Hierarchical reinforcement learning in a biologically plausible neural architecture

by

Daniel Rasmussen

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2014

Abstract

Humans and other animals have an impressive ability to quickly adapt to unfamiliar environments, with only minimal feedback. Computational models have been able to provide intriguing insight into these processes, by making connections between abstract computational theories of reinforcement learning (RL) and neurophysiological data. However, the ability of these models falls well below the level of real neural systems, thus it is clear that there are important aspects of the neural computation not being captured by our models.

In this work we explore how new developments from the computational study of RL can be expanded to the realm of neural modelling. Specifically, we examine the field of hierarchical reinforcement learning (HRL), which extends RL by dividing the RL process into a hierarchy of actions, where higher level decisions guide the choices made at lower levels. The advantages of HRL have been demonstrated from a computational perspective, but HRL has never been implemented in a neural model. Thus it is unclear whether HRL is a purely abstract theory, or whether it could help explain the RL ability of real brains.

Here we show that all the major components of HRL can be implemented in an integrated, biologically plausible neural model. The core of this system is a model of "flat" RL that implements the processing of a single layer. This includes computing action values given the current state, selecting an output action based on those values, computing a temporal difference error based on the result of that action, and using that error to update the action values. We then show how the design of this system allows multiple layers to be combined hierarchically, where inputs are received from higher layers and outputs delivered to lower layers. We also provide a detailed neuroanatomical mapping, showing how the components of the model fit within known neuroanatomical structures.

We demonstrate the performance of the model in a range of different environments, in order to emphasize the aim of understanding the brain's general, flexible reinforcement learning ability. These results show that the model compares well to previous modelling work and demonstrates improved performance as a result of its hierarchical ability. We also show that the model's output is consistent with available data on human hierarchical RL. Thus we believe that this work, as the first biologically plausible neural model of HRL, brings us closer to understanding the full range of RL processing in real neural systems.

We conclude with a discussion of the design decisions made throughout the course of this work, as well as some of the most important avenues for the model's future development. Two of the most critical of these are the incorporation of model-based reasoning and the autonomous development of hierarchical structure, both of which are important aspects of the full HRL process that are absent in the current model. We also discuss some of the predictions that arise from this model, and how they might be tested experimentally.

Acknowledgements

First, my thanks go out to my parents, who have always encouraged me to pursue the things I was passionate about, no matter their difficulty, and served as an example of that approach throughout my life. And to my brother Sean, whose hard work and ability always keep me motivated to stay one step ahead.

No one has had a more profound impact on my academic development than my supervisor, Chris Eliasmith. He opened the field of theoretical neuroscience to me, and his quality of research, knowledge, and supervision are likely to always be the standard against which I judge my own work. And my thanks of course to the other members of the CNRG, especially Xuan Choo, Travis DeWolf, and Trevor Bekolay. Working with this lab has never seemed like work, and many of the best ideas in this thesis have come out of our discussions.

And finally, I must lump a huge number of influential people into an undeservedly brief thank you, or this acknowledgement would go on forever. To my friends, family, and teachers, all of whom made it possible for me to be where I am today in different ways, thank you.

Dedication

To my wife Gill, whose light guides me in everything I do.

Table of Contents

List of Figures

Chapter 1

Introduction

One of the basic problems brains must solve is how to achieve good outcomes in unfamiliar environments. A rat trying to navigate a maze, a bird trying to decide where to forage, or a human trying to impress a new boss—all are faced with the problems of being in an unknown environment, having no clear indication of how to achieve their target, and executing a potentially lengthy sequence of decisions in order to achieve their goals.

Reinforcement learning (RL; Sutton and Barto, 1998) is a computational field that aims to address this type of problem. Specifically, RL seeks to understand how an agent can perform well when it begins with minimal knowledge of its environment and receives only sparse feedback to guide its actions. RL is also one of the best examples of cross-fertilization between computational theories and experimental investigation of the brain. RL theories have been used to provide new explanations for empirical data (e.g., Schultz, 1998), and neurophysiological data has inspired the development of new computational algorithms (e.g., Barto et al., 1983).

This goal of cross-fertilization is realized even more explicitly in computational neural modelling—the practice of building mechanistic models that recreate neural function. Applied to RL, these models can be used to explain how the abstract computations of reinforcement learning could be carried out by real brains. One way to succinctly summarize the motivation for this work is as follows:

1. Brains must solve reinforcement learning style problems somehow, as evidenced by their impressive behavioural performance
2. There are algorithms in RL that provide powerful methods for solving such problems computationally

3. If modellers can show how those methods can be implemented in neural systems, we then have a hypothesis for how the brain could achieve those same solutions[1]

That is the ideal vision of modelling, but as always there are some challenges. One of the most critical issues is the second point, with the question being "just how powerful are these algorithms?" Reinforcement learning has a 30+ year history in computer science; many different techniques have been developed, all with their own strengths and weaknesses. Thus it is quite important which computational account a modeller picks from point 2) to implement in point 3), as the resulting neural theory will have the same strengths and weaknesses (at least) as the computational theory.

Unfortunately, most modelling work has been based on some of the earliest computational theories, and we therefore know that the proposed neural system will have the same limitations as those theories. As an example, one of the most basic challenges is scaling up to complex problem spaces, where the agent must make long sequences of decisions over extended periods before achieving their goal. While even the simplest algorithms may be guaranteed to find the correct solution eventually, in practice as the required sequence of decisions becomes longer it can take impractically long time periods to find that solution. Or the problem can lie in the spatial rather than temporal dimension; for complex, information-rich environments, again many algorithms become unusable. These are problems that real brains have somehow overcome; we regularly navigate incredibly complex tasks in our daily lives, without requiring millennia of practice. Thus neural models that recreate these simple computational theories are somewhat unsatisfying, as we know that they must be missing something critical that the brain is doing.

One of the most important and pervasive features of the brain is its use of hierarchies. Abilities like vision, motor control, and analogical reasoning have all been shown to be heavily dependent on hierarchical processing (e.g., Felleman and Van Essen, 1991). The benefits of hierarchies are also appreciated on the computational side. For example, in recent years hierarchical systems have been at the forefront of computer vision (Hinton et al., 2006) and robotic control (Liu and Todorov, 2009).

Hierarchy has also made its way into the field of RL. The basic idea is to decompose the overall RL task into subtasks, whose solutions can be learned more tractably. Those subtask solutions represent abstract actions, such that if the agent executes that action it will carry out the subtask. The agent then needs to learn how to select between different abstract and primitive actions in order to complete the overall task. Effectively, hierarchies

[1]There is also a corollary outcome 3b, that understanding the successes and failures of those hypotheses can suggest improvements to the computational algorithms.

impose structure on a problem space, thereby reducing the range of possibilities that need to be explored. This helps to alleviate the challenges outlined above, allowing reinforcement learning to be applied to more complex spatial and temporal domains. Several theories have been proposed as to how to implement hierarchical reinforcement learning (HRL; see Barto and Mahadevan 2003 for a review). It is one of the most active fields in RL, and has led to interesting results in many different application domains (e.g., Morimoto and Doya, 2001; Joshi et al., 2012; Cuayáhuitl et al., 2012; Hawasly and Ramamoorthy, 2013).

This brings us to the specific motivation for the work presented here. Hierarchical reinforcement learning is one of the most powerful computational theories of RL, so is a good candidate for an algorithm to choose in point 2) above. We then seek to address point 3): can this theory be adapted so as to be implemented in a biologically plausible neural model? If so, we then have a new hypothesis for how the brain could solve the reinforcement learning challenges it faces, with all the strengths of HRL. The main contribution of this thesis is to demonstrate that it is possible to construct such a model, and explore the results of that work.

Outline

In the next chapter we go into more detail on the background for this work. We begin by presenting the underlying theories used to construct this model—reinforcement learning, hierarchical reinforcement learning, and the Neural Engineering Framework. We then review previous RL and HRL based neural modelling in Chapter 3, to lay the groundwork for the model we present here.

Chapter 4 describes the model that forms the core of this work, which can perform hierarchical reinforcement learning in a biologically plausible neural implementation. There are four main components to the model—action values, action selection, error calculation, and hierarchical composition—which are addressed in turn.

Chapter 5 contains results from the model on various tasks. These results are intended to demonstrate three things. First, that the model works as described in Chapter 4, and to demonstrate in practice the functions described there. Second, that this model has the advantages we would expect from a hierarchical RL system (e.g., it is able to take advantage of the hierarchical structure of a task to learn more quickly). Third, that this model is a plausible neural account; namely, that it is consistent with neural and behavioural experimental data from hierarchical tasks.

Finally, we conclude with a more abstract discussion of this work. This includes an analysis of the design decisions that were made when building this model, and an exploration of alternate implementations and their relative strengths and weaknesses. This leads into a discussion of how the model could be further developed in the future, in order to improve its existing performance or add entirely new functionality. We also describe some of the testable predictions that arise from this work, an important goal of any neural model.

Note that portions of this work have been previously presented in various publications, including Rasmussen and Eliasmith (2013, 2014a,b).

Chapter 2

Background

There are three key theoretical frameworks that form the basis of the model we construct. We begin with traditional reinforcement learning, as that lays out the problems and vocabulary that frame the rest of this work. Next we cover hierarchical reinforcement learning, and how it extends upon the basic principles of RL. We conclude with a discussion of the Neural Engineering Framework (Eliasmith and Anderson, 2003); this is not specific to RL, but is a general framework for constructing neural models based on mathematical/computational specifications, which we apply in this case to construct a neural model of HRL.

2.1 Reinforcement learning

2.1.1 Markov Decision Processes

The basic problem to be solved by reinforcement learning is this: given the current state of the world, what is the best action to take? Most commonly, the "world" is described formally in the language of Markov Decision Processes (MDPs; Howard, 1960). MDPs have four basic components, $\langle S, A, P, R \rangle$. S is the set of states s that the process can move through. A is the set of actions a that can be taken in each state (technically the available actions can be different in each state, but often they are treated as constant throughout the MDP). P is the transition function $P : S \times A \times S \mapsto \mathbb{R}$, which describes, given current state s and action a, the probability of reaching next state s'. R is the reward function

$R : S \times A \mapsto \mathbb{R}$, which describes the reward received if action a is taken in state s.[1] Note that the transition and reward functions do not depend on any information prior to the current state s; this is the definition of the Markov property, which gives MDPs their name. This does not mean that MDPs cannot use prior information, it just means that any such information must be included in the current state representation—the decision process itself cannot look into the past.

Deciding on a course of action can be expressed as choosing a policy $\pi : S \times A \mapsto \mathbb{R}$, which describes the probability of choosing action a in state s. Often the policies are taken to be deterministic, so only one action a has non-zero probability for state s; in that case π is sometimes expressed as $\pi : S \mapsto A$.

2.1.2 State-action values

The total value of an action is a combination of *a)* the immediate reward received for performing that action and *b)* the future rewards expected as the agent continues on from the resulting state. In order to incorporate these two components, the value of taking action a in state s can be described by

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s'} P(s,a,s') Q^\pi(s', \pi(s')) \tag{2.1}$$

Q is called the state-action value function, $Q : S \times A \mapsto \mathbb{R}$. It represents the value of taking action a in state s.[2] The first term corresponds to the immediate reward received for picking action a, and the second term corresponds to the expected future reward (the Q value of the policy's action in the next state, scaled by the probability of reaching that state). Note that Equation 2.1 assumes a deterministic policy. For a stochastic policy we would also sum over all the possible actions, weighted by the probability of selecting that action. γ is a discounting factor, which is used to prevent the expected values from going to infinity (since the agent will be continuously accumulating more reward).[3] In general, Equation 2.1 is known as a Bellman equation (Bellman, 1957).

[1] For the sake of brevity we will tend to refer to the output of R as reward, i.e., positive feedback. But the output of R can also be negative—it represents a generic feedback signal.

[2] Sometimes the values are instead expressed only in terms of the state value, $V : S \mapsto \mathbb{R}$. This can be related to the Q values as $V(s) = \max_a Q(s,a)$.

[3] Some RL approaches use an average-reward formulation that does not require a discount (Mahadevan, 1996). However, the discounted reward formulation is more common and better studied, and so is the one we describe here. A more in-depth discussion of the average-reward approach is contained in Section 6.2.4.

Most often what is sought is an optimal policy, one that maximizes the total reward received. Another way to state this is that the optimal policy π_* maximizes the Q function:

$$Q^{\pi_*}(s,a) \geq Q^{\pi}(s,a) \ \forall \pi, s, a \tag{2.2}$$

The question is how to find such a policy. If Q^{π_*} were known, the optimal policy would be to always select the action with the highest Q value. Similarly, if π_* were known, Q^{π_*} could be computed via Equation 2.1. Clearly there is a problem of circularity—the policy depends on the Q function, and the Q function depends on the policy.

This circularity can be resolved by an iterative process known as "policy iteration" (Howard, 1960). Beginning with some initial policy, the agent can compute a Q function via Equation 2.1 (sweeping across all $S \times A$ combinations until the values converge). Based on that Q function, the agent can then compute a new policy via

$$\pi(s) = \arg\max_a Q(s,a) \tag{2.3}$$

for all states. This is referred to as a greedy policy, as the agent simply selects the highest valued action in each state. The agent can then recompute the Q function based on this new policy, and the new Q function must have higher values than the previous Q function. For example, if we have a new policy where $\pi(s) \neq \pi'(s)$, that must be because $Q^{\pi}(s, \pi'(s)) > Q^{\pi}(s, \pi(s))$. This means that the second term in Equation 2.1 must have a higher value under the new policy, meaning that the Q values in general must increase.

Recomputing the Q function means that the highest valued action may have changed, so the agent can recompute the policy via Equation 2.3. This in turn leads to a new Q function, and so on. Continuing to alternate between recomputing the Q function and the policy is guaranteed to converge to the optimal policy π_*. There are other related approaches, such as value iteration, that find an optimal policy via a similar iterative process, but we will not go into them here (see Sutton and Barto 1998 for more details).

2.1.3 Temporal difference learning

With the above techniques, determining the optimal policy seems relatively straightforward. However, this approach has two major drawbacks. First, it is very computationally expensive for large state spaces, often prohibitively so for real systems, because of the need to iterate over every $S \times A$ combination. Second, Equation 2.1 relies on the premise that R and P are known. This amounts to saying that it is fairly easy to decide on the best choice if we know exactly what the consequences of our actions will be ahead of time. In most real problems this is not the case, and therein lies the difficulty of decision making.

That is the problem reinforcement learning attempts to solve: how to find a good policy π when R and P are not given explicitly. We assume that what is known is the state, s, and an action a is chosen; the agent can then observe the new state s' and any received feedback r. The challenge is how to use that information to learn to act optimally.[4]

One approach is to try to recreate R and P. Every time that action a is taken in state s, the agent can observe r and s'. This allows the agent to update the values for $\hat{R}(s, a)$ and $\hat{P}(s, a, s')$, its estimates of R and P. As the agent continues to explore the state space it will be able to fill in more and more of $\hat{R}$ and $\hat{P}$, and as the exploration goes to infinity the values will approach the true values of R and P. Finding an optimal policy can then be accomplished in various ways, for example by substituting in the estimated values for R and P in Equation 2.1 and then solving via the same iterative method. The optimality of the resulting policy will depend on how closely the estimates $\hat{R}$ and $\hat{P}$ approach the true values. This is called model-based reinforcement learning (Sutton and Barto, 1998), as the agent is building an explicit model of the world. This is a powerful and flexible approach, but the downside is that P can be a very large and complex function, sometimes requiring an impractical amount of exploration to fill in, as well as the computational challenges of using P to calculate an optimal policy.[5]

The more common approach is model-free reinforcement learning, where instead of trying to learn R and P the agent tries to learn the state-action function Q directly. Q implicitly represents the values that can be calculated explicitly if R and P are known. The question then is how to calculate Q given observations of s, a, s', and r. First, Q can be broken down into two components, the immediate and future reward:

$$Q(s, a) = r + \gamma Q(s', a') \tag{2.4}$$

where $a' = \pi(s')$.[6] Note that this is similar to Equation 2.1, except we have replaced the explicit functions with the samples from those functions. That is, the state-action value for s and a is estimated based on the reward received plus the discounted value of the action in the next state. However, the reward may be stochastic and the $Q(s', a')$ values may be

[4]Note that there are extensions of the MDP framework that do not assume the state is directly observable, known as Partially Observable Markov Decision Processes, as well as techniques for performing RL in such environments. However, throughout this work we restrict ourselves to the fully observable MDP formulation.

[5]A more in-depth discussion of model-based reinforcement learning can be found in Section 6.2.5.

[6]For the sake of simplicity we will omit the π superscript from the Q function from now on; it should be assumed that the Q function is that of the agent's policy, unless stated otherwise.

changing over time, so the Q values should be based on more than one observation. A natural solution is to calculate $Q(s,a)$ after every decision, as in Equation 2.4, and then take a running average over successive calculations. This results in the update formula

$$Q_{k+1}(s,a) = (1-\alpha_k)Q_k(s,a) + \alpha_k\left[r + \gamma Q_k(s',a')\right] \tag{2.5}$$

where α_k is the learning rate.[7] Exploring the state space will result in an increasingly accurate estimate of Q under the given policy. Similar to the policy iteration algorithm, the agent can alternate these Q function updates with updates to the policy, based on the new Q values. This algorithm is known as SARSA (Rummery and Niranjan, 1994).

As in the policy iteration algorithm, the optimal policy with respect to a given Q function is the simple greedy policy of Equation 2.3. If we substitute that in for the policy in Equation 2.5, we can observe the following:

$$\begin{aligned} Q_{k+1}(s,a) &= (1-\alpha_k)Q_k(s,a) + \alpha_k\left[r + \gamma Q_k(s',\pi(s'))\right] \\ &= (1-\alpha_k)Q_k(s,a) + \alpha_k\left[r + \gamma Q_k(s', \arg\max_{a'} Q_k(s',a')\right] \\ &= (1-\alpha_k)Q_k(s,a) + \alpha_k\left[r + \gamma \max_{a'} Q_k(s',a')\right] \end{aligned} \tag{2.6}$$

This is the Q-learning update of Watkins and Dayan (1992). The interesting thing about this update is that it allows the agent to learn the optimal Q function without reference to the agent's policy. This is known as "off-policy" learning. Essentially the algorithm assumes that the agent will always follow the optimal policy in future states. SARSA removes this assumption, instead waiting to observe what action the agent actually takes ("on-policy" learning). If the agent is acting optimally with respect to the Q values (i.e., following the greedy policy of Equation 2.3) then Q-learning and SARSA will produce identical results.

An important advantage of SARSA is that it can represent the action values for an agent that is not acting "optimally" (e.g., the agent may be following an exploration policy). For example, imagine an agent walking along a cliff. It usually picks the optimal action, but 10% of the time it picks a random action in order to explore its environment. Q learning will assign equal value to states right on the cliff edge and states farther from the edge; the optimal action is to not walk off the cliff, so the value of a state at the cliff edge is not affected by the existence of the severely non-optimal actions. SARSA, on the other hand,

[7] If α_k is set to $1/n$, where n is the number of observations at $Q(s,a)$, then this formula will calculate an exact average. But more often α_k is set to a constant or used as a "temperature" parameter that decreases over time to encourage convergence.

will notice that when the agent is in cliff-edge states sometimes it randomly wanders off the cliff. It will therefore assign a lower value to actions that bring the agent close to the cliff edge. A policy based on those Q values would therefore avoid the dangerous states, whereas one based on the "optimal" Q function would not.

