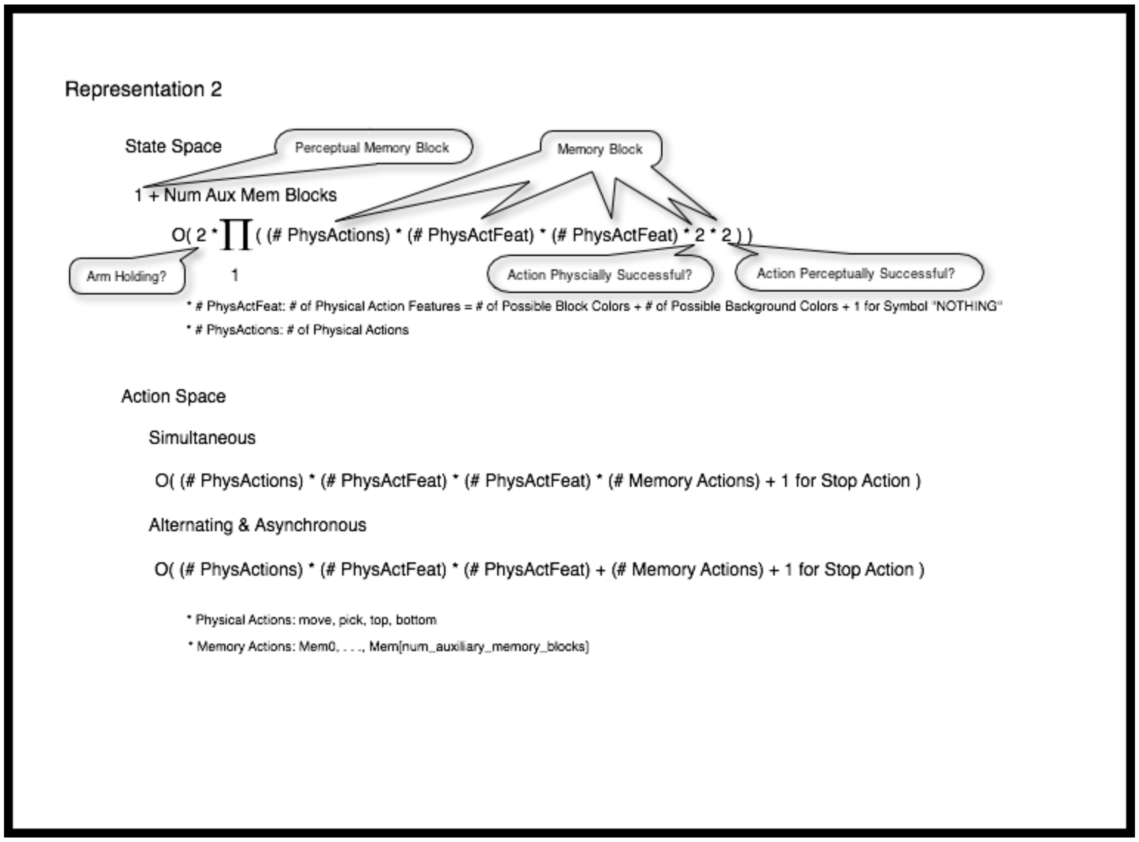
Alternating & Asynchronous

O((# PhysActions) * (# PhysActFeat) * (# PhysActFeat) + (# Memory Actions) + 1 for Stop Action)

* Physical Actions: move, pick, top, bottom

* Memory Actions: Mem0, . . ., Mem[num_auxiliary_memory_blocks]

Figure 1-5: Representation 1 State and Action Space Complexities



Representation 2

State Space

$$O\left(2 * \prod_{1}^{1 + \text{Num Aux Mem Blocks}} \left((\#\ \text{PhysActions}) * (\#\ \text{PhysActFeat}) * (\#\ \text{PhysActFeat}) * 2 * 2 \right)\right)$$

Perceptual Memory Block
Memory Block
Arm Holding?
Action Physcially Successful?
Action Perceptually Successful?

* # PhysActFeat: # of Physical Action Features = # of Possible Block Colors + # of Possible Background Colors + 1 for Symbol "NOTHING"
* # PhysActions: # of Physical Actions

Action Space

Simultaneous

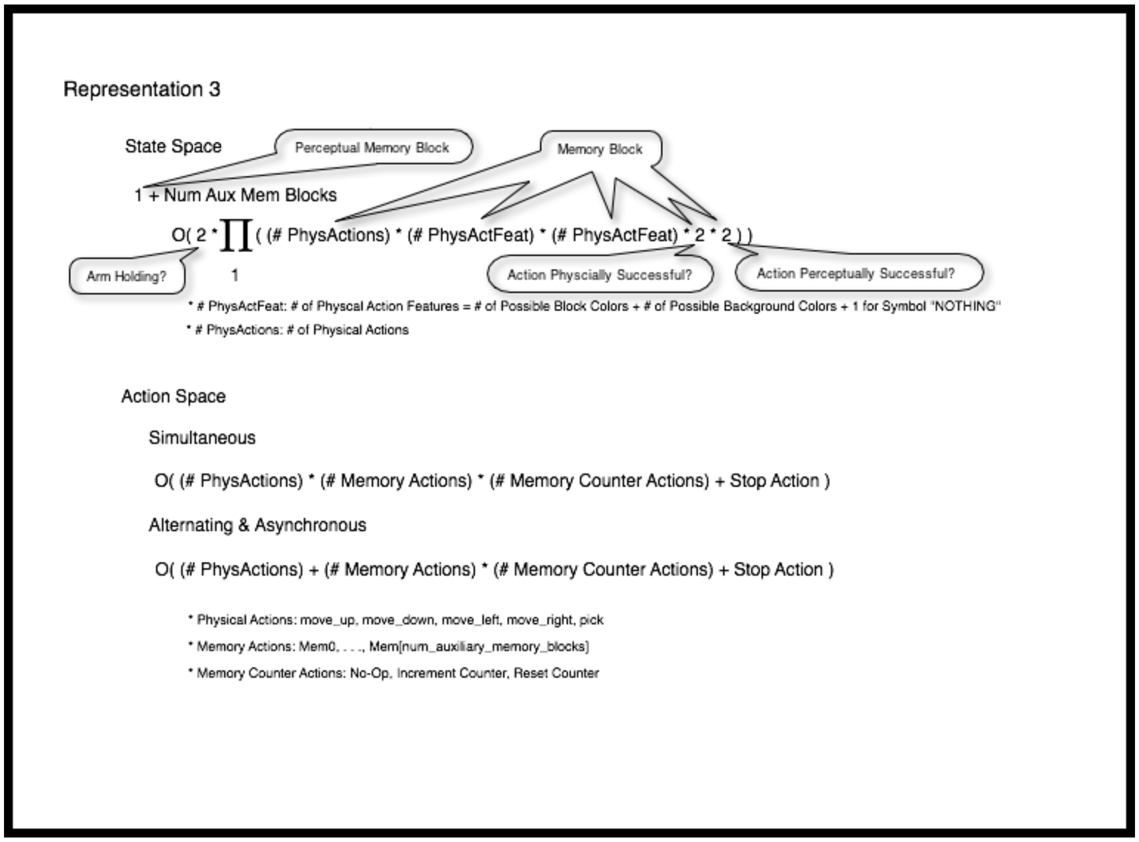
O((# PhysActions) * (# PhysActFeat) * (# PhysActFeat) * (# Memory Actions) + 1 for Stop Action)

Alternating & Asynchronous

O((# PhysActions) * (# PhysActFeat) * (# PhysActFeat) + (# Memory Actions) + 1 for Stop Action)

* Physical Actions: move, pick, top, bottom
* Memory Actions: Mem0, . . ., Mem[num_auxiliary_memory_blocks]

Figure 1-6: Representations 2 State and Action Space Complexities


Perceptual Memory Block
Memory Block
Arm Holding?
Action Physcially Successful?
Action Perceptually Successful?

Representation 3

State Space

$$O\left(2 * \prod_{1}^{1 + \text{Num Aux Mem Blocks}} \left((\#\ \text{PhysActions}) * (\#\ \text{PhysActFeat}) * (\#\ \text{PhysActFeat}) * 2 * 2 \right) \right)$$

* # PhysActFeat: # of Physcal Action Features = # of Possible Block Colors + # of Possible Background Colors + 1 for Symbol "NOTHING"

* # PhysActions: # of Physical Actions

Action Space

Simultaneous

O((# PhysActions) * (# Memory Actions) * (# Memory Counter Actions) + Stop Action)

Alternating & Asynchronous

O((# PhysActions) + (# Memory Actions) * (# Memory Counter Actions) + Stop Action)

* Physical Actions: move_up, move_down, move_left, move_right, pick

* Memory Actions: Mem0, . . ., Mem[num_auxiliary_memory_blocks]

* Memory Counter Actions: No-Op, Increment Counter, Reset Counter

Figure 1-7: Representations 3 State and Action Space Complexities

All complexities use Big-O Notation, which is used to give upper bounds of complexities. In all representations, the state space complexity is of the order of product of sequences, or practically exponential. Representation 1 uses location information in its state space: the 'at color' and 'at background color' parameters. Representations 2 and 3 do not use location information. Representations 2 and 3 also have two action results to store in memory blocks. In the 'simultaneous' decision protocol of representations 1 and 2, actions are a combination of physical actions, action parameters, and memory action. One example would be 'move_Y_W_Mem1'. Representation 3 does not use parameters for physical actions and as a result has a much smaller action space.

Chapter III

Background

3.1 Reinforcement Learning Model

The notion of Reinforcement Learning involves learning through interaction.

> Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told what actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. [Sutton and Barto 1998]

The agent is interacting with a dynamic environment. For example, if an agent's environment has a cup on a table and the agent moves the cup, the agent has effected the environment. If the agent were to break the cup, the agent must continue on with a cup-less environment. Based on the agent's actions, the environment will provide the agent feedback. The agent typically has a goal to accomplish. The goal may be defined in a number of ways or not at all. In some learning approaches used in Artificial Intelligence for example, the agent is not strictly required to know or have a goal. Here, we define a goal as a desired final state of the agent and environment. As the agent interacts with the environment, the environment provides feedback to the agent.