SARSA also removes the operation of searching ahead over future actions. That is, in order to compute the Q learning update a system needs to evaluate the value of all the available actions in state s' and then compute the max. To compute the SARSA update the system just needs to observe the selected action and then retrieve the value of that action, $Q(s', a')$. In a computational system either of these updates is relatively easy to compute, so this is not an important concern. But when building a neural implementation, simplifying the computations involved is often an important design goal.

The downside of SARSA is that it may not find the optimal policy. That is, the learned Q function may not satisfy Equation 2.2. For example, in the above example the true optimal solution might involve walking along the cliff edge; in theory, an agent could receive the greatest reward by walking through the risky states. However, SARSA will not find that Q function, it will find the Q function that reflects the sub-optimal behaviour of the agent.

A different way of looking at this issue is to say that the SARSA update can slow down or confuse the learning process. Consider an agent trying to learn the value of $Q(s, a)$. What the agent is trying to learn is whether a is going to take it to a high value or low value state. In SARSA, the value of that next state, $V(s')$, is being approximated via a sample $Q(s', a')$. Over time those samples will converge to the true value of $V(s')$ (if we assume that the agent's policy converges to the optimal) and therefore $Q(s, a)$ will converge to the correct value, but in the meantime a lot of the $Q(s, a)$ updates are based on incorrect estimates of $V(s')$ (any time $a' \neq \arg\max_{a'} Q(s', a')$). In contrast, Q learning always updates $Q(s, a)$ based on the true value of $V(s')$, thus making more efficient use of its visits to each state.[8]

This distinction between Q-learning and SARSA brings up the important issue of exploration. After a brief period of time in a new environment an agent will have some Q values it has learned based on its limited observations. Thus it could, in theory, calculate a policy as in Equation 2.3 and act accordingly. That policy will be optimal with respect to the agent's current Q values, but it is very likely that those Q values are wrong, as

[8]All of this is dependent on the assumption that SARSA and Q learning will eventually converge to the same Q function, meaning that the agent's policy converges to the simple greedy policy of Equation 2.3. If that is not the case then the targets of the two algorithms are different, so it cannot be said that one is more efficient than the other.

the agent has yet to try the majority of the possible actions across the state space. The problem is that simply following that "optimal" policy is unlikely to improve the Q values, as the agent will keep repeating the limited set of observations that led it to that policy in the first place. This makes it quite likely that the agent will get stuck in a local maximum, instead of finding the true optimal policy.

Exploration is the practice of selecting actions that may appear sub-optimal in order to visit new regions of the state space. The simplest approach to exploration is known as ϵ-greedy exploration. In this approach the agent selects actions according to the greedy policy in Equation 2.3, but occasionally (with probability ϵ) it selects an action completely at random instead. The advantage of this approach is its simplicity, but the disadvantage is that a uniform random action selection ignores the information the agent has already obtained about its environment. The other common approach is soft-max exploration. In this approach the agent follows the policy

$$\pi(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum\limits_{a_i \in A} e^{Q(s,a_i)/\tau}} \tag{2.7}$$

This has the effect that the probability of choosing an action is proportional to the relative Q value of that action. Thus a soft-max policy directs exploration more towards higher-valued actions, compared to the uniform ϵ-greedy exploration. τ is a parameter that controls the randomness of the exploration, similar to ϵ. A common approach is to initialize τ at a high value and then gradually decrease it over the course of learning. This has the effect that at the beginning of learning, when the Q values are very uncertain, the agent will explore broadly throughout the state space. As τ decreases the agent will begin to focus more on the high-value actions, as indicated by the Q values. As τ goes to 0, the policy in Equation 2.7 will approach the greedy policy in Equation 2.3.

Q-learning and SARSA are examples of the general class of techniques known as Temporal Difference (TD) learning (Sutton, 1988). TD learning has its roots in earlier RL techniques, such as the Rescorla-Wagner model (Rescorla and Wagner, 1972), and describes a general approach rather than a specific algorithm; however, in practice, it has become almost synonymous with Q-learning/SARSA. The origin of the name can be seen by rearranging Equation 2.5 (or Equation 2.6) into the following form:

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha_k \left[r + \gamma Q_k(s',a') - Q_k(s,a)\right] \tag{2.8}$$

Here the update is re-framed as a comparison between the predicted value for $Q_k(s,a)$ and the value based on the current observation $(r + \gamma Q_k(s',a'))$. This difference is referred to as the prediction error

$$\delta_k(s,a) = r + \gamma Q_k(s',a') - Q_k(s,a) \tag{2.9}$$

If the two terms match then the prediction error is zero, and the value of $Q(s,a)$ is unchanged. If the prediction does not match the current observation then a prediction error occurs, and future predictions are adjusted in the direction of the current observation.

Although Equations 2.5 and 2.8 are mathematically identical, they differ in how they describe the underlying mechanisms. Equation 2.5 indicates that the Q values are constantly decaying, and this decay is counteracted by a steady input signal. Equation 2.8 indicates that the Q values are relatively constant, and updates are only delivered when a prediction error occurs. If data transmission has a cost, as it often does in real systems, then the latter approach is more efficient. Important for those interested in neural modelling, dopamine neurons appear to be transmitting error signals that align with Equation 2.8 (Schultz, 1998), suggesting that this approach better describes the mechanisms employed in the brain.

2.1.4 Actor-critic architecture

Actor-critic architectures make up another important sub-category of TD learning (Barto et al., 1983). We do not use actor-critic in this work, but it is widely used in neural modelling and so we include a discussion of it here. The basic idea of actor-critic is to decouple the action values (the "critic") from the policy (the "actor"). In Q-learning/SARSA the policy is based directly on the action values; if the values are known, then so is the policy (e.g., see Equations 2.3 and 2.7). In actor-critic the agent learns two separate functions—one mapping from states to values ($C : S \mapsto \mathbb{R}$), and the other mapping from states to actions ($K : S \times A \mapsto \mathbb{R}$).

It is still necessary that the state values and the policy be connected in some way, since we want the policy to reach high value states. In the actor-critic architecture this is accomplished through the learning process—the output of the critic is used to update both the actor and the critic. Since the critic output represents state values, it can be used to compute a prediction error in an analogous way to Equation 2.9 (by comparing the value of successive states). This prediction error can then be used to update the state values:

$$\Delta C(s) = \alpha[r + \gamma C(s') - C(s)] \tag{2.10}$$

The insight of actor-critic is that the same prediction error can be used to update the actor function:

$$\Delta K(s,a) = \Delta C(s) = \alpha[r + \gamma C(s') - C(s)] \tag{2.11}$$

where a is the action that led to s'. That is, the probability of selecting an action should be increased if there was positive prediction error after selecting that action, and decreased for negative prediction error.[9]

Note that if we think of $C(s)$ as an approximation of the true state value $V(s) = \max_a Q(s,a)$, then Equation 2.11 is essentially equivalent to the Q value update (Equation 2.6). In practice it is more similar to SARSA, as $\Delta C(s)$ is calculated based on the states that the agent actually visits (i.e., it is not necessarily the case that $a = \arg\max_a Q(s,a)$). Regardless, it can be seen that the end result of the actor-critic learning will be essentially the same as the other TD algorithms—a function (named K instead of Q) that indicates how preferable the different actions are in a given state.

One important reason one might choose the actor-critic architecture is that separating the K function from the error calculation gives more freedom to the form of the policy function. K can take on whatever form the modeller wants, as long as $\Delta C(s)$ can be used to update it in an appropriate way. For example, in Potjans et al. (2009) K consists of spiking neurons where the first neuron to spike determines the policy, and in Vasilaki et al. (2009) it consists of a ring of interconnected neurons used to indicate direction in 2D space. In contrast, the Q function has to output a value that can be used to compute the TD error, as well as driving the policy; this makes it less flexible than K. Actor-critic also allows the modeller to use different parameters in the two functions, such as different learning rates.

The main downside of the actor-critic architecture is that two recursively connected functions need to be learned, instead of just one. This complicates the learning process. For this reason, methods that explicitly learn Q values (such as Q learning and SARSA) have been more widely used in later RL work (although actor-critic has remained popular in neural modelling and robotics). A more in-depth discussion of the strengths and weaknesses of actor-critic with respect to the work we present here is contained in Section 6.1.

2.1.5 Continuous domains

All the theory up until now has been described in terms of discrete state, time, and action. That is the simplest case, and so tends to be how new developments are initially formulated.

[9]The output of the actor is not necessarily probabilities. Different actor-critic implementations have different ways of interpreting the output of the actor, but the general idea is as described here; positive prediction errors should increase the likelihood of selecting the action that led to that prediction error, and vice versa.

However, all of these techniques can be extended to continuous domains. We discuss work aimed at continuous space, continuous time, and continuous action in turn.

The simplest technique is to take the continuous spaces, discretize them into some number of bins, and then apply the standard discrete techniques. However, this quickly runs into problems of scale. In order to achieve good performance, a fairly fine-grained discretization is usually required; however, as mentioned previously, large state spaces tend to be the Achilles heel of RL algorithms. Discretizing continuous spaces very quickly results in state spaces too large to be feasibly tackled by discrete algorithms. Therefore it is necessary to use RL techniques that work directly in the continuous domain.

The most common approach to working in continuous space is the use of function approximation. For example, when learning Q values the agent is trying to learn a mapping $Q : S \times A \mapsto \mathbb{R}$. In the discrete case this can be represented with a lookup table, with entries for every $S \times A$ that give the value in $\mathbb{R}$. In the continuous case it is impossible to enumerate all the states in a lookup table. Instead, the agent needs to somehow represent the actual function Q, so that it can map arbitrary states to values.

A common way to accomplish this is by breaking the function space up into some number of basis functions. The Q function can then be represented by some linear combination of those basis functions. That is, we can approximate the function $f(x)$ via

$$f(x) \approx \hat{f}(x) = \sum_i w_i \hat{f}_i(x) \tag{2.12}$$

where $\hat{f}_i(x)$ are the basis functions, with the same domain and range as f. If the basis functions are fixed and the free parameters are the weights w_i on those basis functions, this is known as linear function approximation. If the free parameters include parameters of the basis functions, i.e.,

$$\hat{f}(x) = \sum_i w_i \hat{f}_i(x, p_i) \tag{2.13}$$

this is known as nonlinear function approximation. In this work we will focus on linear function approximation.

There are various different approaches to adapting TD learning to operate with function approximation (Busoniu et al., 2010). In this case the function we are approximating is the state-action value function ($f(x) = Q(s, a)$). The simplest, and most common, approach is to take the TD error $\delta_k(s, a)$, as calculated by Equation 2.9, and multiply it by the output of the basis functions to calculate a change in w_i:

$$\Delta w_i = \alpha_k \delta_k(s, a) \hat{f}_i(x) \tag{2.14}$$

This is equivalent to gradient descent over the weights with respect to the prediction error. Intuitively, this means that if for example the prediction error is positive (i.e., $\hat{f}(x)$ was too low) then the weight is increased for basis functions with positive output and decreased for basis functions with negative output, which should push the output of Equation 2.12 upwards.

Tsitsiklis and Van Roy (1997) proved the formal convergence of on-policy TD learning using linear function approximation with this type of update. Off-policy learning is more problematic, due to the disconnect between the target function and the learning samples. However, although there is no general proof, certain categories of off-policy TD learning algorithms have been shown to have guaranteed convergence with linear function approximation (e.g., Precup et al., 2001). TD learning with nonlinear function approximation lacks general convergence guarantees; however, there have been some attempts to explore this possibility, which seem to work in practice (e.g., Menache et al., 2005). It is also important to note that these convergence guarantees refer to learning a value function under a fixed policy. The full RL problem, involving learning a value function, modifying a policy, exploration, and so on, is more complex. But what these results establish is that it is not unreasonable to use function approximation to extend reinforcement learning to continuous space.

The main challenge when working in continuous time is that the learning updates are no longer occurring over fixed steps. In other words, the algorithm needs to consider the actual time that has passed between Q_k and Q_{k+1}. This can be described as a semi-Markov decision process (SMDP; Howard, 1971); an MDP that includes time as a component of the transition and reward functions. Bradtke and Duff (1995) were the first to show how TD learning could be extended to work with SMDPs. We will go into more detail on SMDPs in Section 2.2.1, as they form the basis of hierarchical reinforcement learning.

Doya (2000) took a different approach to continuous time. Instead of trying to adapt TD learning to work over extended time intervals, he removed the time interval completely. Doya re-formulated the TD error in terms of the instantaneous derivative of the value function. Intuitively, the derivative is a result of the discount factor and current reward. Comparing those values to the actual derivative gives a natural analogue of the prediction error from Equation 2.9:

$$\delta(s(t), a(t)) = \frac{1}{\gamma} Q(s(t), a(t)) - r(t) - \dot{Q}(s(t), a(t)) \tag{2.15}$$

The Bradtke and Duff (1995) approach allows us to calculate the TD error at discrete points in continuous time, whereas the Doya (2000) approach gives us a continuous TD error signal. Which one is useful will depend on the application; the former is useful for tasks with decision points separated by time delays, the latter for tasks with continuous control/feedback. In this work we will be primarily interested in the former type of problem.

The use of continuous action spaces is more rare in reinforcement learning. In many tasks it is not clear what a continuous action space would be. For example, imagine an agent whose task is to decide which groceries to purchase; what action lies in between buying apples and buying bananas? In other words, in many cases in reinforcement learning we are interested in choosing between n relatively independent options, and choosing anything other than exactly those n options may not be possible.

Tasks where continuous action makes sense tend to involve spatial control tasks. In those cases the structure of the environment gives a natural structure to the action space; for example, movement in 1D/2D/3D space has an intuitive continuous action space corresponding to the continuous state space. In these cases the solution is typically similar to the continuous state representation—function approximation. The agent's action space is divided up into a set of basis functions, and the agent's output then represents a weighting over those basis functions. For example, an agent navigating 2D space could have basis actions representing movement in the four cardinal directions. The agent would then output a 4-dimensional vector, which would be used to calculate an overall movement direction via

$$\pi(s) = \sum_i a_i \hat{f}_i(s) \tag{2.16}$$

Examples of this approach can be found in Millán et al. (2002) or Strösslin and Gerstner (2003). The work of Doya (2000) also uses continuous actions, but in 1D space. Often in the 1D case the basis function is omitted and the agent's output "weighting" is used directly as the output action.

In summary, the various components of reinforcement learning can all be extended to operate in continuous state, time, and action. However, one feature that becomes apparent in this review is that continuous problems are less explored, and the practical implementation of reinforcement learning in continuous domains still has many open questions.

2.1.6 Complexity

In the previous sections we have seen how reinforcement learning can be used to calculate an optimal policy, one which maximizes the agent's long-term reward. However, as mentioned, in practice these basic approaches have serious difficulty scaling up to complex, real-world tasks. One way to highlight this challenge is to explore the algorithmic complexity of

TD learning. The time complexity of naive Q learning (the number of steps required to find an optimal policy) is $O(|A|^{|S|})$ (Whitehead, 1991). Under some fairly minimal assumptions about the setup of the Q learning algorithm, this can be improved to $O(|A||S|)$ (Koenig and Simmons, 1993). However, the size of the state space $|S|$ is exponential in the dimensionality of the state space (this is known as the "curse of dimensionality"), thus even the more generous analysis results in exponential complexity as the dimensionality of the state space increases.

Exploring the basis for these complexities can give an intuition for the difficulties of reinforcement learning. Imagine an agent placed into an unknown environment at state s_0. All of its initial Q values are 0, thus all it can do is move randomly. As it moves around its environment, all of the terms in Equation 2.8 are 0, thus it has no prediction error and no change in its Q values. This continues until it randomly moves into the goal state, at which point it will have a positive prediction error, and update the Q value for the preceding state s_n. Beginning the next trial, all the Q values in its initial state are still 0, so it commences more random wandering. Suppose it follows the same random path; eventually it will move into state s_n, at which point it updates the Q value for s_{n-1}. This continues until it updates the value for s_0, at which point it has calculated the optimal Q values for that path. In reality the random path will not be the same each time; there are $|A|^n$ possible paths, each of which will take n steps to fully explore (assuming for simplicity that all the paths are the same length and there is no overlap). Thus the key factor is n, the number of steps between the initial state and the goal, which is determined by the size of the state space. This determines both how much wandering will be required to find the goal in a single trial, and how many trials it will take to propagate reward information from the goal back to the start state.

Almost all developments in RL can be framed as efforts to address this basic problem. One approach is to try to reduce the size of the state space, thereby reducing the value of n. Another approach is to perform more efficient exploration, thus reducing how much wandering is required to find the goal. A third approach is to propagate reward information more quickly through the state space, so that fewer repetitions are required. Hierarchical reinforcement learning addresses all three of these avenues for improvement.

2.2 Hierarchical reinforcement learning

The central idea of hierarchical reinforcement learning (HRL) is the notion of an abstract action. Abstract actions, rather than directly affecting the environment like the basic actions of RL, modify the internal state of the agent in order to activate different behavioural subpolicies. For example, imagine a robotic agent navigating around a house. Basic actions might include “turn left”, “turn right”, and “move forward”. An abstract action might be “go to the kitchen”. Selecting that action will activate a subpolicy designed to take the agent from wherever it currently is to the kitchen. That subpolicy could itself include abstract actions, such as “go to the doorway”, but the ultimate result will be a series of basic actions that move the agent to the kitchen.

The incorporation of abstract actions helps to address the challenges faced by RL in a number of different ways (Barto et al., 2013). Perhaps the most basic is that it shortens the length of the decision path to the goal (n in Section 2.1.6). Returning to our example, imagine an agent starting in the bedroom and trying to learn how to navigate to the refridgerator. A long sequence of basic actions will be required in order to complete the task, thus it will take many repetitions to propagate reward information from the refridgerator back to the bedroom. But suppose the agent selects the “go to the kitchen” action, and then a few basic actions to take it from the centre of the kitchen to the refridgerator. Reward information can then propagate directly from the kitchen to wherever the agent selected the “go to the kitchen” action. The abstract actions work like shortcuts, encapsulating whole sequences of decisions (the basic actions that actually carry out the abstract action) in a single choice. In other words, the complexity of learning the value of that abstract choice is relatively independent of the length of the actual decision path that choice will invoke.

Another important advantage of HRL is that it promotes better exploration. One of the weaknesses of RL is that learning tends to begin with a long period of random action selection, or “flailing”. This results in a kind of Brownian motion, where the agent moves around in a limited area rather than moving throughout the state space. One can imagine that if our refrigerator-seeking agent begins selecting random basic actions in the bedroom, it will spend a long time wandering around the bedroom before it gets anywhere close to the kitchen. But if the agent randomly selects the “go to the dining room” action, that will take it to a significantly different area of the state space. Thus the agent’s random exploration is going to result in a much broader coverage of the search space, and therefore is more likely to bring it within proximity of the goal.

A third advantage of HRL is that it lends itself to state abstraction. State abstraction is the process of ignoring parts of the state that are irrelevant to the current task, thus reducing the size of the state space. In HRL it is possible to associate different state abstractions with the different abstract actions. For example, suppose the agent is trying to learn a subpolicy to get to the doorway of the bedroom. In that case it does not really matter what is going on anywhere else in the house, so that subpolicy can be learned based only on the parts of the state pertaining to the bedroom. This will make it much easier to learn that subpolicy.