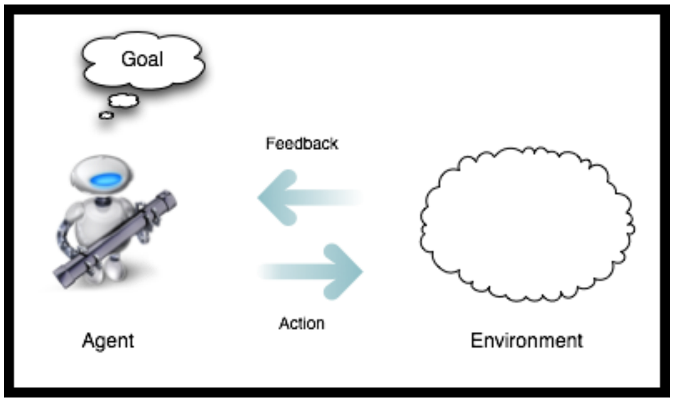


Figure 3-1: Agent-Environment Interaction

Reinforcement Learning is distinguished as a learning problem and not as a learning method. There are different methods for learning reinforcement learning such as Dynamic Programming and Monte Carlo Methods, but they all fall under the umbrella of Reinforcement Learning.

> Reinforcement learning is different from supervised learning, the kind of learning studied in most current in machine learning, statistical pattern recognition, and artificial neural networks. Supervised learning is learning from examples provided by a knowledgeable external supervisor. This is an important kind of learning but alone is not adequate for learning from interaction. In interactive problems it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act. [Sutton and Barto 1998]

The agent has no teacher or supervisor. The only feedback the agent gets is from the environment. Like humans, the agent makes choices and receives consequences. And like humans, the agent must learn from its own experience.

Reinforcement is characterized by four main elements: policy, reward function, a value function, and possibly, a model of the environment. A model of the environment is not necessarily available. From Sutton and Barto [Sutton and Barto 1998]:

- A policy defines the learning agent's way of behaving at a given time.
- A reward function maps each perceived state (or state-action pair) of the environment to a single number, a reward, indicating the intrinsic desirability of that state.
- A value function specifies what is good in the long run. The value of a state is the total amount of reward the agent can expect to accumulate over the future, starting from that state.
- A model of the environment is something that mimics the behavior of the environment.

Below is a list of commonly used terms in Reinforcement Learning:

Table 3-1 Common Reinforcement Learning Terms

A	action space
D	state space distribution

Table 3.1 – Continued

γ	discount factor in real number range between 0 and 1
$L(\pi)$	policy loss
π	policy mapping S to A
$\pi_Q(s)$	greedy policy with respect to Q
$\pi_V(s)$	greedy policy with respect to V
π^*	optimal policy
Π	policy space
Q(s,a)	action-value function estimate
Q^π(s,a)	action-value function
Q*(s,a)	optimal action-value function
R_t	reward received at time t
R_0	cumulative discounted return
S	state space
t	time step index
V(s)	state-value function estimate
V^π(s)	state-value function
$V^*(s)$	optimal state-value function
$V(\pi)$	policy value

The start state is denoted s_0. The start state belongs to the state space. This is denoted $S \ni s_0$. The current state at time 't' is denoted s_t. s_t also belongs to S, $S \ni s_t$. An action at time 't' is denoted a_t. A reward at time 't' is denoted r_t.

The general reinforcement-learning model is given below:

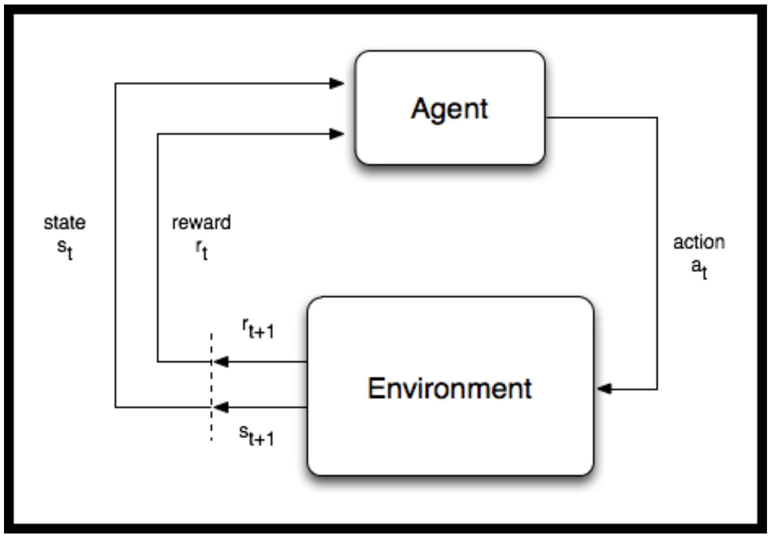


Figure 3-2: Reinforcement Learning Model

In this model, the agent makes an action and the environment returns a reward and a new state of the world. The agent is tasked with choosing actions that increase its total reward over time. Over time, the agent develops a policy, denoted π. The policy is a mapping from states to probabilities of selecting each possible action [Sutton and Barto 1998]. The goal of an environment is an important reward given to the agent from the environment for reaching a particular state. Overall, the agent's objective is still to maximize reward. In order to obtain maximum reward, the agent will in turn seek the goal if its reward is sufficiently high.

There may be many states between an agent's initial state and established goal. Because of this, the agent must work with expected return from states. That is, if the there is sequence of states that lead to the goal, each state leading up to the goal can contain a discounted return.

From Sutton and Barto [Sutton and Barto 1998], the expected discounted return is given by the following equation:

$$\boxed{R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}}$$

Figure 3-3: Expected Discounted Return Calculation

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate.

There is a special property of environments known as the Markov property. A state is represented by the information available to the agent. For an environment to be Markov, the state signal an environment returns to the agent must retain all relevant information to the agent. This relevant information thus includes any previous history needed. An environment is said to maintain the Markov property if the next state only depends on the current state and the action the agent takes. The Markov Assumption is that any previous history needed does not provide any additional information about future states.

In Reinforcement Learning, there are many ways to learn the optimal policy. If the agent is limited in the number of time steps to act, the agent must learn to act within the limited time frame. However, if the agent has all the time in the world to act, the agent's optimal policy may be different from the one learned in limited time. These contrasting policies are known as The Finite Horizon Model and The Infinite Horizon Model. There is yet another model known as The Average Reward Model. In The Average Reward Model, the agent will try to optimize the average reward gained over time. In the experimentation done in this thesis, the Infinite Horizon Model is used. Because of how auxiliary memory blocks an agent is configured to have and thus how extensive and expressive its state representation is, the agent may or may not be able to learn how to reach the given goal. To prevent the agent from potentially getting stuck indefinitely, a maximum number of steps are enforced and the agent has the opportunity to stop after so-many steps.

3.2 Markov Decision Processes

A reinforcement-learning task that maintains the Markov Property is exactly modeled a Markov Decision Process, or MDP [Sutton and Barto 1998]. An MDP is characterized by a tuple of the following four entities:

- S, a set of states
- A, a set of actions
- T, a state transition probability function
- R, a reward function

The state transition probability function, T, takes as input a state and action and returns a new state. The reward function, R, takes as input a state and action and returns a return. Value iteration, policy iteration, Q-Learning, and Sarsa are some methods used to find an optimal policy given an MDP.

3.3 Partially Observable Markov Decision Processes

Fakoor [Fakoor 2012] writes, “A POMDP is an extension of the MDP and is defined by a tuple of six entities: <S, A, Z, R, T, O>. Here S is the set of world states denoted by s_t. Z is the set of observations. A is the set of actions, and T is the state transition probability function.”. T is given as $T(s,s',a) := P(s_t = s' \mid s_{t-1} = s, a_{t-1} = a)$. O is given as $O(s,z) := P(z_t = z \mid s_t = s)$. POMDPs have incomplete observations of the environment. The notion of history is needed. History, in this context, is a sequence of observations and actions. Policies are defined in a POMDP as mapping of histories to actions. Histories can be infinitely long. To cope with this it is usually enough to use a sufficient statistic to represent history, also know as a belief state. The agent calculates a ‘belief state’, and from there the agent can implement various methods to find an optimal policy.

3.4 POMDPs Treated As MDPs

POMDPs treated as MDPs are problems where is agent is not in a fully observable environment but decides to act and make decisions as if it were. State is treated as the state of the agent in the environment. The actual state of the environment is not fully observable and not used as part of any policy that agent learns. Similarly, belief state is not calculated and observations are used to derive an agent-internal memory state, which is treated as if it were a MDP state and is used to learn and represent the policy. POMDPs treated as MDPs are used to model an agent's internal memory state, which is treated as if it were an MDP state and used to learn and represent the policy. State, used for policy learning, is from the agent's perspective. Only by what the agent has in its memory blocks is the agent aware of the state of the environment. In some representations of experimentation used in this thesis, the agent is also aware of its position independent of what is kept in memory blocks.

3.5 Q-Learning

Q-Learning is an off-policy TD control algorithm in Reinforcement Learning. TD, temporal-different learning, is a category of learning methods in Reinforcement Learning that can learn directly from raw experience without having a model of the environment. Sutton and Barto [Sutton and Barto 1998] write, "TD methods update estimates based in part by other learning estimates, without waiting for a final outcome (they bootstrap)". In one-step Q-learning, the equation is given as:

$$\boxed{Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]}$$

Figure 2-4: Q-learning Equation

where α is the learning rate and γ is the discount rate.