Note that while the previous advantages are intrinsic properties of the hierarchical framework, the question of how to come up with useful state abstractions is not trivial (e.g., how does the agent know which aspects of the state are associated with the bedroom, or which it is safe to ignore?). However, this question is more easily addressed in the hierarchical case, as the abstract actions are restricted to limited parts of the task by design. Without HRL the agent must try to find a state abstraction that works for the whole task, which is likely to be more difficult to find and also likely to eliminate a smaller portion of the state space.

The use of transfer learning in HRL is a similar case, in that it is not an intrinsic benefit of HRL but is made easier by the hierarchical framework. Transfer learning is the process of using knowledge gained in a previous task to aid performance in a new task (Taylor and Stone, 2009). While this is possible in other RL frameworks, it is made much easier by the use of HRL. One of the main challenges of transfer learning is trying to separate the knowledge that can be reused and the knowledge specific to the previous task. In HRL, knowledge is already divided into natural modular chunks—the abstract actions. The abstract actions tend to be self-contained, general, and well-defined, making them perfect components for transfer learning. For example, it is easy to see how the "go to the kitchen" action could be reused for navigating to the refrigerator, the oven, the sink, and so on. Once that subpolicy has been learned once, it can be added as an abstract action in these new tasks, thereby conferring all of the benefits described in the previous paragraphs.

2.2.1 Semi-Markov Decision Processes

The potential benefits of hierarchical reinforcement learning have long been recognized. Singh (1992) showed that when a complex task was created by composing several simpler tasks, the Q values for the overall task could be formed by a composition of the Q values on the simpler tasks. The work of Dayan and Hinton (1993) on "Feudal RL" had almost all of the components of modern HRL—policies operating at different levels of abstraction, pseudoreward, state abstraction, and more. However, it is perhaps possible to date the beginning of hierarchical reinforcement learning as a well defined and distinct field to the review work of Barto and Mahadevan (2003). Their work emphasized the fact that several recent hierarchical RL approaches were unified by the underlying theory of semi-Markov Decision Processes (SMDPs; Howard, 1971). This observation gave a common mathematical framework to the field of HRL, which was key to its continued development in the next decade.

The basic observation is that abstract actions take time to complete. When an agent selects a basic action, the result of that action can be observed in the next timestep (by the definition of MDPs). But abstract actions are not completed in a single timestep—there is some time interval that elapses while the subpolicy is executing the underlying basic actions, and only at the end of that delay period can the results of that action be observed. SMDPs extend the basic MDP framework by adding time into the various components, thus allowing them to capture this style of decision problem.

SMDPs extend the MDP framework in three ways. First, the transition function, P, must incorporate time. This is expressed as $P : S \times A \times \mathbb{I} \times S \mapsto \mathbb{R}$, where $P(s, a, t, s')$ indicates the probability of arriving in state s', t time-steps after choosing action a in state s. This is a discrete time SMDP, as time is expressed as a number of time-steps. Continuous time SMDPs also exist, where $P : S \times A \times \mathbb{R} \times S \mapsto \mathbb{R}$ (Puterman, 1994), but in general we will describe things here in the simpler discrete-time case.

Next, the reward for an action a is no longer a single value given when the action is chosen, but instead represents the total reward accumulated from when the action is chosen to when the next state is reached. In terms of subtasks, that means that a chosen subpolicy could execute several actions, some or all of which receive some reward, and the the total reward for choosing that subpolicy is the sum of those rewards.

The final change is in the discounting factor, γ. This is now applied across the time delay, so that states that are not reached for a long time are discounted more than states that are reached quickly.

Thus the state-action value from Equation 2.4, can be re-expressed as

$$Q(s, a) = \sum_{t=0}^{\tau-1} \gamma^t r_t + \gamma^\tau Q(s', a') \tag{2.17}$$

(where the transition to state s' occurs at time τ). That is, the value of selecting action a in state s is equal to the summed reward received across the delay period, plus the action value in the resulting state, all discounted across the length of the delay period. This leads to a similar adaptation of the prediction error equation to

$$\delta(s, a) = \sum_{t=0}^{\tau-1} \gamma^t r_t + \gamma^\tau Q(s', a') - Q(s, a) \tag{2.18}$$

However, with that basic framework in place there are still many different ways to implement HRL, based on issues such as how the hierarchy of actions is structured and how the prediction error is calculated and applied. Different HRL theories are defined by their choices on these issues. In the upcoming sections we explore three of the main HRL approaches in more detail.

2.2.2 Options

The "options" framework was developed by Sutton et al. (1999). The framework is based around the idea of an option, which is a generalization of actions to allow both basic and abstract actions to fit within a common framework. One of the most attractive aspects of the options approach is that this then allows many of the standard RL techniques described in Section 2.1 to extend in a relatively intuitive fashion to the hierarchical setting.

An option consists of three components, $\langle I, \pi, \beta \rangle$. $I \subseteq S$ is the set of initiation states, the states where this option is a valid choice. $\pi : S \times A \mapsto \mathbb{R}$ is the subpolicy for this option, describing which actions will be picked in each state after the agent has selected this option. Note that the actions of the subpolicy could themselves be other abstract actions. The policy is usually defined over the states in the initiation set (or, phrased differently, the option can be initiated in any state in which its policy is defined). $\beta : S \mapsto \mathbb{R}$ is the termination function, giving the probability that the option will terminate in each state. Basic actions can be thought of as a special case where $\mathcal{I}$ is the set of states where that action can be chosen, π always selects the given action, and β is always 1 (so the option always terminates after executing one action).

Acting with options proceeds as follows. If the agent x is in state s, and $s \in \mathcal{I}_o$ (the initiation set for option o), then the agent can select option o (e.g., $\pi_x(s) = o$). The next action will then be selected according to $\pi_o(s)$. This could select another option, but we will suppose that it selects a primitive action that moves the agent into a new state s'. The system now checks whether the option (probabilistically) terminates according to $\beta_o(s)$. If the option does not terminate then another action is selected according to $\pi_o(s')$, resulting in s''. Supposing $\beta(s'') = 1$ (the option terminates), control will then return to the policy that selected o, and the next action will be selected according to $\pi_x(s'')$.

Q values can be learned using essentially the same technique as in Equation 2.8, but with the updated prediction error expression of Equation 2.18. This results in the formula

$$Q_{k+1}(s,o) = Q_k(s,o) + \alpha_k \left[\sum_{t=0}^{\tau-1} \gamma^t r_t + \gamma^\tau \max_{a'} Q(s', a') - Q(s,a) \right] \tag{2.19}$$

This update is applied whenever option o terminates (in state s' at time τ).[10]

As mentioned, the concept of a policy is extended to be a policy over options rather than over actions; that is, $\pi : S \times O \mapsto \mathbb{R}$, where O is the set of available options (including those encapsulating basic actions). The policy can be determined in the same way as the standard RL approaches described in Section 2.1, e.g. through ϵ-greedy or softmax over the Q values learned in Equation 2.19.

Thus we can see that under the options framework all the standard RL processes proceed in roughly the same way. However, Sutton et al. (1999) also suggest a number of extensions to the options framework that leverage the hierarchical structure better than the standard RL approach. First, according to the description above options always continue until they terminate according to their β function. However, it is possible to re-examine the available options at each step, and if another option has a higher state value for that state than the currently executing option, then the current option can be interrupted and the second placed in control. Sutton et al. (1999) prove that the resulting value obtained will be at least as good as the original (and likely better).

Another extension is intra-option learning. Instead of updating only the state-action value function of the currently selected option (as in Equation 2.19), it is possible to update the value of any options that might have selected the same action in the same state. The observation of s, a, s', and r is not tied to the current option, so that observation can be used to update all applicable values. This can significantly speed up learning, as the system is learning different parts of the value function simultaneously.

The third extension incorporates the idea of subgoals. In the basic setup described in Equation 2.19, the agent only learns how to choose between options; no learning occurs within the options themselves (they are assumed to have fixed policies that were learned at some other time). However, if subgoals are defined for the options, then each option can learn to better achieve its subgoal, as well as the overall agent learning to choose between options. This is accomplished by adding a local reward function $g : S \mapsto \mathbb{R}$ to each option, which describes the reward the option will receive for terminating in state s (this is often referred to as pseudoreward). The state-action values for option x can then be expressed in terms of the value of the next state (if the option has not reached a subgoal) or in terms

[10]Note that the options framework uses an off-policy prediction error rather than the on-policy update of Equation 2.18, but the underlying principle is the same.

of the subgoal value (if the option has reached a subgoal):

$$Q_x(s,a) = \begin{cases} \sum_{t=0}^{\tau-1} \gamma^t r_t + \gamma^\tau \max_{a'} Q_x(s',a') & \text{if } s' \text{ is not a subgoal} \\ \sum_{t=0}^{\tau-1} \gamma^t r_t + \gamma^\tau g_x(s') & \text{if } s' \text{ is a subgoal} \end{cases} \tag{2.20}$$

Note that in this case each option maintains its own state-action value function Q_x (used to determine that option's policy). Updating Equation 2.19 to incorporate these Q values and applying it whenever an option terminates will allow the agent to improve the policy of the abstract actions, as well as learning to choose between options itself.

Because of its minimalist approach to HRL, the options framework is an attractive target for neural modelling. Thus, while incorporating many changes based on the needs of a neural implementation, the model we present here will base much of its computation on this approach (see Sections 4.4.2 and 6.1.2 for more details).

2.2.3 Hierarchical Abstract Machines

Another approach to hierarchical reinforcement learning was developed by Parr and Russell (1998), referred to as Hierarchical Abstract Machines (HAMs). The central feature of this approach is the "machine", a finite state machine that has its own internal processing, and which controls the underlying MDP as its output. Machines can be thought of as an abstracted policy; they determine which action to take in each state, but through a complicated internal process rather than a direct $S \mapsto A$ mapping. They are called hierarchical because machines can invoke other machines rather than taking an action themselves. One of the key distinctions between this and the options approach is that the goal of this abstraction is to reduce the size of the policy space, whereas options extend the policy space.

As usual, the task is represented as an MDP with states S, actions A, transition function P, and reward function R. We refer to a hierarchical machine as H_i. Each machine has its own internal states S_i and actions A_i, as well as a transition function $P_i : S_i \times A_i \times S_i \mapsto \mathbb{R}$ governing movements through that internal state. Each machine also has a function $I_i : S \mapsto S_i$ that sets the initial state of the machine when it is invoked, given the state of the MDP.

The states of a machine (S_i) have four different types. "Action" states execute an action in the underlying MDP. These states define a policy $\pi_i : S \times S_i \times A \mapsto \mathbb{R}$ that describes which action to execute given the current state s of the MDP and the current state s_i of the machine. These policies are fixed and defined as part of the machine. "Call" states invoke another machine. These states contain a function $h_i : S_i \mapsto H$ that defines which machine to invoke given the current state of the machine. "Choice" states allow the machine to make decisions by modifying its internal state. They define a policy $c_i : S \times S_i \times A_i \mapsto \mathbb{R}$. Note that these actions (A_i) are different than the MDP actions (A); A actions change the state of the underlying MDP, whereas A_i change the state of the machine. The actual next state of the machine after the action is chosen will depend on the machine's internal transition function P_i. "Halt" states stop the current machine and return control to the calling machine.

To learn to act optimally in a given environment, what needs to be determined are the c functions, the choices that the machines make concerning their internal state. Everything else in the system is fixed; the action policies and transition functions are defined when the machine is created. In other words, machines predefine certain sequences of actions based on their internal dynamics, and it is at choice nodes where a decision needs to be made about which sequence to execute.

In order to understand how to learn the c functions, it is helpful to abstract away from the internal processing of the machine, and treat the whole system as a general SMDP. The combination of a machine and underlying MDP can be thought of as an SMDP operating in an expanded state space $S \times S_i$, call it S_H. The agent wants to learn a policy $\pi_H : S_H \times A_i \mapsto \mathbb{R}$. Note that the only "actions" in this SMDP correspond to the choice states in the machine. It is a semi-MDP because after a choice is made at one of these states the system will run autonomously for some length of time, potentially accruing some reward, until it reaches the next choice state. Therein lies the advantages of the HAM approach; although the original MDP has been expanded to a more complex SMDP with a larger state space, large sections of the policy space are fixed based on the internal dynamics of the machines. The learning problem is reduced to only those parts of the state space that involve a choice state in one of the machines. Another way to look at it is that large sections of the policy of the MDP are predefined by the programmer, who uses his or her domain knowledge to focus the learning on the parts of the policy that cannot be predefined.

Thus the goal is to learn a value function $Q : S \times S_i \times A_i \mapsto \mathbb{R}$ which can be used to derive the policy for c. However, it is only necessary to learn the parts of the value function $Q(s, s_i, a)$ where s_i is a choice state. Learning these values can be accomplished using essentially the same update formula as in Equation 2.19:

$$Q_{k+1}(s, s_i, a_i) = (1-\alpha)Q_k(s, s_i, a_i) + \alpha \left[\sum_{t=0}^{\tau-1} \gamma^t r_t + \gamma^\tau \max_{a'_i} Q_k(s', s'_i, a'_i) \right] \tag{2.21}$$

This update is applied every time a choice point is reached, and τ is the time elapsed between choice points. Note that the expanded SMDP (and associated Q function) is defined only in terms of the underlying MDP and one machine. Each machine maintains its own Q function, and updates are applied to the currently executing machine. Over time the Q functions of the machines will converge to the correct values, meaning that following their policies will result in optimal reward.

The aim of the HAM framework is to provide a flexible and convenient way of specifying policies, including the ability to learn parts of the policies when their behaviour cannot be prespecified. This is realized even more explicitly in later work (e.g., Andre and Russell, 2002), where HAMs are specified using a modified version of Lisp. This is different than the goal we pursue here, where we are interested in learning all aspects of the policy, thus the model we present bears the least resemblance to the HAM approach. However, we do make use of the principle of an expanded state space that can be modified by internal actions, as a way to communicate information between abstract actions (see Section 4.4).

2.2.4 MAXQ

The third hierarchical reinforcement learning framework is MAXQ, developed by Dietterich (2000). MAXQ bears many similarities to the options framework with subgoals, but it makes subgoals the central feature of the architecture rather than an extension. The MAXQ algorithm is based on breaking a task down into subtasks, locally optimizing those subtasks, and then defining how to calculate the value function for the overall task by composing the value functions of the subtasks.

Suppose the overall MDP is broken down into subtasks M_0, M_1, $\ldots$, M_n. These subtasks are analogous to options or machines; they represent an abstract action. A subtask M_i is defined by four components, $\langle S_i, T_i, A_i, R_i \rangle$. $S_i \subset S$ defines the states in which this subtask can be executed. $T_i \subset S$ plays a similar role to β in the options framework, but in this case the subtask always terminates as soon as it reaches a state in T_i, rather than the probabilistic formulation of β. A_i is the set of available actions for the subtask, which can include primitive actions or other subtasks. $R_i : T_i \mapsto \mathbb{R}$ is the local reward function, which describes the reward received for arriving in each of the terminal states. Each subtask has its own policy $\pi_i : S_i \times A_i \mapsto \mathbb{R}$. As in the options framework, primitive actions can be defined as subtasks with $S_i = S$ (or the subset of states where a is a valid choice), $T_i = S$, $A_i = a$, and $R_i \mapsto 0$.

As with the other HRL approaches, the goal is to learn a value function that can be used to determine the optimal policy. In the option and HAM frameworks, this means learning $Q(s, a)$ values that describe the total expected reward for executing action a in state s. When options/machines learn individual value functions, each one contains all the information needed to act. The insight of the MAXQ approach is that each subtask does not need to learn the whole value function. If subtask M_i invokes subtask M_j, then the expected value for M_i is already largely defined by the value function of M_j (since M_j will be in control). M_i only needs to learn the part of the value function that does not depend on M_j.

Let Q_i be the state-action value function for M_i. To begin, the value of choosing a subtask M_j can be expressed in the same way as Equation 2.17:

$$Q_i(s, M_j) = \sum_{t=0}^{\tau-1} \gamma^t r_t + \gamma^\tau \max_{a'} Q_i(s', a') \tag{2.22}$$

In other words, the value for choosing subtask j is the reward accumulated while M_j is running, plus the expected value of the state s' in which M_j terminates. The next step is to notice that M_j also has a value function, which defines a value for $V_j(s) = \max_a Q_j(s, a)$. By definition, $V_j(s)$ describes the reward M_j expects to accumulate when starting in state s and continuing to termination. This is the same quantity being expressed by the first term in Equation 2.22. Thus M_i's value can instead be expressed as

$$Q_i(s, M_j) = V_j(s) + \gamma^\tau \max_{a'} Q_i(s', a') \tag{2.23}$$

Calculating the second term requires knowledge of the termination time and state of M_j (τ and s'). Assuming those are unknown, this can instead be expressed probabilistically as

$$Q_i(s, j) = V_j(s) + \sum_{s',t} P_i(s, M_j, t, s')\gamma^t \max_{a'} Q_i(s', a') \tag{2.24}$$

The second term represents the expected value for M_i to complete its subtask after executing subtask M_j in state s. One way to calculate this would be to explicitly model P_i; this would be a model-based approach to the MAXQ decomposition. However, MAXQ instead takes a model-free approach, summarizing the P_i and Q_i terms via a new function C

(similar to how Q was originally developed to summarize P and R). This gives a function $C_i : S_i \times A_i \mapsto \mathbb{R}$, where

$$C_i(s, M_j) = \sum_{s',t} P_i(s, M_j, t, s')\gamma^t \max_{a'} Q_i(s', a') \tag{2.25}$$

This is called the completion function for subtask M_i. These are the values that each subtask needs to learn; if it knows the C_i function, then it can calculate a state-action value according to

$$Q_i(s, M_j) = V_j(s) + C_i(s, M_j) \tag{2.26}$$

and determine a policy based on those Q values in the standard ways.

The strength of this approach becomes apparent when we consider a hierarchy of subtasks. Suppose subtask i calls subtask j which calls subtask k which executes primitive action a. The state-action value for subtask i can be decomposed into

$$\begin{aligned} Q_i(s, M_j) &= V_j(s) + C_i(s, M_j) \\ &= V_k(s) + C_j(s, M_k) + C_i(s, M_j) \\ &= V_a(s) + C_k(s, a) + C_j(s, M_k) + C_i(s, M_j) \end{aligned} \tag{2.27}$$

That is, the Q values for subtask i, involving a complex hierarchy of actions, can be broken down into a simple linear combination of lower level functions. At the lowest level of primitive actions (e.g., $V_a(s)$) the values are just based on the actual reward received for executing that action in that state. The advantage of this approach is its efficiency, avoiding the duplication of learning/knowledge among the different subtasks. The disadvantage is that it makes the actual retrieval of Q values more complex; M_i does not actually contain the Q values it needs in order to decide on a policy, instead that information is distributed throughout all of M_i's subtasks.

The final issue is how to learn the C values. At root this follows a familiar update scheme:

$$\Delta C_i(s, M_j) = \alpha[\gamma^\tau V_i(s') - C_i(s, M_j)] \tag{2.28}$$

That is, the update is based on a comparison between the predicted completion value for choosing subtask j and the actual observed value $V_i(s')$ once subtask j terminates at time τ.[11]

The challenge is how to calculate the V term. Recall that $V_i(s') = \max_k Q_i(s', M_k)$, and $Q_i(s', M_k) = V_k(s') + C_i(s', M_k)$. Thus calculating V_i involves recursively searching through

[11]The update is only applied if j terminates in a terminal state $s \in T_j$ (i.e. it is not applied if j was "interrupted" by the termination of a higher level subtask).

the subtask tree of M_i, calculating the value for each path in the tree as in Equation 2.27, and finding the maximum value path. Each path terminates at a primitive action a, where the values (e.g., $V_a(s)$) are stored explicitly. These values are updated after action a is taken and a reward is received from the environment:

$$\Delta V_a(s) = \alpha[r - V_a(s)] \tag{2.29}$$

(i.e., just encoding the average reward received).