> In this case, the learned action-value function, Q, directly approximates Q*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early

convergence proofs. The policy still has an effect in that it determines which station-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue be updated. [Sutton and Barto 1998]

The algorithm for Q-learning is given as follows:

```
Initialize Q(s, a) arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., $\epsilon$-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha [ r + \gamma \max_{a'} Q(s', a') - Q(s, a) ]$
    $s \leftarrow s'$;
  until s is terminal
```

Figure 2-5: Q-learning Algorithm

3.6 Sarsa

Sarsa is another TD method used in Reinforcement Learning for learning a MDP policy. In contrast to Q-learning being an off-policy algorithm, Sarsa is an on-policy algorithm. On-policy algorithms continually estimate Q^π for the policy and adjust π greedily with respect to Q^π. Sarsa is an acronym for State-Action-Reward-State-Action. The name refers to the Q-value being updated as a function of the tuple of current state, the action chosen, reward received for action chosen, the resultant state after the chosen action is performed, and the next action. The equation is given as:

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

Figure 3-6: Sarsa Equation

where α is the learning rate and γ is the discount rate.

Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ε-greedy policies by setting ε = 1 / t). [Sutton and Barto 1998]

The algorithm for Sarsa is given as follows:

```
Initialize Q(s, a) arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., $\epsilon$-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., $\epsilon$-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha [ r + \gamma Q(s', a') - Q(s, a) ]$
    $s \leftarrow s'$; $a \leftarrow a'$
  until s is terminal
```

Figure 3-7: Sarsa Algorithm

3.7 Eligibility Traces

Eligibility Traces are a device in Reinforcement Learning that acts as a bridge between Monte Carlo methods and Temporal Difference (TD) methods. Monte Carlo methods use averages of sample returns to solve the Reinforcement Learning problem. The symbol 'λ', in TD(λ) algorithms, refer to the use of eligibility traces. Q-Learning and Sarsa can be combined with eligibility traces and referred to as Q(λ) and Sarsa(λ) respectively.

> An eligibility trace is a temporary record of the occurrence of an event, such as the visitation of a state or the taking of an action. The trace marks the memory parameters associated with the event as eligible for undergoing learning changes. When a TD error occurs, only the eligible states or actions are assigned credit or blame for the error. [Sutton and Barto 1998]

Eligibility traces are useful in tasks with long-delayed rewards, such as the block-stacking task approached in this thesis. Eligibility traces require more computation than one-step methods but give the learning agent the opportunity to learn much faster.

3.8 Function Approximation

The value functions for the methods mentioned so far in this background chapter have been assumed to be represented as a table of states and actions. Q-learning calls this a Q-Table. For problems with large state and/or action spaces, the amount of memory needed to store this table of state-action pairs is prohibitive. Function approximation attempts to generalize, or approximate, the value function. Given a state and action, the function will return an approximate value. This is known as value prediction. As opposed to trying to store a table, a function is learned and tweaked to return values for a given state-action pair.

> The approximate value function at time t, V_t, is represented not as a table but as a parameterized functional form with parameter vector $\Theta_{\bar{t}}$. This means that the value function V_t, depends totally on $\Theta_{\bar{t}}$, varying from time step to time step only as $\Theta_{\bar{t}}$ varies. [Sutton and Barto 1998].

Gradient-descent methods are widely used methods for function approximation.

> In gradient descent methods, the parameter vector is a column vector with a fixed number of real valued components, $\Theta_{\bar{t}} = (\Theta_t(1), \Theta_t (2), . . . , \Theta_t (n))$^t (the T here denotes transpose), and $V_t(s)$ is a smooth differentiable function of for all S ∍ s. [Sutton and Barto 1998]

Although exact values are given for $V^\pi(s_t)$ for each s_t, the function approximator has limited resolution and resources, and therefore no Θ_t can give exact values for each and every state. To refine the function approximator, mean square error (MSE) is minimized on the observed examples. Gradient descent methods adjust the parameter vector after each example observed by a small amount in a direction that reduces the most error in the given example.

> This derivative vector is the gradient of f with respect to $\Theta_{\bar{t}}$. This kind of method is called gradient descent because the overall step in $\Theta_{\bar{t}}$ is proportional to the negative gradient of the example's squared error. This is the direction in which error falls most rapidly. [Sutton and Barto 1998]

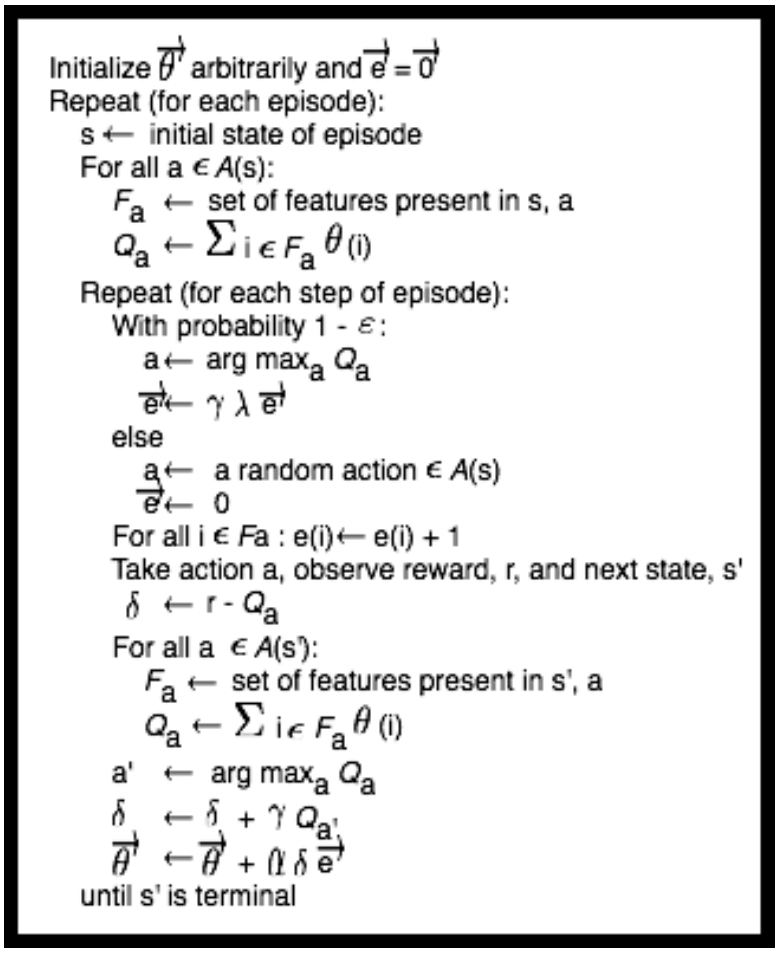
Two popular methods used for gradient-based function approximation are multilayered neural networks using the error backpropagation algorithm and the linear form.

In the linear form, the approximate function, V_t, is a linear function of the parameter vector, $\Theta_{\bar{t}}$. States are represented in terms of features. In coarse coding, the state space is assumed to be continuous. If features were represented as circles in a Venn diagram, a state is where the circles overlap. This representation is known as coarse coding. Tile coding is a form of coarse coding.

> In tile coding the receptive fields of the features are grouped into exhaustive partitions of the input space. Each such partition is called a tiling, and each

element of the partition is called a tile. Each tile is the receptive field for one binary feature. [Sutton and Barto 1998]

One and only one feature is present in each tiling. The number of features present is equal to the number of tilings. Let α be a step-size parameter. Let ‘m’ be the number of tilings. Choosing α = 1/m would result in exactly one-trial learning. Choosing α = 1/(10m) would move one-tenth of the way per update. The number of tilings used affects how closely a function approximator can represent a function. The denser the tiling, the more accurate it is; the computational cost is also higher. The following version of the Q(λ) algorithm is used in the experimentation of this thesis project:



```
Initialize $\vec{\theta}$ arbitrarily and $\vec{e} = \vec{0}$
Repeat (for each episode):
    s $\leftarrow$ initial state of episode
    For all a $\in A(s)$:
        $F_a \leftarrow$ set of features present in s, a
        $Q_a \leftarrow \sum_{i \in F_a} \theta(i)$
    Repeat (for each step of episode):
        With probability 1 - $\varepsilon$:
            a $\leftarrow$ arg max$_a$ $Q_a$
            $\vec{e} \leftarrow \gamma \lambda \vec{e}$
        else
            a $\leftarrow$ a random action $\in A(s)$
            $\vec{e} \leftarrow 0$
        For all i $\in Fa$ : e(i) $\leftarrow$ e(i) + 1
        Take action a, observe reward, r, and next state, s'
        $\delta \leftarrow r - Q_a$
        For all a $\in A(s')$:
            $F_a \leftarrow$ set of features present in s', a
            $Q_a \leftarrow \sum_{i \in F_a} \theta(i)$
        a' $\leftarrow$ arg max$_a$ $Q_a$
        $\delta \leftarrow \delta + \gamma Q_{a'}$
        $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
    until s' is terminal
```

Figure 3-8: A linear, gradient-descent version of Q(λ) with ε-greedy selection policy and accumulation traces

3.9 Exploration Versus Exploitation

There is always a trade off between selecting actions when developing a policy. For an untried action at any given state, one cannot say how good or bad it is to perform the action unless one tries it. The agent will not always have a developed policy. The agent must at times try different actions at a given state. This is known as exploration. In contrast, exploitation would be where the agent performs the best possible action given to it by the agent's policy. The experimenter must decide on an action selection strategy for the agent to use. Depending on the action selection strategy, the agent may explore more at first before following the agent's policy more and more.

In the ε-greedy action selection strategy, the agent uses the best action given by the agent's policy for a proportion of $1 - \varepsilon$ of the steps in an episode and a random action for the rest of the time. This is a simple selection strategy. There are other selection strategies, but this is the one used in the experimentation done in this thesis.

Chapter IV

Agent-Environment Architecture

This chapter describes the high-level agent-environment architecture used in the experimentation of this thesis. Many of the Reinforcement Learning concepts and mechanisms described in the Background are used in the implementation of the experimentation. The 'where' and 'how' these learning mechanisms are used are also described. The goal of experimentation is to learn Focus of Attention in terms of memory management. The application the agent attempts to learn focus in is block-copying. Block-copying is a good venue for this research, because the agent is limited in terms of memory and must learn memory management in order to successfully copy a tower of blocks. The overall strategies used by the learning agent are described in chapter two, 'Approach Taken'. The goal of this chapter is to delineate the software architecture and how reinforcement concepts are used.