One complication in learning the C values is the subgoal reward values, R_i. These are the local rewards a subtask receives for reaching one of its terminal states T_i. These rewards should be considered when choosing actions within subtask i, but they should not be a factor when other subtasks are judging the value of choosing subtask i (as the calling subtasks will not actually receive the local reward, only the rewards from the environment). Thus two completion functions are necessary: one to be used "outside" i (representing the true reward to be expected from M_i) and one to be used "inside" (representing true reward plus the local goals). These are termed C_i and $\hat{C}_i$, respectively. Each subtask chooses its own actions using $\hat{C}$, but it reports C when other subtasks are trying to compute a hierarchical value as in Equation 2.27. These two functions are updated simultaneously:

$$\begin{aligned}\Delta C_i(s, M_j) &= \alpha[\gamma^\tau Q_i(s', \hat{a}) - C_i(s, M_j)] \\ \Delta \hat{C}_i(s, M_j) &= \alpha[\gamma^\tau \hat{Q}_i(s', \hat{a}) + \gamma^\tau R_i(s') - \hat{C}_i(s, M_j)]\end{aligned} \tag{2.30}$$

where $\hat{Q}_i$ is calculated as in Equation 2.26, but using $\hat{C}_i$ instead of C_i. $\hat{a}$ is the best action in the next state s' according to $\hat{C}$, i.e., $\hat{a} = \arg\max_a \hat{Q}(s', a)$. Note that the updates for both C and $\hat{C}$ are based on $\hat{a}$; this is because this is the action that will actually be chosen by subtask i (since it is trying to optimize its local reward).

One of the most important distinctions between MAXQ and the other two frameworks is the optimality of the final policy. Both options and HAMs converge to a hierarchically optimal policy, meaning that, within the constraints of the hierarchy, the agent will maximize the overall reward. MAXQ converges to what Dietterich (2000) terms a "recursively optimal policy". This means that the policy π_i for each subtask is optimal with respect to the parameters of that subtask (S_i, T_i, A_i, R_i), but the overall behaviour may not be optimal. For example, suppose an agent wants to take a sip from a mug, and it has subtasks for gripping a mug and bringing its hand to its mouth. When given the "take a sip" task, options and HAM will learn the optimal way to take a sip from the mug. MAXQ will learn the optimal way to combine its "grip the mug" and "bring hand to mouth" subtasks. It will also learn the optimal way to grip a mug, and the optimal way to move its hand to its mouth. But the result may not be the best way to take a sip from the mug. For example, it may be better to grip the mug differently depending on how the hand will approach the mouth; this would require the agent to perform the "grip the mug" task sub-optimally, in order to perform the overall "take a sip" task optimally. MAXQ will not do this; in effect, it optimizes from the bottom up, so that each subtask just learns to perform its task to the best of its ability given the tools it is provided, with no awareness of the overall context within which it is operating. This is a drawback in terms of pure performance, but it can be helpful for domain transfer. For example, if MAXQ is put in a new environment where it has to pick up a mug and throw it, then it already knows the optimal way to grip a mug, whereas options/HAMs would have learned a suboptimal gripping policy that now needs to be retrained for the new task.

Although the data efficiency of the MAXQ framework is attractive, the complicated recursive search required to calculate every Q value would be very complex to implement in a neural model. Thus in our model we opt to have each abstract action maintain a complete value function. However, an important similarity between our model and the MAXQ approach is the way in which it divides the overall problem into a fixed hierarchy of subproblems that are each solved relatively independently; we adopt a very similar approach here (see Sections 4.4 and 6.1.3). Our model can also adopt either hierarchical or recursive optimality, depending on how the pseudoreward is implemented.

2.2.5 Continuous HRL

The previous discussion of options, HAMs, and MAXQ has all been in terms of discrete time, discrete states, and discrete actions. As with standard RL, this is the simpler case and so is how these new developments are initially formulated. However, compared to the discussion in Section 2.1.5 continuous HRL is still relatively unexplored.

The most formal continuous work has been undertaken in the MAXQ framework. Ghavamzadeh and Mahadevan (2001) translated the MAXQ framework into the continuous domain, showing how all the concepts discussed above could be described in a continuous fashion. However, this work is largely theoretical; we are not aware of any continuous implementations of MAXQ, other than a basic demonstration by Ghavamzadeh and Mahadevan.

There has been no such formal treatment for the options framework, but there have been several models developed that implement the options framework in a continuous environment (Konidaris and Barto, 2009; Mugan and Kuipers, 2009; Neumann et al., 2009). In general, the basic concepts of the options framework (adding options as selectable actions, defining different policies under different options, accumulating reward) are general enough that the essential theory can be applied relatively unchanged. The challenge is how to calculate the appropriate values and learning updates when given continuous input. As with standard RL, the use of function approximation provides a useful bridge. The prediction error can be calculated based on the standard options theory, and then as in Equation 2.14 this prediction error can be used to update the weights on the basis functions. This is a key feature of the Konidaris and Barto (2009) and Neumann et al. (2009) approaches. The idea behind the work of Mugan and Kuipers (2009) is to learn a discrete approximation of the state space, at which point the discrete options learning techniques can be applied.

We are not aware of any work implementing the HAM theories in a continuous environment; the finite state machine framework does not lend itself to this approach.

2.3 Neural Engineering Framework

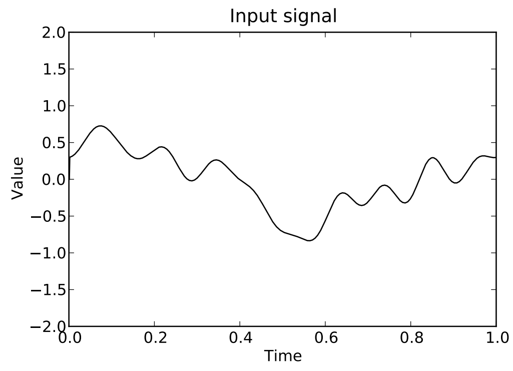
Sections 2.1 and 2.2 addressed the computational theory underlying this work; in this section we turn to the construction of neural models. The basic tool we use in this process is the Neural Engineering Framework (NEF; Eliasmith and Anderson, 2003). The NEF is a mathematical framework for taking a computational description of a system and turning it into a neural implementation. The idea is to take the mathematical variables and computations of RL, such as states and prediction errors, and recreate them using neural activities and connection weights. In this section we will discuss the NEF in a general fashion, while in Chapter 4 we will see how these techniques are applied to construct the model of hierarchical reinforcement learning.

2.3.1 Representation

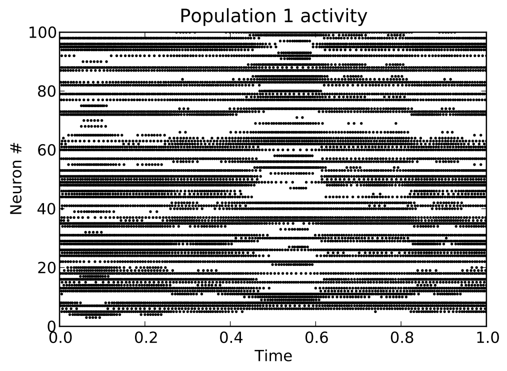
The NEF represents information in a distributed manner, using the combined information from a population of neurons to represent a value (such as the state s in reinforcement learning). There are two important components to representation. The first is encoding a value into spikes—transforming the information contained in the mathematical variable into the activity of the neurons (a to b in Fig. 2.1). The second component is decoding spikes into a value (c to d), so that it is possible for the modeller to interpret what information is being represented by the population.

Encoding a vector $x(t)$ (all computations occur over time, t) into the spike train of neuron a_i is accomplished through a neuron model

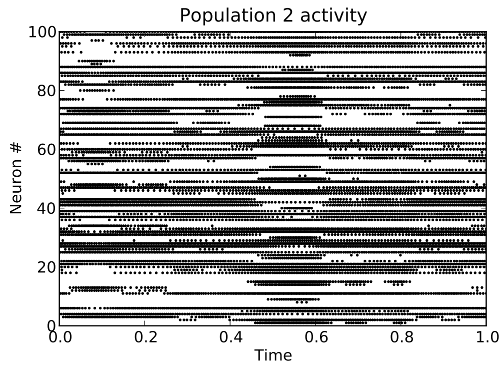
$$a_i(x(t)) = G_i\left[\alpha_i e_i x(t) + J_i^{bias}\right] \tag{2.31}$$



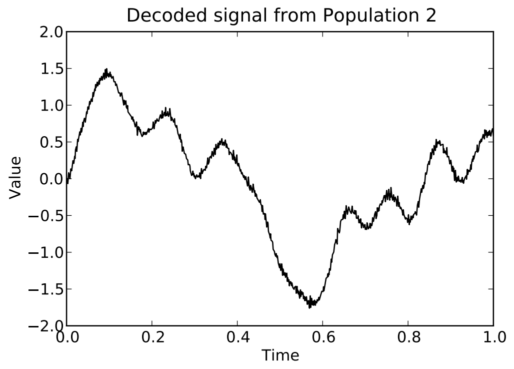
(a)



(b)



(c)



(d)

Figure 2.1: Recordings from a simple network in which an input signal (a) is fed into a population of simulated neurons (b). That population is then connected to a second population (c), with the connection weights calculated to double the represented value. Finally, the activity of the second population is decoded back into a value (d). Note: (b) and (c) are spike rasters; each row corresponds to one neuron, and each dot indicates that the neuron fired a spike at that time.

which describes the activity of neuron a_i as a function of its input current. G_i is a function representing the nonlinear neuron characteristics. It takes a current as input (the value within the brackets), and uses a model of neuron behaviour to output spikes. The variables α_i, J_i^{bias}, and e_i are the parameters of neuron a_i. The parameters α_i and J_i^{bias} do not directly play a role in the encoding of information, but rather are used to provide variability in the firing characteristics of neurons. This allows the modeller to capture the heterogeneity observed in biological neurons. The parameter e_i represents the neuron's preferred stimulus. This is an important factor in the neuron's firing, as it differentiates what properties of the input a neuron will respond to. Specifically, the dot product between e_i and the input (i.e., their similarity) drives a particular cell. In summary, the activity of neuron a_i is a result of its unique response (determined by its preferred stimulus, e_i) to the input $x(t)$, passed through a nonlinear neuron model in order to generate output activity.

In this work we use leaky integrate-and-fire neurons (LIF; Lapicque, 1907), which strike a balance between computational simplicity and realistic neural dynamics. The main component of the LIF model can be captured by a single differential equation, which describes how the neuron's membrane potential V changes as a result of the input current J (the value within the brackets in Equation 2.31):

$$\frac{\partial V}{\partial t} = \frac{1}{\tau^{RC}}(J - V) \tag{2.32}$$

τ^{RC} is a parameter of the model that describes the membrane capacitance—how quickly the membrane voltage changes as a result of the presence or absence of input current. The difference $J - V$ gives the neuron a ramping behaviour, where the voltage will increase quickly initially and then taper off as it approaches the input.

The other aspect of the LIF model is the spiking behaviour. When V passes a threshold (generally set to 1), we say that the neuron has spiked. V is then set to 0 for a period τ^{ref}, known as the refractory period, after which Equation 2.32 takes over again. Typical τ^{RC} and τ^{ref} values are 0.02 and 0.002, respectively.

One of the convenient aspects of the LIF model is that it has an analytic solution for the firing rate:

$$a_i(J) = \frac{1}{\tau^{ref} - \tau^{RC}\ln(1 - \frac{1}{J})} \tag{2.33}$$

This means that if we are not interested in individual spike timing we can avoid simulating the differential equation in Equation 2.32 and just directly calculate the firing rate. In this work we will use both the spiking and rate model in order to demonstrate that either works, but predominantly use the latter.

One important thing to note is that the formulation of Equation 2.31 makes no assumptions about the neuron model G_i. Thus the techniques of the NEF allow different neuron models to be substituted in without affecting the principles we outline here.

The second, equally important, part of neural representation is the opposite of encoding: decoding the activity of a population into a value, $\hat{x}(t)$. This is accomplished through the formula

$$\hat{x}(t) = \sum_i h(t) * a_i(x(t))d_i \tag{2.34}$$

where $*$ denotes convolution. Essentially this is modelling the unweighted current that would be induced in the post-synaptic cell by the spikes coming out of a_i. $a_i(x(t))$ is the output of Equation 2.31. The function $h(t)$ is a model of the post-synaptic current generated by each spike; convolving that with $a_i(x(t))$ gives the total current generated by the spikes from a_i. More generally, if the output of a_i is not spikes, $h(t)$ can be thought of as a filter that will be convolved with the neural activity to model the effect of the post-synaptic current.

The d_i parameters are the optimal linear decoders, which are calculated analytically so as to provide the best linear representation of the original input x. That is, we want to solve the system

$$Ad = x \tag{2.35}$$

where x is an array of points covering the possible values of $x(t)$ and A is a matrix of the form

$$\begin{bmatrix} a_0(x_0) & a_1(x_0) & \dots & a_n(x_0) \\ a_0(x_1) & a_1(x_1) & \dots & a_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ a_0(x_m) & a_1(x_m) & \dots & a_n(x_m) \end{bmatrix}$$

This is a linear least squares problem, which can be solved by various standard methods (Lawson and Hanson, 1974). The end result will be a d_i for each neuron that minimizes the squared difference between x and $\hat{x}$.

Roughly speaking, the decoders can be thought of as creating a mapping from the output current onto the input value that created that output. Since the input was transformed into current using the nonlinear neuron model, it is impossible to recreate the input perfectly using linear decoders. Thus the resulting value is an approximation, $\hat{x}(t)$, of the original input, $x(t)$. That is why we use large populations of neurons, with heterogeneous properties—so that the combined activity of all the neurons in the population counteracts the inaccuracy in any one neuron. Given enough neurons it is possible to get a very accurate recreation of the original input; for a detailed analysis of the link between population size, neuron properties, and representational accuracy, see Eliasmith and Anderson (2003).

2.3.2 Transformation

In addition to *representing* values, we also need to carry out *transformations* on those values (for example, calculating prediction error). The simplest transformation is no transformation, simply passing the value from one variable to another (e.g., calculating $y = x$). Assuming that there are two populations a and b which will represent the x and y values, respectively, then this amounts to a question of how to set the weights on the synaptic connections between a and b. Referring back to Fig. 2.1, the connection weights between the two populations—2.1b to 2.1c—are what need to be determined. The goal is to set the weights in such a way that when the neurons in population a are firing to represent x, this will cause the neurons in population b to fire in such a way that they also represent x. Note that this is not equivalent to making their firing activity identical, as the a and b neurons have different random properties and therefore different decoders.

Recall that the activity of neuron b_j is a result of its nonlinear response (G_j) to the input current. However, the input to population b is no longer a direct value, but is instead the output from population a. The output of population a is given by Equation 2.34, so to calculate the firing of population b, Equation 2.34 can be substituted in for $x(t)$ in Equation 2.31:

$$\begin{aligned} b_j(x(t)) &= G_j\left[\alpha_j e_j (\sum_i h(t) * a_i(x(t)) d_i) + J_j^{bias}\right] \\ &= G_j\left[\sum_i h(t) * a_i(x(t)) \alpha_j e_j d_i + J_j^{bias}\right] \\ &= G_j\left[\sum_i h(t) * a_i(x(t)) \omega_{ij} + J_j^{bias}\right] \qquad (2.36) \end{aligned}$$

In other words, the input current of neuron b_j is equal to the output current of all the a neurons connected to b_j, multiplied by the connection weights. In most neural simulations the connection weights ω need to be learned. The NEF formulation allows the connection weights to be analytically determined: $\omega_{ij} = \alpha_j e_j d_i$. Referring back to the descriptions of the variables in Equations 2.31 and 2.34, what this means is that the connection weight between neuron a_i and b_j is equal to the preferred stimulus of b_j multiplied by the decoders for a_i (all scaled by the gain, α_j, on b_j). Intuitively this makes sense; the output of a_i is being decoded into a value (via d_i) and then re-encoded into the input of b_j (via e_j), but the connection weights are compressing that decoding/encoding into a single step.

So far the value has not really been transformed, the output of a is simply being passed directly to b. Let us suppose we want to calculate a scaled version of the value represented in a, e.g., $y = 2x$. If the output of a is x, multiplying that output by 2 will give $2x$. That is,

$$2\hat{x}(t) = 2\sum_i h(t) * a_i(x(t))d_i \tag{2.37}$$

Substituting that into the encoding of b_j, in the same way as Equation 2.36, results in the connection weights $\omega_{ij} = 2\alpha_j e_j d_i$. In other words, if we want to double the represented value, we simply double the connection weights. More generally, if we want to multiply the vector x by the matrix C, we can calculate the weights as $\omega_{ij} = \alpha_j e_j C d_i$.

Another transformation is combining two values ($z = x + y$). This is almost identical to Equation 2.36, except that now the input current is coming from two populations instead of one:

$$c_k(x(t) + y(t)) = G_k \left[\sum_i h(t) * a_i(x(t))\omega_{ik} + \sum_j h(t) * b_j(y(t))\omega_{jk} + J_k^{bias} \right] \tag{2.38}$$

where $\omega_{ik} = \alpha_k e_k d_i$ and $\omega_{jk} = \alpha_k e_k d_j$. A second, analogous term has been added to incorporate the second input population. This can be continued in the same way to combine any number of inputs.

These two transforms can be combined to compute arbitrary transformations of the form $z = C_1 x + C_2 y$. This is simply Equation 2.38, except $\omega_{ik} = \alpha_k e_k C_1 d_i$ and $\omega_{jk} = \alpha_k e_k C_2 d_j$. Thus with these techniques any linear transformation can be computed using neural connection weights.

Notice that the linear transformation discussion has been solely about encoding from value to spikes, there has been no discussion of decoding from spikes to values. This is because the decoding is unaffected by these linear transformations. Recall that the optimal linear decoders map from output current to the input value that caused that current. These manipulations are only changing the input—the mapping remains the same. However, it is also possible to compute different mappings. Instead of calculating the decoders via $Ad = x$ (Equation 2.35), we can instead calculate them as

$$Ad = f(x) \tag{2.39}$$

where f is some arbitrary function. Using the same linear least squares solving process will then result in a set of decoders that approximate $\hat{f}(x)$ instead of $\hat{x}$. Using those decoders to calculate the connection weights as above will then result in a transformation that approximates the desired function.

This allows for the computation of nonlinear transformations. For example, one of the most common nonlinear operations is multiplication ($z = xy$). There are a number of methods to perform multiplication using neurons (Eliasmith and Anderson, 2003; Polsky et al., 2004); here we will describe a technique that uses only linear dendrites, as this is the most conservative assumption regarding dendritic computation. With this approach, multiplication involves a two stage process, using an intermediate population M. The M population takes x and y as input and combines them into a two dimensional value $m = [x\ y]$. This is a linear transformation, and so is accomplished as in Equation 2.38. Then the decoders of M, instead of mapping onto the same two dimensional space as the input, map onto a one dimensional space by setting $f(m) = m_1 \times m_2$ in Equation 2.39. Note that the decoders are still only linear weights and so can only approximate a nonlinear transformation such as multiplication. The accuracy of the approximation is proportional to the number of cells, so the multiplication can be made more or less accurate depending on how many neurons the modeller wants to use.

These techniques can also be used to calculate recurrent connections. The idea is the same as Equation 2.36, except instead of the input coming from a different population, it comes from the output of the same population. This results in the connection weights $\omega_{ij} = \alpha_j e_j d_i$ where e and d are the encoders and decoders for population a. With no other transformations applied, this will cause the population to feed its own represented value back to itself. This results in the population performing integration over its inputs; given no input it will maintain its current value, and otherwise it will add up all the inputs it receives. This technique is useful for creating simple memory components, as we will see in Chapter 4.

2.3.3 Learning

The previous section discussed how to analytically calculate the connection weights for a desired transformation. However, in some cases the required transformation is not known ahead of time; for example, in reinforcement learning the Q values are not known until the agent actually starts exploring its environment. In those cases the weights need to be learned online.

There are many learning rules that can be implemented in the NEF, but in our work we use the Prescribed Error Sensitivity rule (PES; MacNeil and Eliasmith, 2011). This is described by the formula:

$$\Delta d_i = \kappa E a_i(x) \tag{2.40}$$

where κ is the learning rate and E is some error signal to be minimized. Note the similarities between this and the basis function weight update in Equation 2.14. The neuron activation functions are the basis functions, and we multiply the basis function output by the error to calculate a change in the weight (d_i) on that basis function. The rule as expressed here is in terms of changes to the decoders; the effect of this on the actual connection weights can be observed by multiplying both sides by the encoders and gain of the neurons to which a_i connects:

$$\Delta \omega_{ij} = \kappa \alpha_j e_j E a_i(x) \tag{2.41}$$

This learning rule will cause the transformation calculated by the decoders/connection weights to be adjusted in the direction of E. For example, if the output of the a population represents Q values, and E is the TD error, this will cause the Q values to be adjusted as in Equation 2.8.

2.3.4 Basal ganglia network

Several NEF models make use of a specific model of the basal ganglia (e.g., Stewart and Eliasmith, 2011; Eliasmith et al., 2012; Stewart et al., 2012; Choo and Eliasmith, 2013; DeWolf and Eliasmith, 2013), including the work we present here. This model is based on work by Gurney et al. (2001), who employed a mixture of experimental and computational work to build a computational model of basal ganglia function. This computational model was then translated into a detailed neural model using the principles of the NEF by Stewart et al. (2010).

The basic observation of Gurney et al. (2001) was that the basal ganglia architecture is well suited to computing an “arg max” type function over its inputs. That is, given a set of n inputs, it is good at finding which input has the highest value. This makes it perfect for performing action selection; if we think of the basal ganglia inputs as action values, then the effect of the basal ganglia will be to select the highest valued action.

The central operation is off-centre on-surround connectivity. The basal ganglia operates through inhibition, thus off-centre corresponds to inhibiting—and therefore selecting—the target action. This occurs via the direct striatal–globus pallidus internus (GPi) connections. On-surround means that each input will also be disinhibiting the other actions, which occurs via connections from the striatum to GPi through the subthalamic nucleus

(STN). The net result of this will be that the highest value action is inhibited, and all the other actions are disinhibited.

However, the problem with this feedforward style of arg max calculation is that it is very sensitive to the range of its input values. If the values are too high there will not be enough disinhibition, and several actions will get selected. If the values are too low the opposite occurs, and no actions will be selected. Thus the basal ganglia model contains a secondary circuit that attempts to normalize the internal values. This follows the same off-centre on-surround pattern as above, but projecting onto GPe instead of GPi. Thus the GPe also contains the max action, but instead of being output from the basal ganglia this value is used to inhibit the STN and GPi. The STN is controlling the level of the on-surround in the previous circuit, thus by inhibiting this population (in concert with the GPi) the GPe is controlling the level of disinhibition in the main selection circuit. This allows the overall basal ganglia model to perform a more robust arg max calculation.

This network is often combined with a simple model of the thalamus in order to transform the basal ganglia output into a more usable form. The model is based upon the observation that the thalamus is constantly trying to activate different actions, but is inhibited by the basal ganglia output (Redgrave et al., 1999). This has the effect of converting the negative selection of the basal ganglia (where the selected action is inhibited and all others are disinhibited) into a positive selection (where the selected action is excited and all others are inhibited). The thalamus model also includes mutual inhibition between the different actions to further ensure that only one action is selected.

2.3.5 Nengo

The actual construction and simulation of NEF models is carried out by a software suite called Nengo (Stewart et al., 2009; Bekolay et al., 2014). Nengo is an open-source project (`http://www.nengo.ca`), and developing, maintaining, and extending Nengo has been an important part of this work.

Nengo implements the NEF mathematics and provides a high-level functional perspective to the modeller. This allows the model to be specified in terms of mathematical variables and transformations, which Nengo then translates into neural activities and connection weights. Models can be constructed via a graphical interface, but for complex models Nengo also provides a Python scripting interface. For example, the network shown in Figure 2.1 could be constructed via:

```
# construct two populations of 100 neurons, representing 1-D values
```

```
a = nengo.Ensemble(100, 1)
b = nengo.Ensemble(100, 1)

# connect a to b with a transform that doubles the represented value
nengo.Connection(a, b, transform=[[2]])
```

Nengo will then run all of the encoding, decoding, and transformation equations in order to construct the specified model. Nengo also supports various different learning rules, including the PES update used in this work (Equation 2.40).

A key goal of Nengo is to support the efficient simulation of large, complex neural models. Concurrent with the work in this thesis, Nengo has been rewritten from the ground up to better support this goal, leading to significant speed improvements (see Bekolay et al. 2014 for benchmarking results). However, the scale of these models is still limited by computational constraints. This issue is discussed in more detail in Section 6.2.7.

For more detailed examples of working with Nengo, see the documentation at `http://www.nengo.ca/documentation`. All of the code used in this work is available in the supplementary material of this thesis.

Chapter 3

Previous work

In this chapter we will review previous efforts to build neural models capable of RL and HRL. The work in this thesis is aimed specifically at HRL, but interesting comparisons can also be made to basic RL modelling and so we include a review of that work. In addition, the previous work in HRL is fairly sparse, so including RL models gives a larger body of work to compare against. We also include models with varying degrees of biological detail. Many of these models are not intended to provide insight into actual neural function, and so are not concerned with biological plausibility—they simply use neural networks as a computational tool. However, these models can still provide useful points of comparison, and so we include them in this discussion.

3.1 Reinforcement learning

In this section we address neural models of standard (non-hierarchical) reinforcement learning. After a brief comment on artificial neural network approaches, we focus on models that seek to provide insight into how real neural systems might perform RL. We begin with a discussion of associative RL models; this is the most common approach in neural RL modelling, but it omits important aspects of the full RL problem. We then explore models that go beyond associative RL, examining different approaches to addressing the challenges of reinforcement learning.

3.1.1 Artificial neural networks

One of the most well-known success stories in RL involved a neural network model. Tesauro (1992) used a neural network to represent the state of a backgammon board, and trained it with RL to predict the value of different board positions. Using the predictions of that network to guide move selection resulted in a system able to achieve expert level play. However, this is an example of work that uses neural networks for purely computational purposes, with no intention of biological plausibility. For example, the network used a very carefully crafted state representation, with specific neurons tied to specific aspects of the state (e.g. one neuron active if there is one piece on a position, a different neuron if there are two pieces, a different neuron if there are three pieces, etc.). It is unlikely that real neural systems have such carefully tuned backgammon representations. In addition, the neuron model itself was a simple sigmoidal function, rather than something that more closely matches the behaviour of actual neurons (e.g., generating spikes) such as the LIF model (Equation 2.32). Finally, Tesauro's model used a learning update that could not be computed by a local synaptic learning rule. However, these points are not intended as critiques of the work—the purpose of this model was to build a system that was good at backgammon, not to understand how humans play backgammon. The important contribution of this work from our perspective is as an early demonstration of the power that neural networks can provide, as well as the feasibility of combining neural networks with RL.

Other cases where neural networks are used for purely computational purposes tend to involve the extension of RL to continuous domains. As discussed in Section 2.1.5, function approximation becomes crucial in these problems, and neural networks are often a useful function approximation method. These networks are typically used only for approximating the value function; other aspects of the RL algorithm, such as TD error calculation, are not computed neurally. For example, the SARSA learning algorithm was originally developed as a useful tool for extending Q learning to neural network models (Rummery and Niranjan, 1994), where the SARSA update was used as the error signal to train a network via backprop to represent the value function. Neural networks are also useful for problems with continuous time and action; for example, Millán et al. (2002) used neural networks to represent a continuous action space, and Baddeley (2008) used the continuous TD error formulation of Equation 2.15 (Doya, 2000) to train a neural network. Again, the aim of these networks is purely functional—neural networks are employed here as a computational tool, not in an effort to understand biological neural systems.

The next step of biological detail involves models that attempt to connect conceptually to real neural systems, even if their actual implementations omit important neural features.

For example, Strösslin and Gerstner (2003) used a simulated neural network for the same reason as those above—in order to represent a value function over continuous state, trained by an externally computed TD error. They went further though, by connecting the neural basis functions to the receptive fields of place cells in the rat hippocampus. Even though the model itself is still quite simplified (e.g., the "neuron model" is simply a linear summation of inputs), this is a useful exercise. Tying the components of the model to components in the brain makes that mapping testable. For example, they map their action selection mechanism to the nucleus accumbens in the ventral striatum. That then requires the existence of information flow from the hippocampal value representation to the nucleus accumbens, which is a testable element of the model. If such connectivity is observed in real brains (as it is) that lends support to the algorithmic proposal of this model.

The work of Hasselmo (2005) is another example of this approach. Although many aspects of the model are simplified (e.g., binary neurons and connection weights), he ties the model conceptually to features such as cortical microcircuits and Hebbian learning. One interesting feature of the Hasselmo (2005) model is that the error signal is computed internally, through neural computation, rather than received as input (although the error signal is a simplified one customized to the structure of the model, rather than a true TD error). Hasselmo also compares the results of his model to real neural data, such as the firing patterns of neurons in medial prefrontal cortex during a rat spatial navigation task. Again, even though these comparisons are more conceptual than implementational, they give a basis to judge whether or not real neural systems could be using a similar approach to that proposed by the model.

3.1.2 Associative reinforcement learning

As the move is made to more biologically plausible models, often there is a trade-off between biological detail and functional power. Purely computational systems have the option to ignore some of the challenges faced by real physical systems, such as limited precision, capacity, and local information transfer. Thus when biologically based models add these extra constraints, it is often necessary to simplify the computations they are performing.

One simplification common to these models is that they restrict themselves to "associative reinforcement learning". In associative RL the agent does not consider the future impact of its actions (i.e., the value of the subsequent state), it just tries to pick whichever action will result in the largest immediate reward. That is, instead of representing the state-action value as

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s,a,s') \max_{a'} Q(s',a')$$

it is simply

$$Q(s,a) = R(s,a) \tag{3.1}$$

The majority of work in biological RL modelling has been on this type of learning (Seung, 2003; Florian, 2007; Baras and Meir, 2007; Izhikevich, 2007; Urbanczik and Senn, 2009; Stewart et al., 2012).[1]

The learning update in this case is simply an average over the immediate rewards:

$$Q_{k+1}(s,a) = (1-\alpha)Q_k(s,a) + \alpha r \tag{3.2}$$

Note that this is equivalent to the classic Rescorla-Wagner rule (Rescorla and Wagner, 1972). In this update it can be seen that there are no concerns about reward propagation or comparing different state values. The task is just to match the output of $Q(s,a)$ to the signal received from the environment.

The main problem to be solved in these models is how to apply the learning update in Equation 3.2 via a realistic neural learning rule. The key aspect of synaptic learning rules is that the weight update can only be based on information that is actually present at the synapse. Generally this includes the activity of the presynaptic neuron, the activity of the postsynaptic neuron, and a global reward/error signal which is broadcast across large groups of neurons via neuromodulators such as dopamine. Learning rules that violate these assumptions include rules like backpropagation, as it requires error information from neurons downstream of the postsynaptic neuron,[2] or rules that deliver a customized error signal to each neuron.

One broad class of realistic local learning rules is known as Hebbian learning. The signature of Hebbian learning is that it is based on the correlation between the presynaptic (a_i) and postsynaptic (b_j) activity:

$$\Delta\omega_{ij} = \alpha a_i b_j \tag{3.3}$$

RL models typically use reward-modulated Hebbian learning, which adds a reward term into this equation:

$$\Delta\omega_{ij} = \alpha R a_i b_j \tag{3.4}$$

[1]Often associative RL is applied to what is known as a bandit task. In bandit tasks there is no state; the actions selected by the agent do not affect the environment, they only affect the immediate reward (which the agent receives after every choice). In this case the target function to be learned can be further simplified to $Q(a) = R(a)$.

[2]There is some evidence for backpropagating action potentials (Stuart et al., 1997), but it is still unknown what information is contained in that backpropagation.

Intuitively, the presynaptic–postsynaptic correlation indicates the importance of a synapse in the current network state; if the correlation is high, then that synapse is contributing significantly to the current processing of the network. Adding the reward term indicates that if a synapse is important and the reward is positive then the synapse is probably doing something right, so its weight should be increased (and vice versa if the reward is negative). If, for example, a_i represents the state and b_j represents the Q value, this will cause the weights to be increased/decreased on the synapses connecting highly active state neurons to highly active Q neurons. The result of this will be a learning update akin to Equation 3.2. Seung (2003) and Urbanczik and Senn (2009) are examples of associative RL models using some variation of this approach.

The other learning rule commonly used in these systems is spike-timing dependent plasticity (STDP; Markram et al., 1997). This can be thought of as a time-based extension of Hebbian learning; instead of using the correlation between presynaptic and postsynaptic activity, this rule uses correlations between presynaptic and postsynaptic spike times. There are different forms of the rule, but the general idea is that if the postsynaptic neuron fires a spike within some time window (on the order of 10s of milliseconds) *after* a presynaptic spike then the weight of the synapse is increased, while if the postsynaptic neuron fires *before* the presynaptic neuron (negative correlation) the weight is decreased. As in the Hebbian case, reward can be incorporated into this update multiplicatively, so a positive correlation with positive reward leads to weight increase, positive correlation with negative reward leads to weight decrease, and so on. Florian (2007), Baras and Meir (2007), and Izhikevich (2007) use reward-modulated STDP to perform associative RL.

We also include in the class of "associative RL" models that compute the TD error outside the model and then feed it in as an input signal. For example, in Frémaux et al. (2013) the model contains a representation of the value function in spiking neurons. They then externally compute a value analogous to the continuous TD error of Doya (2000), and feed it in to an error modulated Hebbian learning rule (similar to Equation 3.4) to update the value function representation. We call this associative RL because the model just needs to pick the action in each state that will result in the highest immediate reward—the difference is that the "reward" in this case is the externally computed TD error, rather than the environmental reward. Of course the overall system does end up performing TD learning, but if we are interested specifically in neural mechanisms then this system is solving an associative RL problem.

3.1.3 PVLV and PBWM

The PVLV (Primary Value–Learned Value) model of O'Reilly et al. (2007) was one of the first attempts to go beyond associative RL in a neural model. It consists of two systems (Primary Value and Learned Value), which together are used to generate a prediction error signal that can be used to train a reinforcement learning system.

The PV system is an associative RL process, the same as those described above—it attempts to learn a mapping from the current state to immediate reward:

$$PV(s) = R(s, a) \tag{3.5}$$

Note that O'Reilly et al. are not concerned with action selection in this model, they are only concerned with learning the value function in order to generate the error signal. Thus they take the agent to be moving through a fixed sequence of states. In that case it is more natural to associate the reward directly with the state, i.e. $PV(s) = R(s)$, which is how O'Reilly et al. frame it. The learning signal for the PV system is just the difference between the reward and PV value:

$$\Delta PV(s) = \alpha[R(s) - PV(s)] \tag{3.6}$$

Thus over time the PV system will become a model of the reward signal.

The LV system learns based on the same error signal as the PV system $(R(s) - LV(s))$, but conditioned on the PV signal. That is,

$$\Delta LV(s) = \begin{cases} \alpha[R(s) - LV(s)] & \text{if } PV(s) > \theta \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

If we think of $PV(s)$ as equivalent to $R(s)$, as it will be after some period of learning, what this means is that the LV system can only learn in rewarded states.

At first it is not clear what the LV system adds to the basic associative RL framework. The key is to note that the PV and LV functions are being approximated via neural networks. In other words, updates to state s will generalize to nearby states. In the PV system any over-generalization is corrected by the error signal in Equation 3.6. But in the LV system there is no correction, because the error signal is explicitly blocked in other states. Thus the LV function represents a generalized version of the PV function.

This can be thought of as a very rough form of prediction, essentially predicting that states similar to rewarded states will lead to reward. This is not a very helpful prediction in large state spaces with long decision sequences, as it is limited by the generalization distance. It also relies on the assumption that all the states similar to the rewarded states are good, which may not be true for some problems. However, in tasks that do not violate that assumption and that have a relatively short decision sequence, the LV signal is a way of propagating reward information to the states immediately preceding the reward.

The actual error signal output from the PVLV model is based on separating the PV and LV systems into excitatory and inhibitory components. The prediction error is then the difference between those components. In the case of the PV system the excitatory component is just $R(s)$ and the inhibitory component is the PV signal described above, thus $\delta PV(s) = R(s) - PV(s)$. For the LV prediction error, O'Reilly et al. create two of the LV systems described in Equation 3.7 (LV^+ and LV^-) but with different learning rates. The prediction error is then the difference between these two systems, i.e. $\delta LV(s) = LV^+ - LV^-$. α^+ is set to be faster than α^-, so that initially $\delta LV(s)$ will be positive, and then as $PV^-(s)$ catches up $\delta LV(s)$ will go to 0. The overall PVLV error signal is then set to

$$\delta(s) = \begin{cases} \delta PV(s) & \text{if } PV(s) > \theta \\ \delta LV(s) & \text{otherwise} \end{cases} \tag{3.8}$$

In other words, it is the PV error in rewarded states and LV elsewhere. The result of this will be that the PVLV system will give an initial prediction error in all states near the reward, which will decrease to zero over time. As mentioned, in large decision problems this is likely not helpful, but in short decision problems the PVLV signal will give a prediction error in the rewarded state and in the states preceding reward, which can allow learning to expand slightly out from the reward.

O'Reilly and Frank (2006) present a model that applies the error signal calculated by PVLV to a specific model architecture based on the prefrontal cortex, basal ganglia, and working memory (the PBWM model). The basic feature of this model is a working memory system that can store input values over time. What is being learned is a gating signal on the input to this memory; essentially the system is learning whether the stimulus should be stored in memory or not. The basal ganglia is used to compute this gating signal, based on inhibitory competition between populations of "Go" and "No-Go" neurons. Thus what needs to be learned is the connection weight from the stimulus to the Go and No-Go neurons; if the weight is higher on the Go neurons then the stimulus will be stored in working memory, and vice versa.