4.1 Learning Architecture

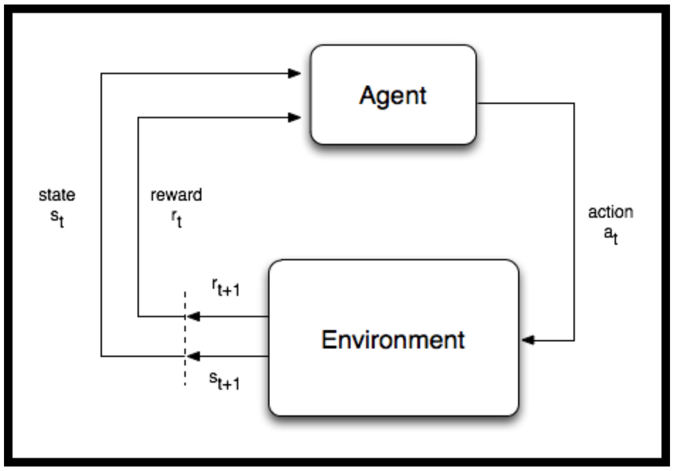


Figure 4-1: Agent-Environment Interaction

The agent and environment are separate entities. The environment is aware of where the agent is and of the block locations. The agent's state information only contains what is in the perceptual memory block, auxiliary memory blocks if any, whether or not the agent is holding a block, and possibly position information depending on the representation of experimentation.

The environment keeps a unique mapping to the blocks. Each block contains properties such as block color, its type (tower block or stackable block), whether it is moveable, whether it is being held, its position information, what's above the block, and what's below the block. Whenever the agent successfully performs an action such as moving a block or picking up a block or dropping a block on top of something, the environment must update the associated blocks' properties. The agent-environment interaction can be likened to that of client-server. See Figure 4-1. The environment contains the state information of the agent. The agent, like a client, gets the state information from the client. Whenever the agent performs an action, the agent makes a request to the environment, like a server, to perform an action and the environment returns the agent's new state information to the agent. If the action performed is successful, the environment is in charge of updating information regarding the agent and the blocks. After an action is performed, the agent queries the environment to see if the goal and sub-goals have been reached. If the goal is reached, the environment will return the goal reward to the agent. If the 'goal state is final' flag is set, the episode will immediately end. The environment will award the agent intermediate reward if a sub-goal is reached, meaning some of the blocks are stacked correctly. Sub-goals must be reached in order. If the agent undoes a sub-goal, the agent will receive negative the intermediate reward.

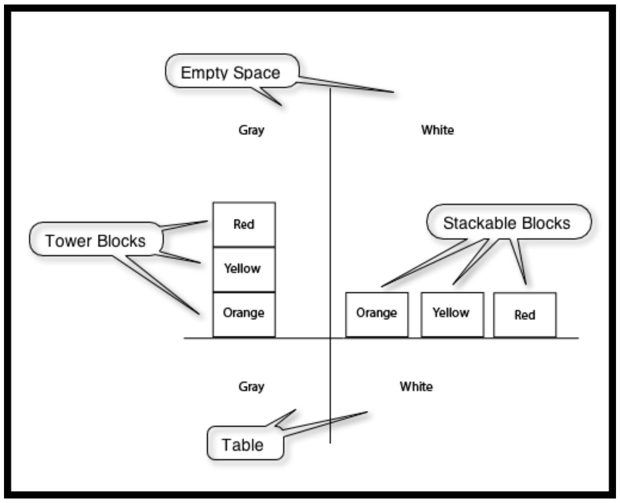


Figure 4-2: Representations 1 and 2 Environment

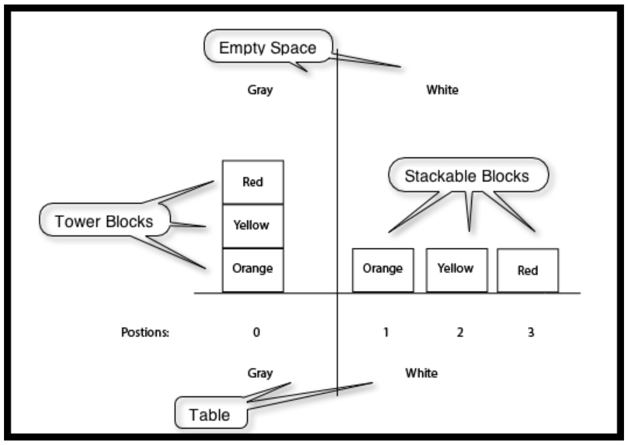


Figure 4-3: Representation 3 Environment

Figure 4-2 and Figure 4-3 show how the environment orients blocks for each representation.

The agent uses Q(λ) with a tile-coding based function approximator and a linear gradient-descent method to learn the value function. Actions are selected based on an ε-greedy strategy. Tilings are created based on derived state and action spaces. The experimenter is tasked with adjusting the configuration file and creating an input file specifying the block configuration. Decision protocol is specified in the configuration file. Different representations of experimentation are coded and run separately. Much of the code is similar. The environment, the agent's state information, and action space change from one representation of experimentation to another.

4.2 Blocks in the Environment

At the beginning of an experiment, a 'run', blocks are created and positioned in the environment based on an input file that specifies the blocks and their properties. Based on the block specification and representation of experimentation and decision protocol, the state and action spaces are derived. Based on parameters specified in the input file, the agent creates a tile set. Based on the tile set and whether the agent is using previously stored weights, the agent initializes the thetas $\vec{\Theta}_{t,\cdot}$, a concept described in Section 3.8. The agent is using Q(λ) for learning. After the agent and environment are initialized, the required number of runs, episodes, and steps are performed. At the beginning of each episode, the agent and environment are reset. The agent will return the policy learned, but the agent's rewards and position are reset. If the tower blocks are set to randomize, the environment will initialize the blocks based on that. At the end of each episode, the agent's total reward is appended to an output file. The output file is used for analyzing the agent's learning performance.

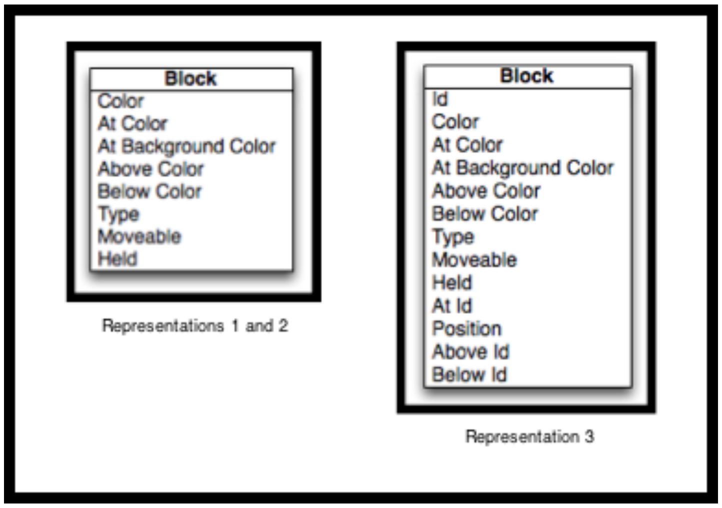


Figure 4-4: Block Properties

As show in Figure 4-4, blocks have properties. Blocks in Representations 1 and 2 have a unique 'At Color' and 'At Background Color' mapping. In all representations, to prevent the agent from confusing stacked blocks its trying to copy and the other blocks, the agent is not allowed to move a block to different background color. Notice in Figure 4-4 that in Representation 3 that a Block has an 'Id', 'Above Id', and 'Below Id'. In Representation 3, there may be duplicate block colors for the agent to copy. The Environment in Representation 3 must internally represent blocks differently in order to distinguish specifically which blocks are next to each other.

4.3 Actions

Physical actions are actions the agent may perform to affect the environment and the agent's position in it. Such actions include moving, picking up a block, dropping a block, and depending on the representation of experimentation takes perception actions (top, bottom). The success or failure of physical actions depends on where the agent is relative to the blocks, the positions and properties of the blocks, and if the agent is holding a block or not. The success or failure of physical actions does depend on what the agent has in memory. Albeit the agent may not be able to learn a policy if the agent doesn't have enough memory (state information). This does not affect results of individual physical actions.

Memory actions only allow the agent to move what's in perceptual memory to an available auxiliary memory block. If the representation of experimentation is 3, the agent may at the same time adjust the counter in the same auxiliary memory block being updated.

4.4 The Agent

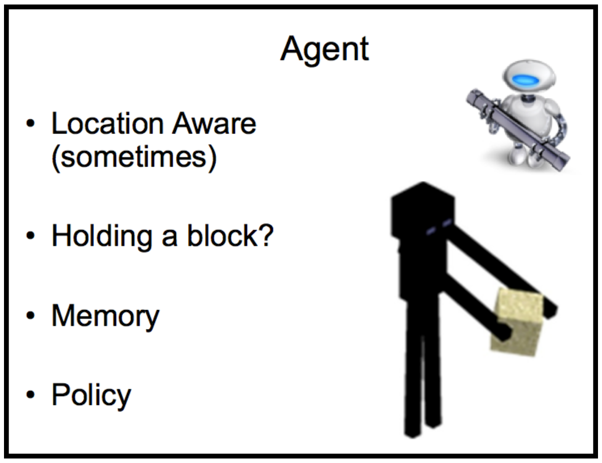


Figure 4-5: Agent Overview

The agent is in charge of keeping track of policy, memory blocks, whether it is holding a block or not, and possibly its location. The agent is only location aware in Representation 1. 'Location aware' in this context means that the agent's internal state contains location information separately from memory blocks. In Representation 1, the agent is aware of what foreground and background colors the agent is at. In Representations 2 and 3, the agent's internal state information does not contain location information separately from memory blocks, and the agent must use what's in memory to determine where the agent is located. In all representations, the agent is always aware of whether it is holding a block or not. The agent, however, does not know which block is being held. The agent must use a memory block to remember what block was picked up.