PVLV is used to compute the error signal in this model. It is able to use its generalization based error propagation to propagate the reward signal from the reward time (at the end of the working memory maintenance period) to the time of stimulus onset (when the Go/No-Go decision needs to be made). As an example, suppose storing stimulus "A" in working memory results in reward. If the network is presented with "A" and randomly makes the correct decision to store it in memory ("Go"), then it will get a reward signal at the end of the trial. This will cause the PV system to learn a positive value for the state of having "A" in memory at the end of the maintenance period. The LV system can then generalize that to the state of having "A" in memory at the beginning of the maintenance period. This will cause a positive prediction error when "A" is loaded into memory.

The learning rule is set up so that positive errors increase the weight from the stimulus to the Go neurons and decrease the weight to the No-Go neurons, and vice-versa for negative errors. For example, when the system gets a reward for storing "A" in memory, it increases the weight from all the active stimulus neurons to the "Go" neurons, which will increase the likelihood of storing "A" in memory in the future. Note that this requires that each stimulus value is represented by a different group of neurons. If the same neurons were involved in representing both "A" and "B", this would also increase the "Go" weighting for "B" stimuli, which is not desirable. Thus there need to be different neurons representing each possible stimulus value. This is not ideal, as it means that the dimensionality of the neural state increases with the number of stimuli values (thus exponentially increasing the neural state space). However, as long as the input only takes on a few different values this system will be sufficient for learning which of those stimuli should be stored in working memory.

This system can be extended by having multiple working memory storage spots (called "stripes" by O'Reilly and Frank), each with its own input gate, thereby allowing multiple values to be stored in memory. In this case a separate learning process is needed for each input gate. O'Reilly and Frank (2006) compute an individual error for each gate by multiplying the "Go" activity of that gate by the PVLV signal. Since the reward is based on the contents of working memory, only gates that are open (high "Go" signal) are contributing to the current prediction error (by changing the contents of memory). Therefore the result of this heuristic is that the learning update will only be applied to the gates that are contributing to the error.

O'Reilly and Frank (2006) use this stripe system to simulate sequential decision making (i.e., non-associative RL). For example, a simple 2-step decision problem might be "store A in memory, then if A is in memory store B in memory". This kind of problem can be solved in this system by storing "A" in one stripe and "B" in another, and making the content of the stripes part of the RL state. Effectively what this does is load all of the sequential decisions into the current state, so that a sequential RL problem can be turned into an associative RL problem. However, this approach will quickly run into scaling problems as the length of the sequence increases. Effectively it makes the dimensionality of the state space proportional to the length of the decision sequence, and since the size of the state space increases exponentially with the dimensionality this will be problematic for even relatively simple tasks.[3]

The main contribution of this work is the detailed comparisons it makes to experimental data. For example, O'Reilly et al. (2007) show how the PVLV model can explain the pattern of dopamine responses observed experimentally in several different conditioning paradigms, such as blocking (where learning one stimuli–reward association prevents the animal from learning associations from new stimuli to the same reward). The authors also establish a detailed mapping between functional components of the models and neuroanatomical components in the brain. O'Reilly et al. (2007) go through each of the major components of the model (PV^+, PV^-, LV^+, LV^-) and link to data showing specific brain regions that contain the signals contained in those components and connect to each other in the way required by the model. Similarly, O'Reilly and Frank (2006) show, for example, how the Go/No-Go gating structure can be implemented by the direct and indirect pathway through the basal ganglia. This type of work is very helpful for verifying the biological plausibility of a model, as it provides a map for comparing neurophysiological data, such as neural recordings and lesion studies, to the model. It is also helpful in future modelling work, as new models can either use the same mapping (increasing the evidence for that association) or propose an alternate one (providing a useful way to experimentally differentiate the two models).

3.1.4 Eligibility traces

A different approach to reward propagation is the use of eligibility traces. The basic idea of an eligibility trace is to add a slowly decaying representation of some signal of interest. Adding eligibility traces into RL is a way of adding memory into the system, so that instead of operations only applying to the current state they can be applied to states in the recent past.

This was first introduced in the computational realm as TD(λ) (Sutton and Barto, 1998). TD(λ) modifies the value update in Equation 2.8 to be

$$Q_{k+1}(s,a) = Q_k(s,a) + \lambda^t \alpha_k \delta(s', a') \tag{3.9}$$

where λ is the decay rate of the eligibility trace (some value < 1) and t is the number of steps between s and s'. In other words, rather than just updating the state immediately

[3]Note that this is on top of the already problematic scaling related to the number of values each stimuli dimension can take on, discussed above.

preceding the prediction error, we update all the states leading up to that prediction error, weighted by the decaying eligibility trace. This allows reward to propagate across multiple states, rather than one state at a time.

Neural models can incorporate eligibility traces by adding the trace to the Hebbian or STDP weight update. For example, the reward modulated Hebbian rule (Equation 3.4) can be modified to

$$\omega_{ij} = \alpha R \lambda(a_i b_j) \tag{3.10}$$

where $\lambda(a_i b_j)$ is a decaying representation of the correlation between a and b. If, for example, a_i is representing the state, this will be analogous to Equation 3.9 in that it will update the weights of all the recently active states, as they will still have positive eligibility traces, rather than just the currently active state.

Thus a model can use essentially the same associative RL framework as the models in Section 3.1.2, but with the benefit of eligibility traces the model can learn a value for the states leading up to the reward, rather than just the state with immediate reward. Vasilaki et al. (2009) and Friedrich et al. (2011) are examples of this approach, combining associative RL with eligibility traces.

However, there are a few downsides to the eligibility trace approach. The main one is that, similar to the PVLV model, it just extends the associative RL framework some limited number of steps. After some point the eligibility trace will have decayed to a point where the model cannot distinguish it from zero, which will mark the limit of how far away from the goal the agent can make effective decisions.[4] In associative RL the limit is one step, here it is n steps, but there is still a fixed limit.

The length of n will be determined by the decay rate of the eligibility trace, which brings up the second issue: it is not biologically plausible to have long eligibility traces. The question is what the underlying neural mechanisms are that are actually implementing the eligibility trace; what is it that is preserving the neural activity over time? Most models attribute eligibility traces to the post-synaptic currents resulting from neurotransmitter release at a synapse (Izhikevich, 2007). However, post-synaptic currents only preserve information on the order of milliseconds, which will not give a useful extension of n for most RL problems. If the eligibility trace is being used to propagate information across a task involving long sequences of decisions, as in the models of Vasilaki et al. (2009) and Friedrich et al. (2011), then the model needs to posit eligibility traces lasting at least tens of seconds, for which there is not a well-established neural mechanism. Effectively what these

[4]Note however that in purely computational systems (with perfect precision in the represented values) there is no such limit, since the eligibility trace can be tracked indefinitely.

rules do is solve the RL problem by offloading reward propagation onto some unspecified synaptic memory mechanism; without an explanation for how that memory mechanism works, there are still important open questions about how this theory could actually be implemented in the brain.

3.1.5 TD Error with eligibility traces

The most thorough neural reinforcement learning model is the work of Potjans et al. (2009). Their model also makes use of eligibility traces, but not in the same way as above. Rather than using eligibility traces to replace the TD error calculation, this model uses eligibility traces to compute the TD error.

The main difficulty in the TD error calculation (Equation 2.9) is the need to compare $Q(s,a)$ and $Q(s',a')$. Since both those values are samples from the same Q function, the model needs a way to preserve the $Q(s,a)$ value while the state switches to s'. In the Potjans et al. model only a value function $V(s)$ is represented rather than state-action values, but the challenge is the same.

The key idea of Potjans et al. (2009) is to apply two different eligibility traces to the output of the neurons representing V. When the state transitions from s to s' this gives two traces

$$e_1(s') = (1-\lambda_1)V(s) + \lambda_1 V(s') \tag{3.11}$$

and

$$e_2(s') = (1-\lambda_2)V(s) + \lambda_2 V(s') \tag{3.12}$$

where λ indicates the decay rate of the eligibility trace.

It can be observed that $\lim_{\lambda_1 \to 1} e_1(s') = V(s')$ and $\lim_{\lambda_2 \to 0} e_2(s') = V(s)$. That is, the eligibility trace with the higher decay rate can be thought of as a representation of $V(s')$ and the slower trace can be thought of as $V(s)$. The TD error can then be computed as $r + \gamma e_1 - e_2$. Thus if the V neuron output is connected to a postsynaptic cell via the two eligibility traces, the synapse has all the information locally it needs to compute the TD error. Note that it is not necessary that e_1 or e_2 be exactly equal to $V(s')$ or $V(s)$, because all that is required is a relative measure of whether the state value is increasing or decreasing. If $\gamma e_1 > e_2$ that will result in a positive prediction error and an increase in the synaptic weight, and vice versa, until the value function converges to the point where prediction errors are zero.

The advantage of this approach is that it does not require implausibly long eligibility traces. The eligibility traces are not required to propagate information across multiple decision steps; the propagation is being taken care of by the TD update, and the eligibility trace is only required to preserve information across a single state change.

However, this is also a downside of this approach, in that it imposes a fixed time period during which the TD error can be computed. If the TD update does not occur within the time window dictated by the decay rate of the slow trace, then both e_1 and e_2 will approach $V(s')$ and it will not be possible to compute a meaningful TD error. And given biologically plausible decay rates, that time window is quite short (tens of milliseconds).

One effect of this limitation is that it requires Potjans et al. (2009) to posit a relatively complex mechanism that moves synapses into and out of a "plastic" state, so that weight updates will only occur within that time window. This is problematic from the perspective of biological plausibility, but it also indicates a more practical problem with this technique; it is somewhat fragile, and relies on fine-grained timing and coordination of several different signals, which can be difficult to accomplish in a realistic neural model. In addition, this approach requires different neurons to be associated with each state, so that only the synapses corresponding to state s will be in the plastic state when the TD update occurs. This means that the required number of neurons increases with the number of states, and since the number of states increases exponentially with the dimensionality of the state space, this will be problematic when scaling up to complex state spaces.

The fixed time window is also problematic when we consider an SMDP framework. A fixed window is feasible in an MDP framework because rewards and state transitions all occur on a fixed schedule, which we can assume falls within that window. But in an SMDP environment rewards can occur at arbitrary times throughout the delay period; we have no guarantee that they will fall within the eligibility trace time window, in which case this method will not be able to compute the TD error. This is an even greater problem in the case of hierarchical RL, as the state may be changing during the delay period; in that case the value trace from the beginning of the delay period will have long since been replaced by intermediate values by the end of the delay period. Thus while the Potjans et al. (2009) model is a solution to the basic TD RL problem, we will not be able to use this method as we move to more complex RL algorithms.

3.2 Hierarchical reinforcement learning

In contrast to standard RL, there has been almost no previous work on recreating the computational theory of hierarchical reinforcement learning in a neural model. There has been limited work on combining function approximation with HRL (Bakker and Schmidhuber, 2004; Jong and Stone, 2009; Cuayáhuitl et al., 2012), but none using even artificial neural networks, not to mention biologically based systems. In this section we will discuss the two closest pieces of related work. The first is work by Botvinick et al. (2009) that examines how the actor-critic architecture can be extended to incorporate the needs of HRL. The second is work by Frank and Badre (2012) that is not an effort to directly implement HRL but does operate in a hierarchical fashion.

3.2.1 Actor-critic HRL model

The work of Botvinick et al. (2009) examines how the actor critic architecture (Section 2.1.4) can be modified in order to implement the options framework of HRL (Section 2.2.2). The implementation itself is purely algorithmic, with no neural components, but Botvinick et al. include a detailed discussion of how their model could map onto neural components in theory, and so we include it in this review.

The most significant change is that the actor and critic need to maintain different functions for each abstract option. If the agent selects option o_1 it then needs to select subsequent actions according to the policy of o_1; thus we need one actor function for o_1, another for o_2, and so on. Similarly, the critic will need to maintain different value functions for each option; this is because each option can have its own subgoals, thus the value of a state can be different depending on which option is currently active.

Previous work on neuroanatomical mappings of the actor-critic architecture generally situate the actor in the dorsolateral striatum and the critic in the ventral striatum/dopaminergic systems. Botvinick et al. maintain this mapping, but it needs to be extended to allow for multiple functions. They propose that prefrontal regions (dorsolateral prefrontal cortex and supplementary motor area) maintain a representation of the current option. This representation then activates different pathways in the dorsal/ventral striatum through a gating mechanism, in order to switch between the different actor/critic functions (this is based on the guided activation theory of Miller and Cohen 2001).

The actor also needs to check whether the current option has terminated (e.g., by reaching a goal state), and communicate that termination to the critic in order to trigger a learning update. Botvinick et al. point to previously observed signals in the dorsolateral striatum that are aligned to the beginning and end of extended temporal action sequences as neurophysiological evidence for this function.

Because we are now in an SMDP environment, the critic will need to track the accumulated reward over the course of the option. It will also need to remember the state in which the option was initiated, so that the appropriate $C(s)$ and $K(s, o)$ values can be updated when the option terminates. Botvinick et al. attribute these functions to the orbitofrontal cortex, based on evidence there of reward signals spanning temporally extended action sequences.

One final extension is the need to generate the pseudorewards. In Botvinick et al. (2009) the pseudorewards are taken to be given externally, thus they are not included in the actor-critic structure they propose here.

As mentioned, Botvinick et al. implement this actor-critic model in a purely computational framework, using lookup tables to represent the actor/critic functions and directly computing the required learning updates. Thus there are still open questions about how exactly these HRL extensions would be implemented in a neural system. However, what this work demonstrates is that the broad algorithmic requirements of HRL are consistent with behavioural and neurophysiological data, and the neuroanatomical mapping Botvinick et al. provide helps guide the development of more detailed models. The neuroanatomical mapping of this model (Section 4.5) draws heavily from that of Botvinick et al. (2009).

3.2.2 Hierarchical PBWM

In Frank and Badre (2012) the authors extend their previous PBWM model (see Section 3.1.3) to a hierarchical architecture. In the PBWM model the stimulus is connected to a Go/No-Go gating system, which controls whether that stimulus is loaded into working memory. In the Hierarchical PBWM model (HPBWM) there are two of these systems in parallel. The two levels interact through output gating; the output of the higher level working memory is connected to another gating system that controls the output of the lower-level working memory. The output of the low-level working memory drives the system's motor response.

To explain the operation of this model in more detail, imagine a task where the agent receives a sequence of stimuli "AX, AZ, BX, BY, BZ, AY, AX". When the agent sees an "A", it should press a button whenever it sees an "X". If it sees a "B" it should never press the button. Thus the correct response to the above stimuli sequence would be "press, -, -, -, -, -, press". To solve this task, the HPBWM model would first need to learn that "A" and "B" stimuli should be stored in the high level working memory, and "X", "Y", and "Z" should be stored in the low level. This can be accomplished via learning on the input gates, which works just like in the PBWM model (i.e., there is no hierarchical interaction). This requires that "A" and "B" are represented in one input "stripe", and "X", "Y", and "Z" represented in a different input stripe. The input gate for the low level can then learn to be closed for the first stripe and open for the second, and vice versa for the high level.

Given that the stimuli are routed to the appropriate memories, the question then is how to learn the joint conditions (i.e. if “A” is in one and “X” is in the other then press the button). In this model the interactions are all based on output inhibition. For example, suppose that the motor response is to press the button whenever “X” is output from the system (i.e., out of the low level working memory). The output gate on the low-level working memory therefore needs to learn to inhibit the memory output when there is a “B” in the high level working memory. This is the key hierarchical addition in the HPBWM model.

The error signal used in the learning is the same PVLV signal throughout, as in the basic PBWM model. Recall that this error signal drives the Go/No-Go activity in the gates (increasing Go weighting for positive errors and decreasing for negative). Imagine the network gets the input “BX”. If the output gate is open, this will result in a negative reward, and a negative prediction error from the PVLV system. This will decrease the weight on the connection from the “B” neurons in the high level working memory to the “Go” neurons in the output gate, thus making it more likely that the gate will be closed for future “B” stimuli. In contrast, opening the gate when “A” is in memory will lead to a positive prediction error, making it more likely that the gate will be open in the future. Thus over time the system will learn to open the gate for “A” and close it for “B”.

The downsides of this model are essentially the same as for the base PBWM model; namely, it will have difficulty scaling up to complex state spaces (with a large number of dimensions, or a large number of values for each dimension), due to its localist representation scheme. In addition, this approach will only really work on associative HRL problems. In theory the same technique can be used as in the PBWM model, where a sequential decision problem is transformed into an associative RL problem by incorporating all the past steps into the current state. However, as discussed in Section 3.1.3, this will be impractical for even relatively short decision sequences. Since one of the primary motivations for HRL is to enable scaling up to long, complex decision sequences, it can be seen that this model will not help in that regard.

However, it is important to note that the Frank and Badre (2012) model is not intended to be a direct implementation of the computational theory of HRL. It is designed specifically for tasks with hierarchical spatial structure, not hierarchical temporal structure. That is, the above task is hierarchical in that the response to a stimulus such as “AX” has a hierarchical dependence of one part of the stimulus (“X”) on the other (“A”). There is no hierarchical structure in the “AX, AY, BX, ...” temporal sequence. So the above problems are not intended as a critique of the Frank and Badre (2012) model, but rather to show that the problem it addresses is not the same as the general computational theory of HRL.

As with the PBWM model, one of the main strengths of the HPBWM model is its extensive comparisons to experimental data. For example, Frank and Badre map the different hierarchical levels of the model onto an observed rostral–caudal hierarchical organization in prefrontal cortex (Badre et al., 2010). They also compare the results from their model to human performance on the same task. Together with the Botvinick et al. (2009) work, this gives quite a thorough account of the neural basis of hierarchical processing, which is helpful in any future model of HRL.

3.3 Summary

The previous modelling work can be best summarized by highlighting some of the open problems that remain to be solved. Even ignoring HRL, it can be seen that there remain many challenges within the biological modelling of basic reinforcement learning. Much of the existing work has been limited to associative RL (Seung, 2003; Florian, 2007; Baras and Meir, 2007; Izhikevich, 2007; Urbanczik and Senn, 2009; Stewart et al., 2012) or extensions of associative RL (O'Reilly and Frank, 2006; O'Reilly et al., 2007; Vasilaki et al., 2009; Friedrich et al., 2011; Frémaux et al., 2013).

The most complete RL model is that of Potjans et al. (2009). This model is able to compute a TD error signal within the model, and use that error to learn a value function and policy. However, as discussed in Section 3.1.5, the eligibility trace approach of this model imposes a number of limitations. One of the most critical is that it forces a localist representation scheme, which will be problematic when trying to scale this technique up to complex problems. In addition, the eligibility traces impose a fixed time window on the TD update, which is not suitable for SMDP problems. This is not a flaw in the model, as it was only designed for MDP problems, but it means that it is not obvious how to extend the approach of Potjans et al. (2009) to operate in a hierarchical fashion.

In the case of HRL, it can be seen that there is almost no prior work in the field of neural modelling. Botvinick et al. (2009) have done extensive work analyzing what would be required to construct a biologically plausible model of HRL, but an actual neural implementation remains theoretical. The closest related work is that of Frank and Badre (2012). This model is capable of performing limited hierarchical reasoning, but it is only practically applicable to problems with relatively simple state spaces and associative learning.

Thus it can be seen that there are many open questions concerning neural implementations of RL and HRL. In the next chapter we discuss the new model developed in this work, and how it helps to address these questions.

Chapter 4

Model

This chapter describes the key contribution of this work: a neural model of hierarchical reinforcement learning. We have divided the structure of the model into three main components, which we term action values, action selection, and error calculation (shown in Figure 4.1).

We begin by discussing each of these components in turn, and show how they implement their respective aspects of reinforcement learning. Together these components form a flat, non-hierarchical system. Although the underlying design decisions were made with the needs of a hierarchical system in mind (e.g., SMDP processing), this aspect of the model can be understood without any reference to HRL. We largely present it as such, as it represents an interesting contribution on its own. After the basic model is presented, we then show how these elements can be composed into a hierarchical structure.

We present the model without reference to any particular task or environment. The model is designed as a generic reinforcement learning system, thus the implementation is not based on any specific task. Rather, the components are constructed in as general a fashion as possible, so that this same model can be applied in many different environments.

To this end, the agent treats the environment largely as a black box. Its only insight into the environment is a real valued vector that the environment makes available—the state representation. This state is assumed to be continuous in both time and space. The only way for the agent to interact with the environment is by outputting a real valued vector representing an action.[1] The agent assumes that the action space is given (i.e., the

[1]This model is designed for the case where there are a finite set of actions to choose from, each represented by a different vector. We discuss how it might be extended to operate with continuous actions in Section 6.2.3, but for the discussion here we focus on discrete actions.

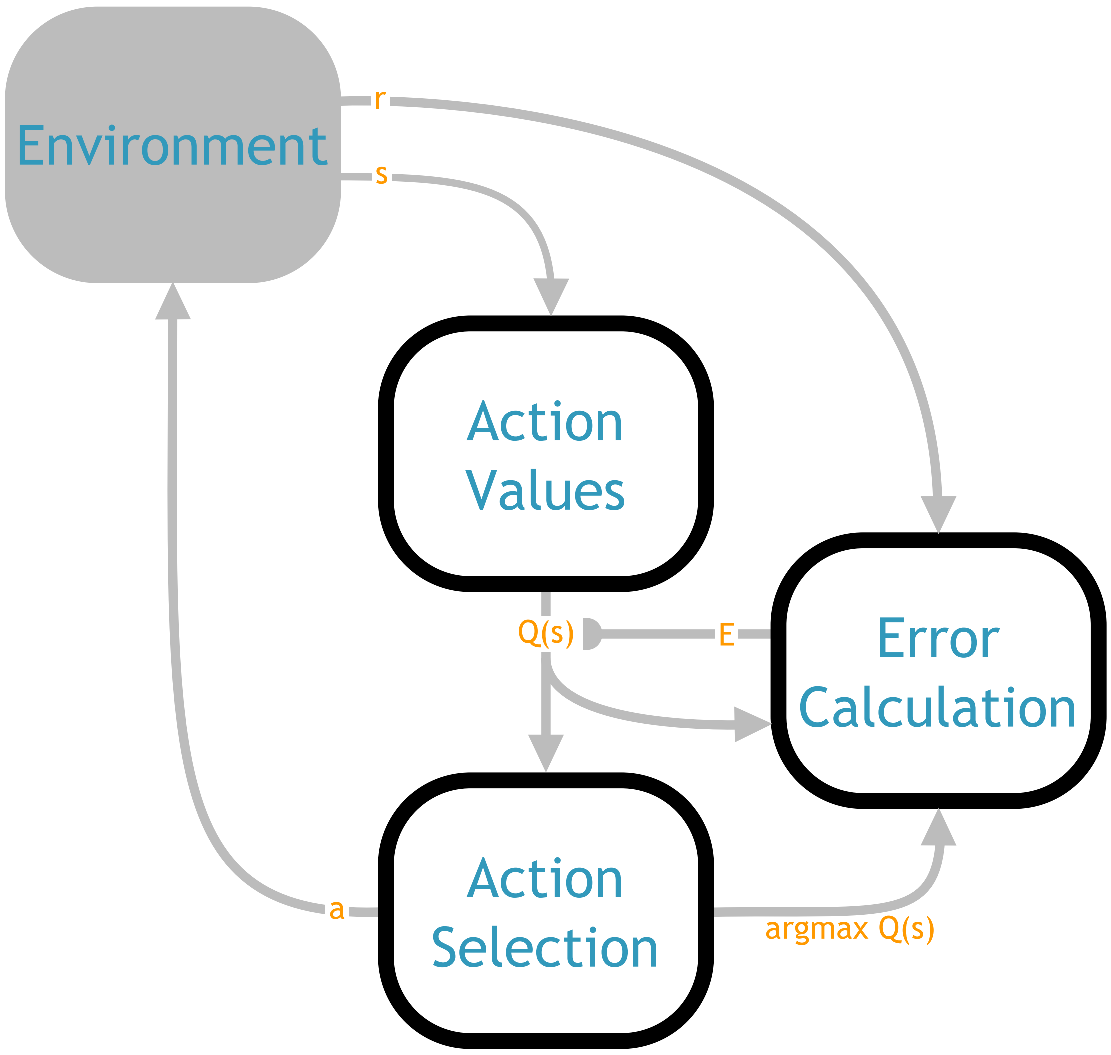


Figure 4.1: Overall architecture of the model, showing the three main components and the functional values flowing between them. The action values component computes the Q values given the state from the environment. The action selection component determines the highest valued action, and sends the action itself to the environment and the identity of the selected action to the error calculation component. The error calculation component uses the Q values and environmental reward to calculate the TD error, which it uses to update the Q function in the action values component. Triangular arrowheads indicate a vector value connection, semicircle indicates a modulatory connection that drives learning.

agent knows what options it has to choose from), but it does not know anything about the effect or value of those actions. The environment takes the action output from the agent and updates the state through some unknown internal mechanisms. The only feedback the agent gets is a scalar value from the environment, the reward, and it will seek to select actions that maximize the long term cumulative value of that reward.

Throughout this discussion we focus on the computational side of this work—that is, the practical concerns of how to implement the required functions in a neural model. We will discuss how the constraints of biological plausibility shape the design of the system, but for the sake of clarity we avoid going in-depth into neuroanatomical comparisons. We save that discussion until the conclusion of the chapter, where we examine in detail how the various aspects of the model map onto neurophysiological data.

4.1 Action values

The first basic element of the model is the representation of state-action values—that is, we want to build a neural representation of the Q function. Although this could be seen as the most crucial aspect of the model, its structure is relatively simple.

The central feature of this component is a single population of neurons. These neurons take the environmental state s as input, and output an n-dimensional vector (n is the number of available actions, $|A|$) where each element represents the Q value of that action in the current state. We will refer to this vector as $Q(s)$, i.e., $Q(s) = [Q(s, a_1), Q(s, a_2), \ldots, Q(s, a_n)]$. Note that when we say that the population outputs a vector, we refer to the value decoded using the techniques of the NEF (see Section 2.3). The actual output is a vector of neural activities with length m, where m is the number of neurons in the population, but it is generally more useful to refer to the represented value.

The question is what the decoders should be so that, given an input of s, the neurons will output $Q(s)$. Since we do not know the correct Q function ahead of time, we cannot analytically determine the decoders as in Equation 2.35.[2] Therefore the decoders need to be learned, via the learning rule in Equation 2.40 (i.e., $\Delta d_i = \kappa E a_i(x)$). E in this case is an n dimensional vector containing the error for each dimension (i.e., each action). In theory this could be a different error for each dimension, but in this model the error is equal to the TD error for the selected action and zero for all other actions. We will discuss how this error is computed in Section 4.3.

[2]If, for the sake of argument, we did know the Q function, we could simply set $f(x) = Q(x)$ in Equation 2.39. This can be useful to seed the Q function representation with some initial values.

The effect of this error will be to adjust the decoded value for the selected action in the direction of the prediction error. It accomplishes this by increasing the weight on all the active neurons if there was a positive error, and vice versa for negative. Note that this is essentially linear function approximation with a weight update as in Equation 2.14. Tsitsiklis and Van Roy (1997) proved the convergence of this approach under on-policy updates, which we will employ in this model. Note that the model developed here violates many of the simplifying assumptions made in that proof (e.g., non-independent basis functions, continuous time, an SMDP framework, etc.). However, this gives us some confidence that this approach is a reasonable one.

4.1.1 State encoding

It can be seen in Equation 2.40 that the learning update is dependent on the activity of the neurons being decoded (in this case these are the neurons representing the state). This leads to an important point on the practical usage of Equation 2.40—namely, the neural activities must be correlated with the important variables of the task.

To see why, imagine a set of neurons where each neuron responded with a constant activity level across the state space (i.e., no correlation between activity and state). Since we are using a linear decoding, this means that the decoded value will be constant across the state space; in other words, this population will only be able to learn a flat, uniform Q function. It will learn the best flat approximation of the Q function given the errors it receives, but that will still likely be a very poor representation of the true Q function.

Another way to put this is that the Q function we learn is a linear composition of the basis functions (the neural response functions). If we want a good Q function approximation, then those basis functions need to give a useful encoding of the function space. The shape of the neural response function is determined by the NEF encoding formula (Equation 2.31). The neuron model G is constant across all the neurons (the LIF model in Equation 2.32), and α and J^{bias} are constant across the state space. Thus the key factor is $e_i x$, the dot product between the state and the neuron's encoding vector.

In other words, having appropriate encoders is important to the quality of the Q function representation. For example, imagine an environment with a 2-dimensional state, where all the neurons are equally sensitive to the two dimensions (i.e., $e_i = [c_i, c_i]$). The only type of function that can be represented by this population will be one that is symmetric around the line $y = x$. This is a somewhat trivial example, and an obviously poor encoding, but it highlights the role the encoders play in the Q value representation.

The standard NEF approach is to use random unit vectors for the encoders. This works fairly well, in that it spans the space of possible basis functions; by simply adding more neurons we are guaranteed to get a good basis representation at some point. However, in practice we often have limited neural resources, both in terms of the simulation (where adding more neurons increases the computational costs) and the biology (where the brain has a limited number of neurons to work with). If the state space is uniform, then random encoders is about the best we can do.[3] However, in most problems the state space is not uniform; some areas of the space are visited more frequently, some not at all. Randomly choosing encoders can result in many neurons that are not providing a useful encoding of the state space.

Thus it can be helpful to choose the encoders more carefully, ensuring that they lie in useful regions of the state space. In our model the encoders can be manually adjusted on a per-task basis. This requires the input of some domain knowledge on the part of the modeller, which is not ideal; a complete description would provide some explanation for how these encoders could be automatically learned from the environment. However, the challenge of that kind of unsupervised learning is an entire research field to itself, and not one we propose to solve in this model.[4] Also note that we are not increasing the representational power of the model by manually generating encoders, since the representational space is upper bounded by the random encoding, in the limit. We are just making the representation more efficient—providing a better representation with fewer neurons. The encoding vectors we use also represent a prediction as to the kinds of neural responses that usefully encode a given problem space. We presume that real brains are trying to solve a similar problem—how to best encode a space with limited resources. Thus, for example, an experimenter could compare the response functions of neurons in our model to recorded responses of neurons in some brain area, to see if their response has been optimized in the way we predict. Alternatively, experimental data can be used to suggest useful/appropriate encodings for a model.

In some cases it can be helpful to go beyond the linear vector-based encoding. For example, one of the downsides of the standard encoding vector approach is that it does not lead to very sparse neuron activity. Neuron activity will differ in magnitude, but overall neurons tend to be active across broad regions of the state space. The area of activity can be imagined as a cone extending from the origin in the direction of the encoding vector. Thus this encoding tends to provide good differentiation in activity if the input moves

[3]Evenly tiled encoders would be the best, but in practice there is little difference between tiled and random given a certain minimum number of neurons.

[4]See Voelker et al. (2014) for an example of work that begins to explore the issue of learning encoders in the NEF.

laterally around the origin, but poor discrimination radially. This can be problematic for learning Q functions with local nonlinearities in them, as adjusting the weight on a neuron will adjust the Q function across a broad region of the state space. In addition, the neural activities are not evenly distributed across the state space. A greater proportion of neurons will be active for inputs at the "edge" of the state space (vectors with a larger norm). This can induce a bias in the learning rule, in that weight updates in one region of the state space will have a greater impact than the same weight update in a different state. The latter problem tends to conflict with the former, in that the more evenly distributed the activity is, the less sparse it will be.[5]

One way to resolve these issues is to project the state into a higher dimensional space. The actual state space then represents a slice through the higher dimensional encoding space, which will end up giving sparse, evenly distributed activations. For example, a 1D variable x can be projected into a circle in 2D space via $\tilde{x} = [\sin(x), \cos(x)]$. Encoders distributed around that circle will activate when the state passes through their region of the circumference. From the perspective of the 1D space, this will look like a neuron with a bump of activity around some value of x; thus we will have activity that is sparser (restricted to certain values of x) and evenly distributed across the state space. To return to the cone intuition, effectively what this does is force the input to move laterally through the encoding fields rather than radially. This principle works in the same way in higher dimensional spaces.

Another way of looking at this issue is from the perspective of a feature of the NEF called function representation (Eliasmith and Anderson, 2003). This involves specifying a set of basis functions, and passing the input through those basis functions. This will result in an m-dimensional vector, where m is the number of basis functions, at which point the standard vector encoding techniques can be used to encode that new vector. The basis functions can then be chosen in such a way as to give the kinds of encoding properties the modeller wants.

The previous technique can be seen as a special case of this one, where the basis functions are the spherical coordinate transforms. Another useful basis space are Gaussian radial basis functions, i.e.

$$\tilde{x} = \bigoplus_{p \in P} e^{\frac{-\|x-p\|^2}{2\sigma^2}} \tag{4.1}$$

(we use the $\bigoplus$ notation to mean concatenation of the elements into a vector). If the

[5]This is because the way to increase the sparsity of the representation is to increase the firing threshold of the neurons. But increasing the firing threshold pushes all the activity cones out from the origin, which reduces their overlap radially and thereby increases the bias of activity towards the outside of the space.

centres of these Gaussians ($p \in P$) are randomly distributed throughout the state space, this will result in the kind of activity we want—sparse, local, and evenly distributed. Adjusting the spacing and width (σ) of the Gaussians will give different levels of sparsity and smoothness. This highlights the strength of the function representation approach, namely, that it provides a lot of flexibility to the modeller.

The particular type of encoding that will be most useful differs from problem to problem, depending on the shape of the Q function. Thus, as discussed at the beginning of this section, this model does not define a generic state encoding scheme, but rather takes it to be something that is adjusted on a per task basis. However, this section gives some guidelines on what constitutes a useful encoding. In addition, the encoding schemes described here are some of the most general, and useful across a range of tasks. The random vector encoding is the most straightforward, and so is likely the first choice when beginning to explore a problem. If there is a known structure to the state space, then it may be possible to achieve more efficient encoding by picking the encoding vectors more carefully. Alternatively, if experimental data is available specifying certain neural encoding properties, then the encoding vectors can be chosen to match. If the task requires sparser activity, the Gaussian radial basis function encoding is a good way to accomplish this most of the time. Only in rare cases is it likely to be necessary for the modeller to go beyond these approaches, for example by defining more specialized basis functions.

4.1.2 Dual training system

Theory

Examining the decoder learning equation ($\Delta d_i = \kappa E a_i(x)$, Equation 2.40) reveals another important challenge for learning action values: the weight update is based on the neuron's current activity, $a(x)$. In the context of the action values component, the input is the state, so the update is dependent on the activity in the current state, $a(s)$. The problem is that the TD error (Equation 2.9) cannot be calculated until the agent arrives in s', because it requires the comparison $Q(s', a') - Q(s, a)$. At that point the input is s', not s, so the neuron activity represents $a(s')$ rather than $a(s)$. If the TD error is applied at that point, the effect would be to adjust $Q(s')$, not $Q(s)$. Thus the model needs to somehow apply the learning update based on prior neural activities. This is not a problem purely computational approaches worry about, as they can simply store the previous activation and recall it when needed. However, a biological model needs to explain how this can be accomplished neurally.

We need a brief aside here to mention what we mean by s and s', in the context of a system operating in continuous time and space. That is, $s(t)$ is a continuously changing signal, it is not divided up into previous and current states. By s we mean the value of $s(t)$ when the action was selected, and by s' we mean the value of $s(t)$ when the action terminates (or the current time, if it has not yet terminated). That is, if the system selects an action at time t_0, then $s = s(t_0)$. Action a will execute for some time τ (recall that a could represent an abstract action in the hierarchical case) and then terminate. The value of $s(t)$ at termination, $s(t_0 + \tau)$, is s', which is the value we want when computing the SMDP TD error as in Equation 2.18. We use the notation s and s' for the sake of simplicity, and to connect to the background discussion in Sections 2.1 and 2.2.

The problem is how to preserve the neural activity from when a was selected, $a(s)$, until a terminates. The standard approach to preserving neural activity over time is to use eligibility traces (e.g., Izhikevich, 2007). For example, if we changed the learning update to be

$$\Delta d_i = \kappa E e(a_i(x)) \tag{4.2}$$

where e denotes a decaying eligibility trace, then the learning update would be applied based on the previous activity contained in the eligibility trace, which could be $a(s)$. However, as discussed in Section 3.1.4, eligibility traces can only preserve information over fixed and, realistically, short time periods. In an SMDP environment there is an unknown and potentially lengthy time period separating s and s', so there is no guarantee that $e(a(x))$ will contain any useful trace of $a(s)$. Thus some other method is needed.

In this model we solve the problem via a dual training system, shown in Figure 4.2. Rather than a single population representing the Q function, the network contains two populations. Both are representing the same Q function, but one receives the current state as input and the other receives the previous state.

The TD update is only applied to the decoders of the previous state population. When the TD update is calculated in s' the activity of these neurons is $a(s)$, thus the appropriate Q values are updated, $Q(s)$. However, the action selection and error calculation needs to be based on the Q values of the current state, $Q(s')$; this is why we require the second neural population which receives the current state as input. The question then is how to update the decoders of the latter population.

The key idea is that the output of the previous state population can be used to train the current state population. Whenever s and s' are the same (or within some range in the continuous case), the output of the two populations, $Q(s)$ and $Q(s')$, should be the same.

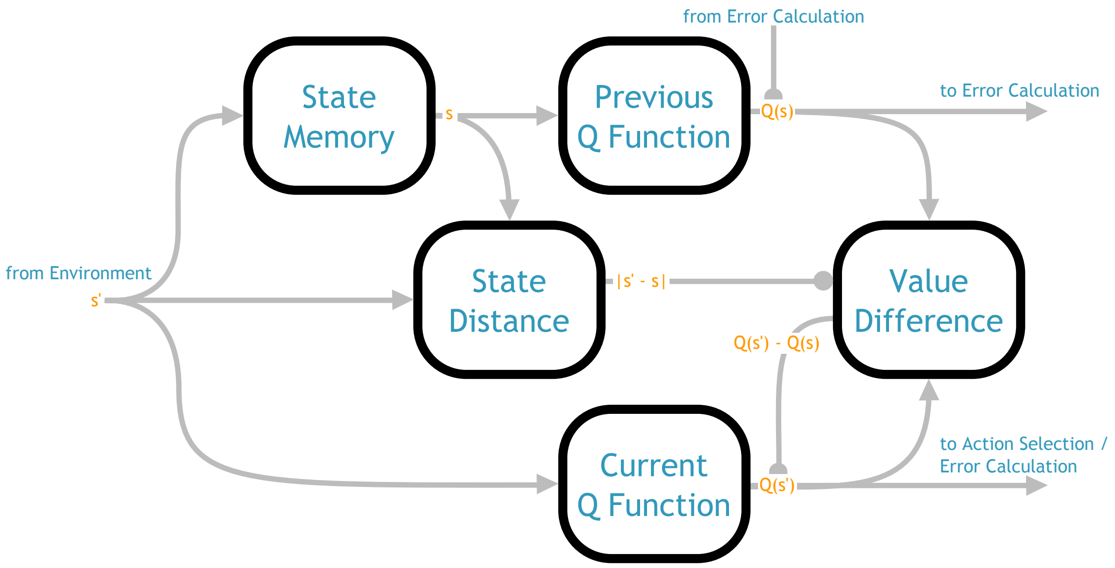


Figure 4.2: Architecture of the action values component. Computes the current and previous Q values ($Q(s')$ and $Q(s)$) based on the current and stored state, respectively. The previous Q function is trained by the TD error signal from the error calculation component. The current Q function is trained to match the previous Q function output whenever the distance between the current and previous state is below a threshold. The output of $Q(s)$ and $Q(s')$ are sent to the error calculation component, and $Q(s')$ is sent to the action selection component. Circular arrowhead denotes an inhibitory connection.