4.5 Memory Blocks

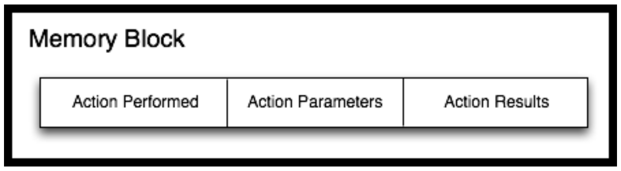


Figure 4-6: Memory Block

As shown in Figure 4-6, a memory block contains the ‘Action Performed’, ‘Action Parameters’, and ‘Action Results’. The term ‘Action Performed’ is the physical action performed by the agent in the environment such as ‘move’ or ‘pick’. In Representations 1 and 2, ‘Action Parameters’ contain parameters specified with the physical action. For example, ‘move B W’ would have ‘B’ and ‘W’ as part of ‘Action Parameters’ in a memory block. In Representation 3, ‘Action Parameters’ contain the resulting percepts the environment returns to the agent after an action. For example, if the agent takes the action ‘move_up’, the environment might return the percepts ‘R’ and ‘W’ for the agent to store ‘Action Parameters’ of a memory block. For all representations, the agent stores an action’s result(s) into the ‘Action Results’ of a memory block.

The agent will always have a perception memory block that will store the result of the agent’s last action in the environment. The agent also has a configurable number of auxiliary memory blocks. The agent uses memory actions to move what’s in the agent’s perception memory block into a designated auxiliary memory block. The agent’s total number of memory blocks is the agent’s perception memory block plus the total number of auxiliary memory blocks. The agent’s memory blocks are part of the agent’s internal state space. As the number of memory blocks increases, the agent’s internal state space grows exponentially.

4.6 Runs, Episodes, and Steps

Running experiments involves the terms of: runs, episodes, and steps. A step is one step for the agent in the environment. For example, the agent is at state A and takes an action to get to state B. The agent has made one step in the environment. A series of steps in the environment will be called an episode. For each episode, the agent's step count is reset to zero and the agent and environment are reinitialized to their respected starting conditions. For the environment, if the tower blocks are being randomized, the tower blocks will be randomized at the beginning of each episode. A 'run' is a series of episodes. The agent will learn a policy over the course of a 'run'. Multiple runs can be enabled in the experiment's configuration file, but the experimentations done in this thesis have always used one run per block configuration and memory setup. The purpose of using multiple runs is to compare policy performance. For each experiment, the maximum number of steps per episode is specified as well as the maximum number of episodes per run. The experimenter must be careful setting the maximum number of steps per episode allowed. Too few steps allowed handicaps the agent from reaching the goal.

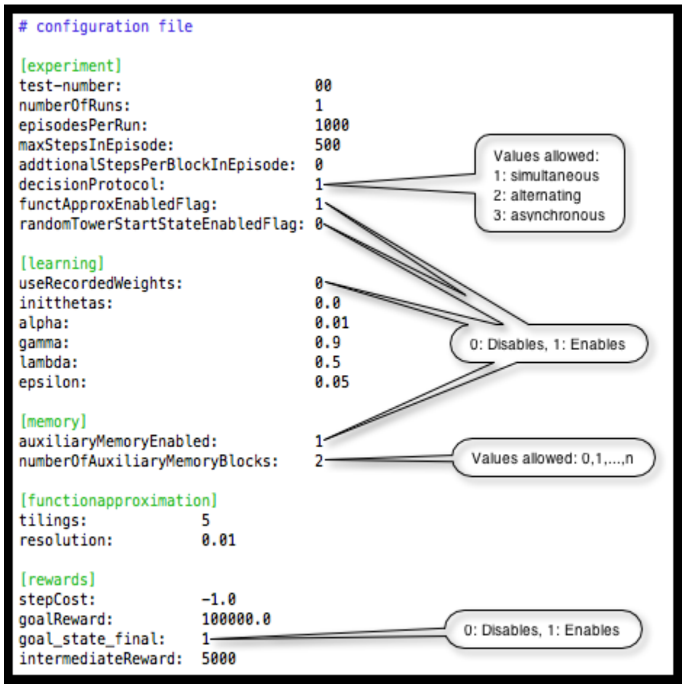
4.7 Goals and Sub-Goals

The agent is tasked with reaching the goal of the simulation. In the block-copying task, the goal of the agent is to make the unstacked blocks look like the tower of stacked blocks. Upon reaching the goal, the agent receives the goal reward. In the configuration file later described, the goal can be 'final', meaning once reached the agent immediately stops and the episode is finished. If 'final' is not enabled, the agent is expected to learn to stop after meeting the goal. If the agent were to unstack the blocks that were used to reach the goal, the agent would receive a negative goal reward. This is to prevent the agent from receiving the goal reward more than once and maintain the Markov property of the experiment. The experimentation done in this thesis always requires the goal state to be final. The agent has the opportunity to 'stop' after enough steps in an episode have been made. The agent is not allowed to stop early in an episode and is encouraged in this way to attempt to reach the goal.

For tasks that require a long sequence of steps before reaching the goal, the agent may wish to stop early. To prevent this and encourage the agent to pursue a correct sequence of steps to reach the goal, the agent is given intermediate reward at designated sub-goals. A sub-goal is a desired state of the environment that is necessary to reach the goal. Note that the state of the environment is independent of the state of the agent (what's in memory). Stacking blocks on top of one another to reach a desired stack configuration are sequential tasks that must be done in order. A sub-goal in the experimentation done in this thesis is when a set of the stackable blocks is stacked correctly. The order of sub-goals is enforced. For a goal of three blocks being stacked correctly, that the first two blocks must be stacked correctly will be the first sub-goal. The second and third blocks being stacked correctly will be the second sub-goal. Upon stacking all three blocks correctly, the agent will also receive the goal reward. If the agent undoes a sub-goal, the agent will get negative the intermediate reward. This is to prevent the agent from accumulating intermediate rewards and also to maintain the Markov Property.

4.8 Experiment Configuration

A configuration file is used in the experimentation to adjust parameters of the experiments such as decision protocol, number of auxiliary memory blocks to use, maximum number of steps in an episode allowed, etc. See the figure below.



```
# configuration file

[experiment]
test-number:                        00
numberOfRuns:                       1
episodesPerRun:                     1000
maxStepsInEpisode:                  500
addtionalStepsPerBlockInEpisode:    0
decisionProtocol:                   1
functApproxEnabledFlag:             1
randomTowerStartStateEnabledFlag:   0

[learning]
useRecordedWeights:                 0
initthetas:                         0.0
alpha:                              0.01
gamma:                              0.9
lambda:                             0.5
epsilon:                            0.05

[memory]
auxiliaryMemoryEnabled:             1
numberOfAuxiliaryMemoryBlocks:      2

[functionapproximation]
tilings:             5
resolution:          0.01

[rewards]
stepCost:            -1.0
goalReward:          100000.0
goal_state_final:    1
intermediateReward:  5000
```

Figure 4-7: Sample Configuration File

Most terms and parameters are either self-explanatory or have been explained earlier in this document. The term 'test-number' is used to label output files. The term 'additionalStepsPerBlockInEpisode' is used to add to the maximum number of steps in an episode allowed based on the number of blocks in the experiment. This is useful for scripting runs of experiments because additional steps in episodes in experiments using more blocks. The term 'initthetas' is used to initialize the tilings used with function approximation. It can be likened to a term used to initialize elements in a table. The terms alpha, gamma, lambda, and epsilon are terms used in the Q(λ) algorithm, a reinforcement-learning algorithm. The term 'tilings' gives the number of tile sets to use for function approximation. The term 'resolution' is used in making tiles in the tile sets.

Chapter V

Experimentation And Results

This chapter is intended to convey results of experimentation and commentary. Strategies used in experimentation are described in chapter two, ‘Approach Taken’. Software architecture and other notes are included in chapter four, ‘Agent-Environment Architecture’. Reinforcement Learning concepts and mechanisms are described in chapter three, ‘Background’. This chapter breaks results into what happens across representations when increasing the number of blocks, using different decision protocols, and increasing the number of memory blocks. Below are the results of experimentation.

5.1 Representations

As explained in the chapter two, ‘Approach Taken’, different representations have different state and action spaces. Different representations may be inherently harder for the agent to learn an optimal policy. For example, Representations 1 and 2 require the agent to specify parameters to actions, while Representation 3 does not. Representation 1 has actions that may only succeed if the specified parameters are only one position away. For example, ‘move W W’ would only succeed if the agent is one position away from a white foreground and background color. Representation 2 may elect to move directly to a valid position even if the agent is more than one position away. In Representation 3, the agent does not specify parameters and simply has actions like ‘move_up’ or ‘move_left’ for example. These differences in representation of experimentation can require the agent to need to perform different numbers of actions to reach the goal.

See the following figure:

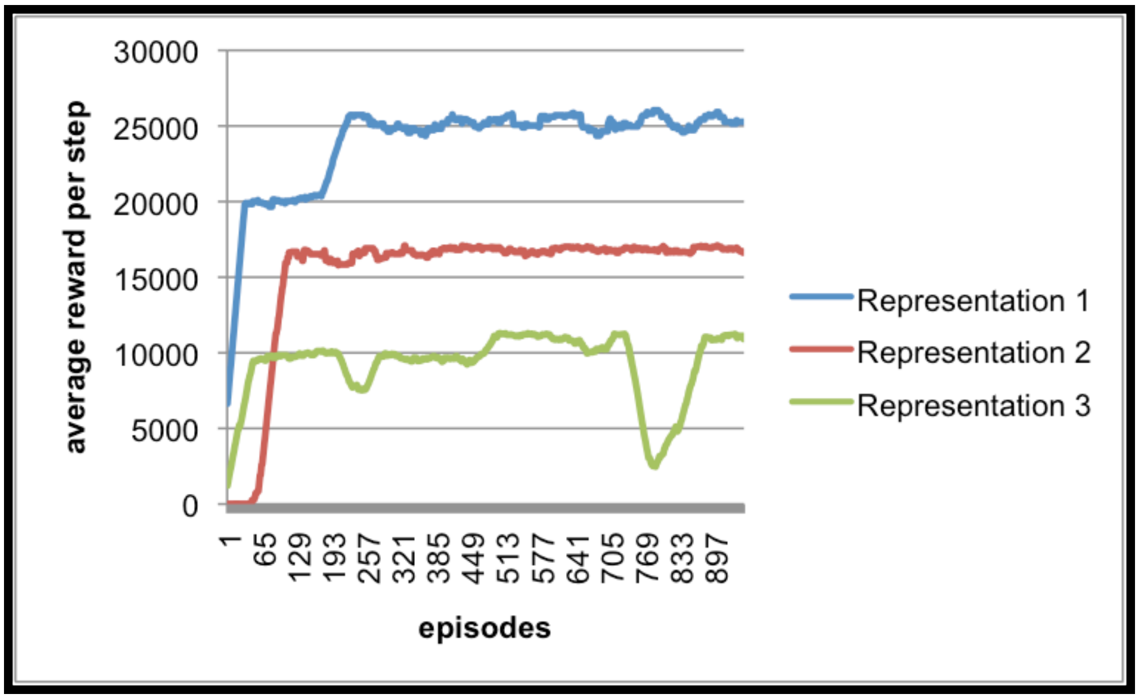


Figure 5-1: Random Tower of 2 Blocks with No Aux Memory Blocks using Asynchronous Decision Protocol

Figure 5-1 shows for a simple example, randomized tower of two blocks with no auxiliary memory and using the 'asynchronous' decision protocol that the agent learns an optimal policy for all representations of experimentation. Note that the agent converges on a different average reward per step for each representation. This phenomenon is a result of the different representations requiring different numbers of steps to reach the goal. Higher averages of reward per step are a result of the agent reaching the goal in a lower number of steps. Representation 3, for example, has more actions to perform and thus has converged to a lower average reward per step than the other representations.

As the complexity of the task increases, the difficulty level for learning is more apparent for each representation.

The following figure shows a more complicated example:

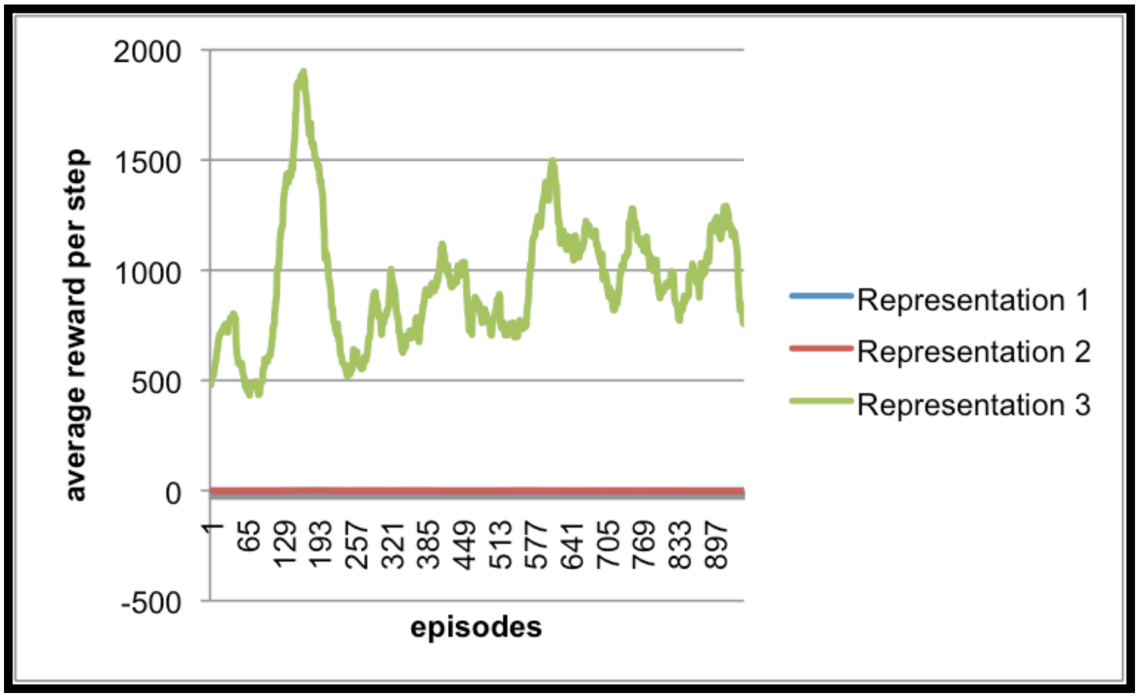


Figure 5-2: Random Tower of 3 Blocks with 1 Auxiliary Memory Block using Simultaneous Decision Protocol

Figure 5-2 shows that Representation 3 is far easier to learn for the agent than Representations 1 and 2. Figure 5-2 is not intended to show any of the representations converge on a policy. The point of Figure 5-2 is to show that given the same number of blocks to learn to copy and the same number of auxiliary memory, the agent has an easier time learning and reaching the goal with Representation 3 than with the other representations. Representations 1 and 2 have the potential to learn the scenario of Figure 5-2 given enough episodes to learn. However as will be shown in Section 5.4, some block copying tasks are not learnable for the agent if the agent does not have enough auxiliary memory. The agent may also fail due to limitations in the function approximator and the available learning time.

5.2 Number of Blocks

For each representation, the agent is given the task of copying a specified number of blocks. The number of blocks affects the state and action space complexities. As the number of blocks increases, the state space the agent must learn increases exponentially. As the state space increases, the function approximator will have a more and more difficult job representing state space and the value function.

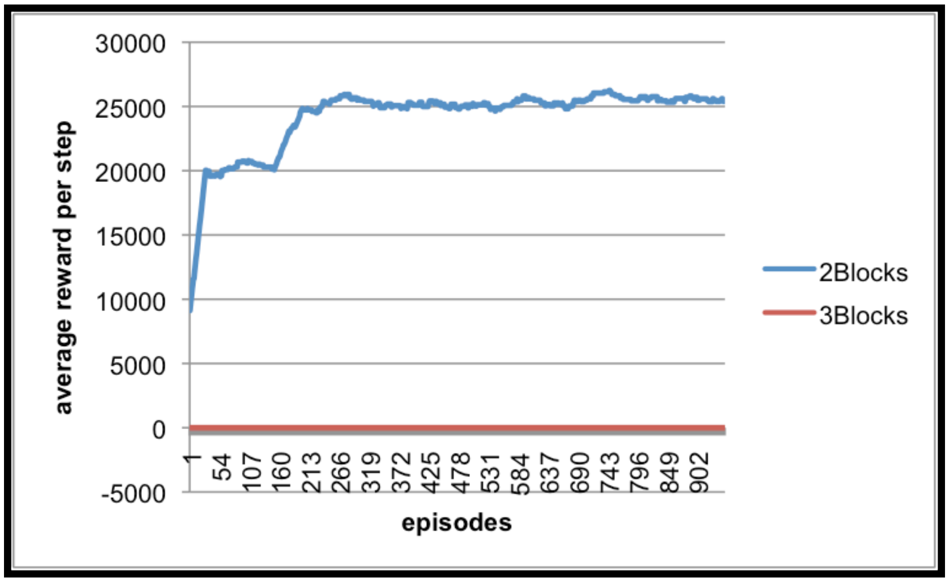


Figure 5-3: Representation 1 Random Tower with 1 Auxiliary Memory Block using Simultaneous Decision Protocol

Figure 5-3, shows that one auxiliary memory block is sufficient to learn two blocks for Representation 1, but one auxiliary memory block is not enough to learn three blocks for Representation 1.

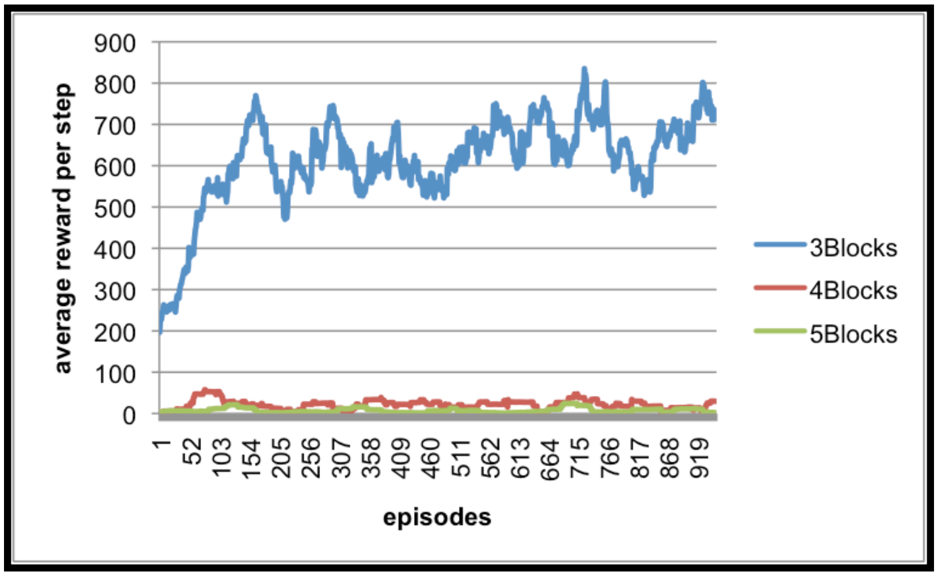


Figure 5-4: Representation 3 Random Tower with 4 Auxiliary Memory Blocks using Simultaneous Decision Protocol

Figure 5-4 is a more complicated example. Figure 5-4 shows that in Representation 3 the agent is able to start learning to copy three blocks with four auxiliary memory blocks. Four and five blocks are exponentially more complex for the agent to learn.

5.3 Decision Protocols

Decision protocols affect the size the agent's action space and can affect how well the agent learns a given task. The decision protocol 'simultaneous' enforces that the agent makes a physical action, 'move' or 'pick' for example, and a memory action at the same time. 'Simultaneous' has a larger action space but has the advantage of directly associating what actions to remember and when. The 'asynchronous' protocol lets the agent take either a physical action or a memory action at any time. 'Asynchronous' has a smaller action space but must make an extra action (a memory action) for each physical action the agent wishes to remember.

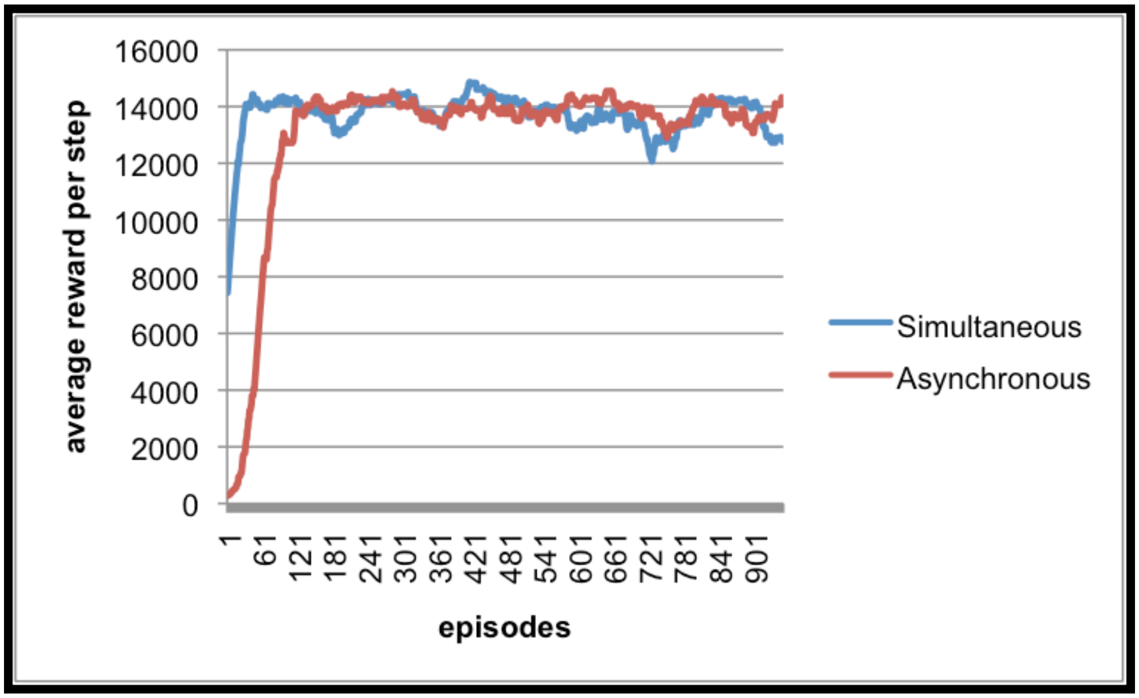


Figure 5-5: Representation 3 Non-Random Tower of 2 Blocks with 1 Auxiliary Memory Block

Figure 5-5 shows that both ‘simultaneous’ and ‘asynchronous’ decision protocols are able to converge on an optimal policy in the given simple scenario of copying two non-randomized blocks with one auxiliary memory block in Representation 3. The ‘simultaneous’ decision protocol converges slightly faster than the ‘asynchronous’ decision protocol in Figure 5-5.

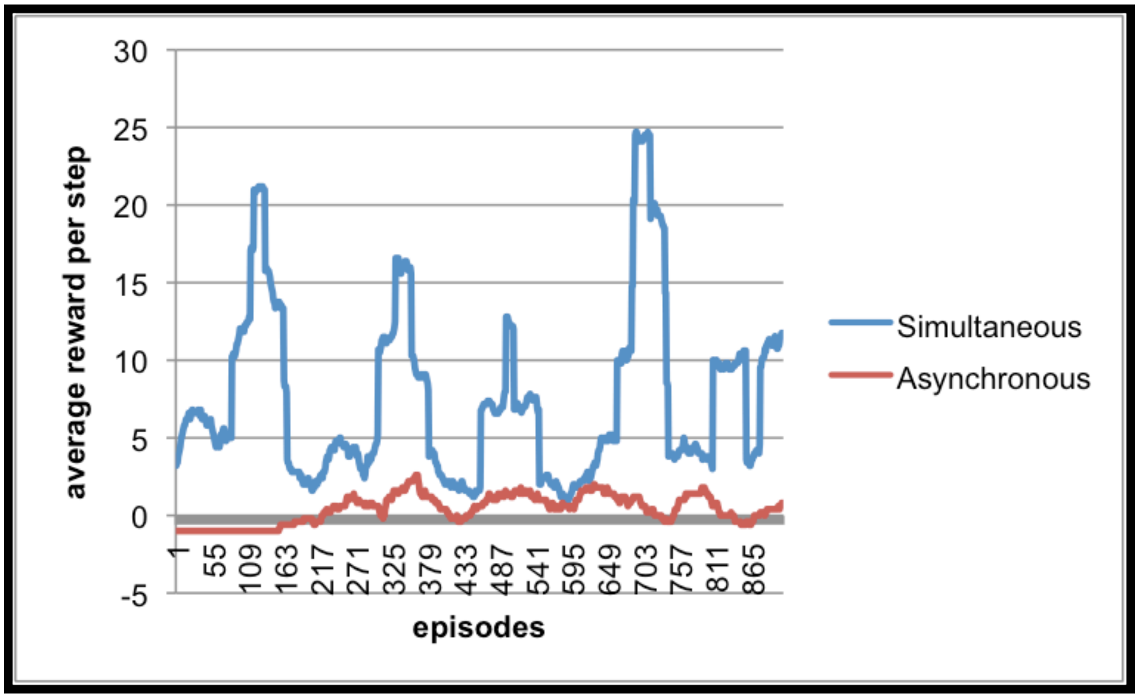


Figure 5-6: Representation 3 Random Tower of 5 Blocks with 4 Auxiliary Memory Blocks

Figure 5-6 clearly shows that the 'simultaneous' decision protocol performs better than the 'asynchronous' decision protocol for high numbers of blocks and auxiliary memory blocks. Figure 5-6 does not show convergence and is not intended to. Figure 5-6 is intended to show the general difference in learning performance between different decision protocols. The comparison between 'simultaneous' and 'asynchronous' decision protocols is a little unfair, because the 'asynchronous' protocol has more actions to perform. If a low maximum number of steps per episodes is enforced, the 'simultaneous' decision protocol may be able to accomplish tasks that the 'asynchronous' can not. However, even when compared using a high enough maximum number of steps per episode, the 'simultaneous' decision protocol performs better than the 'asynchronous' decision protocol.

5.4 Number of Auxiliary Memory Blocks

The number auxiliary memory blocks affect what block copying tasks the agent is capable of learning. The number of auxiliary memory blocks also affects the learning rate for the agent. If the agent has more auxiliary memory blocks than required to learn a task, the agent by definition is capable of learning the task but at a slower learning rate.

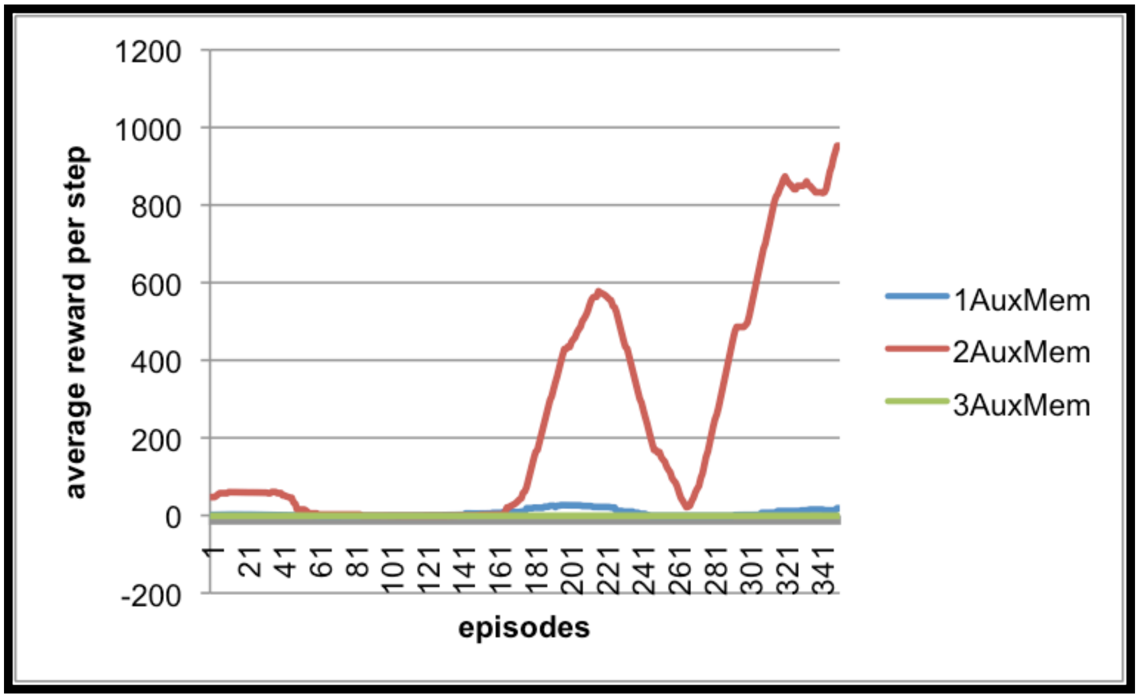


Figure 5-7: Representation 1 Non-Random Tower of 3 Blocks using Asynchronous Decision Protocol

Figure 5-7 shows the agent is able to start learning to copy three non-randomized blocks using two auxiliary memory blocks. However, the agent is unable to learn the task with one auxiliary memory block. Again by definition, the agent is able to learn the task given in Figure 5-7 with three auxiliary memory blocks but at a much slower rate.

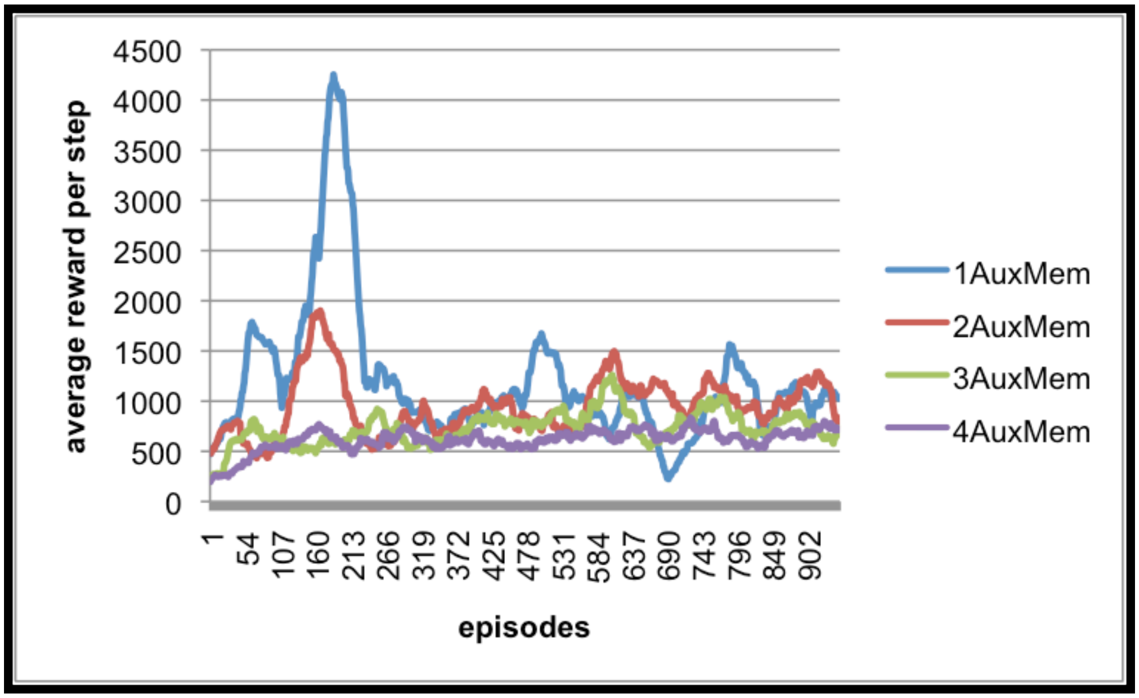


Figure 5-8: Representation 3 Random Tower of 3 Blocks using Simultaneous Decision Protocol

Figure 5-8 shows different numbers of auxiliary memory blocks used to start learning a block copying task of three randomized tower blocks in Representation 3. Figure 5-8 shows how the learning rate changes based the number of auxiliary memory blocks. The higher spike given by '1AuxMem' in Figure 5-8 can be explained by an educated guess of the agent choosing enough of the correct random actions to reach the goal more often early on but the agent may not have enough auxiliary memory (state information) to learn a policy to consistently reach that high of an average reward per step. '2AuxMem' in Figure 5-8 appears to be the most promising amount of memory for the agent to learn block copying task given in Figure 5-8. Figure 5-8 may be showing the limitations of what the function approximator used in experimentation can represent. If the function approximator is unable to representation the state space sufficiently, learning becomes unfeasible. At higher numbers of blocks and auxiliary memory blocks, average reward per step appears to oscillate even with more episodes. At low enough numbers of blocks to copy and number of auxiliary memory blocks, the agent is able to converge to an optimal policy.

Chapter VI

Conclusions and Future Work

Often one’s environment offers more information that can be processed in a reasonable amount of time. Tasks can conventionally be performed numerous ways to varying degrees of success. Memory is a limited resource and typically needs to be managed. Learning Focus of Attention is motivated by a number reasons among which are filtering of information, focusing on a task at hand and discerning what improvements can be made, and memory management.

The work done in this thesis focuses on the ‘memory management’ and ‘focusing on a task at hand’ aspects of Focus of Attention. The chosen application for Focus of Attention is block copying. The agent is given a tower of blocks to copy. There are also unstacked blocks on the floor. The agent must memorize parts of the tower of blocks it is trying to copy in order to correctly stack the unstacked blocks on the floor. The agent is forced to learn memory management because it has limited memory and must manage what to hold onto in memory. The agent is able to remember actions and whether they were successful or not. Actions include things such as movement, picking up or placing a block, and memorizing parts of the tower of blocks being copied. The agent is also learning to focus on tasks at hand by remembering what actions are important to perform and when.

The work in this thesis uses different representations of experimentation as well as different decision protocols. The agent has physical actions and memory actions. Physical actions are actions that affect the environment and the agent’s position in it. The result of each physical action is stored in the agent’s perceptual memory block. The agent has a configurable number of auxiliary memory blocks. Memory actions move what’s in the perceptual memory block into a specified auxiliary memory block. The agent’s decision protocol dictates how and when physical actions and memory actions are performed. The ‘simultaneous’ decision protocol means that the agent must decide and take a physical action and memory action at the same time. The ‘alternating’ decision protocol enforces that the agent must alternate between taking physical actions and memory actions. The ‘asynchronous’ decision protocol allows the agent to take physical actions or memory actions at any time. Different representations of experimentation model the environment and agent in different ways, and the agent has different actions depending on the representation. In Representation 1, actions have parameters and actions may only succeed if the parameters are only one position away from the agent. For example, the agent may only move one position away. Representation 2 is similar to Representation 1, but allows the agent to move directly to a valid specified location. In Representation 3, actions do not have parameters. The agent has actions such as ‘move_left’ and ‘move_right’. In Representations 1 and 2, the agent must specify parameters to actions and the environment will respond back with successful or failure. In Representation 3, the agent does not specify parameters to actions and the environment responds back with successful or failure plus what parameters the agent tried to perform an action on. For example, if the agent tries to ‘move_left’, the environment returns to the agent where the agent tried to ‘move_left’ to.

Function approximation has been used to great success in the experimentation of this thesis project. Different representations of experimentation have been implemented and tested. Representation 3 far outperforms the other representations. This is related to the fact the Representation 3 uses a significantly smaller action space by not using parameters with actions. It is reasonable for the environment to inform the agent what perceptual feedback some actions give. For example, if a human were to walk forward until he/she hits something, it is reasonable for the environment to inform the human what was walked into. The agent seems to have issues still learning to copy higher numbers of blocks. This can possibly be addressed by including position information in agent’s state but outside of memory blocks. In some representations of experimentation, the agent must use what’s in memory to determine both its position and also to learn the tower of blocks to copy. This creates a bottleneck for the agent to learn. Although it increases the complexity of the agent’s state space, the benefits could be substantial. Another explanation for agent not being able to learn higher numbers of blocks is that the function approximator used in experimentation reaches its limits in what states it can represent.

Future work includes augmenting the agent's state information using Representation-3-like strategies, using more sophisticated tiling systems, and applying the concepts used in this thesis to different tasks. Decision protocol 'alternating' has also not been fully explored. A more sophisticated expansion to this work would be trying to abstract actions by creating symbols that map to sets of actions.

References

[Sutton and Barto 1998] R. S. Sutton, A.G. Barto, “Reinforcement Learning: An Introduction”, MIT Press, 1998.

[Sutton 1996] R. S. Sutton, “Generalization in Reinforcement Learning: Successful Examples Using Spare Coarse Coding”, Advances in Neural Information Processing Systems 8, pp. 1038-1044, MIT Press, 1996.

[Russel and Norvid 1995] Stuart Russel, Peter Norvid, “Artificial Intelligence A Modern Approach”, Prentice Hall Publication, 1995.

[Rajendran and Huber 2005] Sriividhya Rajendran, Manfred Huber, “Learning Task-Specific Sensing, Control And Memory Policies”, International Journal on Artificial Intelligence Tools, Vol. 14, Nos. 1 & 2, 2005.

[Rajendran 2003] Sriividhya Rajendran, “Developing Focus of Attention Strategies Using Reinforcement Learning”, M.S. thesis, CSE, UTA, Arlington, TX, 2003.

[Fakoor 2012] Rasool Fakoor, “Reducing The Complexity Of Reinforcement Learning In POMPs By Decomposition Into Decision And Perceptual Processes”, M.S. thesis, CSE, UTA, Arlington, TX, 2012.

[McCullum 1994] R. Andrew McCullum, “Instance-Based State Identification for Reinforcement Learning”, Department of Computer Science, University of Rochester, Rochester, NY, 1994.

[Daw 2005] Nathiel Daw, “Attention-Gated Reinforcement Learning of Internal Representations for Classification”, Neural Computation, 17, 2176-2214, 2005.

[Li et al. 2007] Lihong Li, “Focus of Attention In Reinforcement Learning”, Journal of Universal Science, Vol. 13, No. 9, 2007.

[Baird 1993] Leemon C. Baird III, “Advantage Updating”, Technical Report WL-TR-93-1146, Wright-Patterson Air Force Base, OH, 1993.

Biographical Information

Stephen Ratz was born on October 23, 1984 in Fort Worth, TX, USA. He received his Bachelor of Science in Computer Engineering with Minor in Mathematics from The University of Texas at Arlington in 2009. He has worked as a Hardware Technician, Data Center Technician, Firmware Engineer, Systems Engineer, and as a Software Engineer. He began his graduate studies in Computer Engineering with focus on Intelligent Systems from The University of Texas at Arlington in Fall 2010. He received his Master of Science in Computer Engineering from The University of Texas at Arlington in Spring 2013.