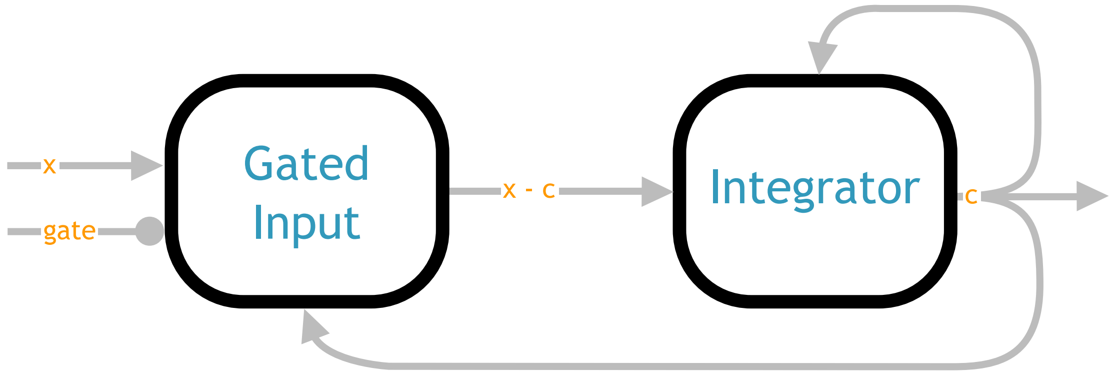


Figure 4.3: Generic memory circuit. x represents the target value, and *gate* controls when that value is loaded into the memory. In the context of the dual training system, $x = s'$ and $c = s$.

Therefore the difference $Q(s) - Q(s')$ can be used as the error signal for $Q(s')$. That is,

$$E = \begin{cases} Q(s) - Q(s') & \text{if } \|s - s'\| < \theta \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

This error signal is then used to update the decoders via the same decoder update rule (Equation 2.40). In this case the neuron activity is $a(s')$ and the goal is to update $Q(s')$, so there is no problem.

In summary, we solve the problem of backpropagating the error in time by maintaining an explicit representation of the previous state. The TD error is calculated in the current state, but then applied to the population representing the previous state. The learning is then propagated forward to the current Q population by training the current Q population to match the previous Q population whenever $s \approx s'$. We will now go into more detail on how each of those steps is implemented neurally.

Implementation

The first question is how to store the previous state. We already discussed in Section 2.3.2 how an integrator can be constructed using the NEF, and how an integrator can be used to preserve information over time (as given no input the population will maintain its current value). However, there are a few useful features that can be added to the basic integrator to make it a more powerful memory system.

First, it is helpful to be able to give a desired storage value as input to the memory, rather than a derivative. This can be accomplished by setting the integrator input to be the difference between the desired value and the current value stored in the integrator. The difference can be calculated by a population that takes the desired and current value as input, the latter with a transform of -1 (see Section 2.3.2, Equation 2.38). Integrating this input will cause the value stored in the integrator to move to the desired value, at which point the difference will be zero and the integrator will maintain the desired value.

Another important feature is that we be able to control when the value is updated. For example, in the case of storing the previous state, the memory system will be receiving the continuous state signal $s(t)$ as the target value, but we only want to update the stored value at a particular point in time (when an action is selected). When the state changes in the future the memory should not continue to update, or it will just continue to represent the current state. This can be accomplished by creating an inhibitory connection between a signal that indicates when the value should be updated and the population calculating the difference. By "inhibitory connection" we mean connecting a signal to the neurons of a population via negative connection weights. Thus if the signal is zero the population will not be inhibited, so the difference will be calculated and fed into the integrator. If the signal is positive (e.g., one) the population will be inhibited, so its output will be zero and the integrator will maintain its current value.

In this model the timing signal is produced by the environment, not generated internally by the neural model. More accurately it is an action termination signal, rather than an action selection signal. That is, the signal indicates that the previous action has completed, which, by proxy, means that a new action needs to be selected. There are two ways in which an action can terminate: after reaching some state, or after some time period has elapsed. For example, an action such as "move forward" could terminate when a specific state in front of the current one is reached, or it could just move the agent forward for some time period and then terminate. The termination conditions are defined as part of the action, which, as mentioned previously, we take to be given along with the task. It would certainly be possible to compute these signals neurally; for example, a timer can be implemented by an integrator with a constant input. However, the boundaries of the model need to be drawn somewhere, and in this case we decided to compute these signals outside the model.

The complete memory circuit is shown in Figure 4.3. The network has two inputs, one indicating the value to be stored and the other indicating when the value should be stored. In the case of storing the previous state, the former signal is the state signal from the environment, and the latter signal is the one just discussed. Thus the stored value will be the value of the state at the beginning of the current action period.

The output of the memory network is then fed into the population of neurons representing the Q function for the previous state. The output of that network is trained via the learning rule in Equation 2.40. E in this case is the prediction error as in Equation 2.8; we discuss the neural circuit that calculates this value in Section 4.3. Note that this learning rule is described in terms of changes to the decoders d_i. This is exactly equivalent to a change in synaptic connection weights, as the weights can be calculated from the decoders by multiplying them by the gain and encoders (α and e) of the postsynaptic population, which are fixed (see Equation 2.41). The end result is that the Q function representation in the population is updated in the direction of the prediction error. The specific part of the function that is updated is dependent on the activity of the neurons. The neural activity is $a(s)$, meaning that the value of $Q(s)$ will be modified, thus implementing the desired TD update.

The population representing the current Q values receives the current state as input, which comes directly from the environment. The same learning rule is used to update the decoders of this population, but with a different error signal (Equation 4.3). The first aspect of this error signal is the difference $Q(s) - Q(s')$. This can be calculated by a population that takes the output of the two Q function populations as input, with a transform of -1 applied to the current state function (see Equation 2.38).

The second aspect of Equation 4.3 is the conditional, namely that the error should be zero whenever the distance between s and s' exceeds the threshold θ. To calculate that distance the first step is to calculate the difference between s and s'—this is done in the same way, by a population that receives s and s' as input, the latter with a transform of -1. Distance can then be calculated as the absolute value of this difference (i.e., $abs(s - s')$). The absolute value can be computed by setting $f(x) = abs(x)$ in the decoder calculation for the difference population (see Equation 2.35). x is the total input to the population, which in this case is $s - s'$, thus the overall population will be computing $abs(s - s')$. Summing across the dimensions (a simple linear transform, see Equation 2.36), will then give $\|s - s'\|_1$. Note that here we have used the L1 norm to represent distance. The L2 norm (or any other distance metric) could be used, but this is a more complex function involving squares and square roots, and thus more difficult to approximate using neurons. Since all that matters is whether the distance exceeds some fixed threshold, it is not that critical what the distance metric is and so we adopt the simplest.

The output of this population represents the distance between s and s'. The next step is to threshold that distance at θ, in order to get a signal that is zero when $\|s - s'\|_1 < \theta$ and one when $\|s - s'\|_1 \geq \theta$. This signal can then be used as an inhibitory signal, as in the memory network, in order to gate the output of the population calculating $Q(s) - Q(s')$.

The gating signal can be computed by creating a population that takes the state distance as input and setting its output function to be

$$f(x) = \begin{cases} 0 & \text{if } x < \theta \\ 1 & \text{if } x >= \theta \end{cases} \tag{4.4}$$

However, functions with sharp nonlinearities like that are difficult to approximate using the standard neural responses, and the end result of this will be a fairly fuzzy threshold around θ. This can be improved by using the neuron properties themselves to help the population approximate the function. Namely, neurons already have a built-in firing threshold; if their input does not exceed that threshold, their activity will be zero. The threshold point is determined by the value of J^{bias} in Equation 2.31. The standard NEF approach is to choose the thresholds randomly across the possible range of x. However, in this case we can set all the neuron thresholds to align with the threshold of the desired function. Thus the neural activity will always be exactly zero when $x < \theta$ and non-zero when $x \geq \theta$, which will result in a more accurate approximation of the desired function.

The final step is to connect this thresholded signal to the Q difference population via an inhibitory connection. Just as in the memory network, this means that when the signal is zero the output will be the normal output of the population $(Q(s) - Q(s'))$, and when the signal is one the output will be zero. Thus the output of the difference population will implement Equation 4.3. This output becomes the E in the decoder learning rule for the current state Q population. This will cause the current Q values, by proxy, to be updated according to the TD error, thus coming to represent the correct Q function. Those Q values are the key output of the action values component, which will drive the behaviour of the action selection component.

4.1.3 Important parameters

Throughout the model description we will point out the important tunable parameters of the model. This is not an extensive description of the model parameters—biological neural models have hundreds of parameters, related to different neural properties, connections, initialization conditions, and so on. However, many of these parameters do not significantly affect the function of the model, or they can be determined based on the basic design of the network and need not be modified. Instead we focus on the parameters that someone using the model might want to adjust in order to modify its performance on a given task.

The first such parameter in this component of the model is the number of neurons in the Q function representation. From a computational perspective this can be thought of as the number of basis functions in the function approximation. Roughly speaking, the more neurons used, the better the function approximation will be. Each neuron represents a degree of freedom in the learning process, thus the more neurons available the more closely the population will be able to fit the true value function.

To a machine learning audience this will immediately raise the question of overfitting. However, generally speaking, problems like overfitting are not as great a concern in RL. The goal is not to be able to generalize the Q function to novel areas of the state space—we do not have separate "training" and "test" states. The goal is to learn the best possible approximation based on the agent's samples as it explores the environment. When the agent enters an unfamiliar region of the state space the expectation is that those Q values will be learned by the TD learning process, not based on generalization from the Q function. Of course we do want some local generalization; the purpose of function approximation is to use a discrete set of basis functions to represent a continuous function space—if the basis functions do not generalize at all then the function approximation becomes a lookup table. However, the problem in that case has more to do with the shape of the basis functions, rather than the number of basis functions (i.e., decreasing or increasing the number of basis functions would not solve the problem). The basis function shape is determined by the state encoding, which was discussed in Section 4.1.1.

The real problem with increasing the number of neurons is that it increases the resource demands of the model. By this we mean both the computational resources, in that the more neurons the model contains the slower it will be to simulate, as well as the neurophysiological resources, in that the physical system we are attempting to model has a limited number of neurons. If a proposed model required trillions of neurons to represent the value function, that would be a problem for the biological plausibility of that model. However, in practice our simulation capacity falls well below the resources of real brains, so it is the former concern that dominates.

The tuning of this parameter amounts to a balancing act, where the modeller attempts to use the minimum amount of neurons necessary to achieve the desired accuracy. The required accuracy depends on the shape of the Q function. If, for example, the function is relatively smooth and has large differences between the values of the different actions, that can be represented more coarsely and therefore requires fewer neurons. If the task requires fine discrimination (e.g., a maze where several different paths all have almost equal length) that will require a more accurate Q function and more neurons. This parameter also interacts with the discussion of state encoding in Section 4.1.1. If the neurons are set to have a very local/sparse encoding (e.g., because the Q function is highly nonlinear) then more neurons will be required to effectively tile the space.

In conclusion, it is impossible to suggest a value for this parameter independent of the specific task. The best approach is to determine the minimum number of neurons empirically—define the minimum accuracy required for the target Q function, and increase the number of neurons until that accuracy is achieved. For reference, the tasks used in the results of this work use between 500 and 1200 neurons in each Q function population.

Another important parameter is the learning rate, α, used in the TD update formula (Equation 2.8), which is implemented by κ in the decoder learning update (Equation 2.40). This parameter is task dependent in that it depends on the magnitude of the prediction errors as well as the desired learning speed. The former is determined primarily by the reward structure of the task. For example, a task with rewards ranging between ± 100 will have very different prediction errors than a task with rewards from 0.5–1. The desired learning speed is determined primarily by the reliability of the prediction errors. For example, if there is significant noise in the prediction errors, perhaps because of an inaccurate Q function representation or stochastic rewards, then less weight should be given to each individual update, meaning a lower learning rate. If the prediction errors are reliable then the learning rate can be increased, to speed up the rate of convergence (although if the learning rate is too high it will cause the update to overshoot the target value, leading to oscillations).

One final practical note is that translating the α in Equation 2.8 to the κ in Equation 2.40 depends on the units of $a(x)$. That is, if $a(x)$ represents the average firing rate in Hertz, that will necessitate a different κ than if $a(x)$ represents a postsynaptic current. However, if the conversion between the different $a(x)$ units is known (that is, we know what the equivalent activity would be in the different units for the same x) then the learning rate can be converted in the same way.

The last parameter to discuss is the threshold θ on the state difference, used to gate the learning on the current state Q function. The modeller needs to decide what counts as "nearby" in the state space of a given task (i.e., what is the Δs such that $Q(s) \approx Q(s+\Delta s)$. This value will differ depending on what s represents. If the value is too small then it will take a long time for learning to transfer from the previous state Q function to the current state Q function. If it is too large, then learning in the previous state Q function will generalize to inappropriate states in the current Q function. However, in practice the model is not overly sensitive to small changes in this parameter, once it is in approximately the correct range. It can generally be set based on a quick examination of the state space, but again this requires some domain knowledge on the part of the modeller.

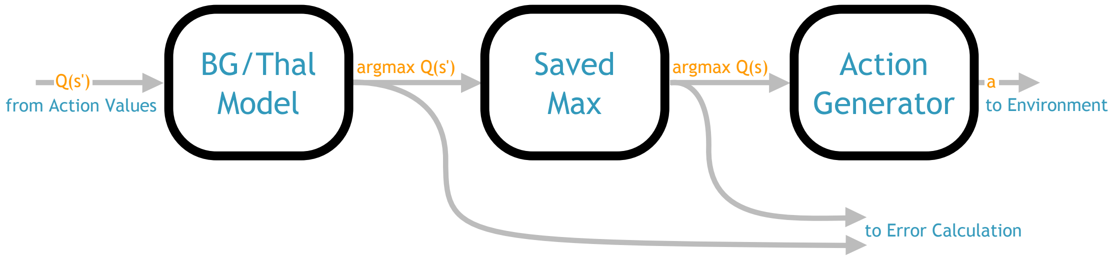


Figure 4.4: Architecture of the action selection network. The leftmost component represents the model of the basal ganglia and thalamus described in Section 2.3.4. The centre component stores the output of the previous component across the SMDP time delay. The rightmost component transforms the argmax vector into the vector associated with the selected action.

4.2 Action selection

The task of the action selection component is to select an action based on the output of the action values component. In other words, it needs to convert the Q values into a policy. The core of this component is the basal ganglia/thalamus model discussed in Section 2.3.4, based on work by Gurney et al. (2001) and Stewart et al. (2010). Recall that the function computed by this network is essentially an arg max; given a vector of Q values as input, it will compute

$$\arg\max Q(s) = \bigoplus_{i=1}^{n} \begin{cases} 1 & \text{if } a_i = \arg\max_a Q(s) \\ 0 & \text{else} \end{cases} \tag{4.5}$$

That is, a vector of the same length as $Q(s)$, with 1 for the highest valued action and 0 elsewhere.

Thus the basal ganglia is well suited to computing a simple greedy policy, the model just needs to convert the output of Equation 4.5 into an action for the environment. Recall from the introduction of this chapter that actions in this model are represented by an abstract vector. The environment knows what the vectors mean, but the agent does not; to the agent they are just a set of generic possibilities to choose between. For example, the agent might have four vectors it can output: $[0.3, 0.2, \ldots]$, $[-0.1, 0.4, \ldots]$, and so on. To the environment these might represent movement in the four cardinal directions in a 2D space, but to the agent those are just the four different outputs it can produce.

The mapping from Equation 4.5 to action vectors can be accomplished by a single population. First, a matrix A is constructed by stacking the available action vectors, i.e.

$$A = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

The output of the thalamus is then connected to a neural population using that matrix as the transform (see Equation 2.36). The resulting value will be whichever row in the matrix corresponds to the 1 in the thalamus output, i.e., the vector of of the highest valued action. This therefore implements the basic action selection process. However, there are a number of subtleties that need to be addressed.

The first problem is that the maximum valued action is going to be constantly changing as the agent moves through the environment. This is a problem because in a hierarchical/SMDP framework the agent needs to have a stable action output. That is, if the agent selects some abstract action a, that decision needs to be maintained until a terminates, despite the intermediate changes in the state brought about by the subpolicy of a.[6]

Stable action selection can be achieved by storing the selected action. For this we can use exactly the same memory circuit as was used to store the previous state (see Figure 4.3). The only difference is that the target value in this case is the output of the basal ganglia/thalamus model. The gating signal is the same, causing the stored value to be updated whenever an action terminates. Thus whenever the previous action terminates, a new one will be selected by loading the maximum valued action into this memory component. The content of the memory then drives the final output of the policy—the action that is delivered to the environment.

The above setup computes a simple greedy policy. However, as discussed in Section 2.1, RL policies also require some random exploration; otherwise the agent would just continue to follow the first decent path it finds. Exploration is accomplished in this model by adding random noise (drawn from $\mathcal{N}(0, \sigma)$) to the Q values coming from the action values component. The effect of this is akin to a soft-max policy (Equation 2.7). The probability that an action is selected is equal to the probability that the noise pushes the value of that

[6]The options framework does have an extension that allows abstract actions to be interrupted (see Section 2.2.2), meaning that the output action switches whenever a different action has a higher value in the current state. There are potential benefits to this approach (Sutton et al., 1999), and it would be interesting to explore the effect in this model. However, in this work we opted for the simpler approach of executing each action to termination.

action higher than the max value action. That is,

$$\pi(s,a) = p(\mathcal{N}(0,2\sigma) > \max_b Q(s,b) - Q(s,a)) \tag{4.6}$$

$$= \frac{1}{2} - \frac{1}{\sqrt{\pi}} \int_0^z e^{-t^2} dt \tag{4.7}$$

where $z = \frac{\max_b Q(s,b) - Q(s,a)}{2\sqrt{2}\sigma}$ (the action probabilities also need to be normalized by dividing by $\sum_a \pi(s,a)$).[7] Intuitively, the probability of selecting an action is proportional to how close that action is to the max, which is the essential function of the soft-max policy. This addition allows this component to implement all the required functions of an RL policy.

4.2.1 Important parameters

The action selection component is relatively static, and has few parameters that need to be adjusted. The main variable is the noise that drives exploration, in which the key factor is the variance of the noise—the greater the variance, the more random the agent's action selection will be. In the results described in this work we use Gaussian white noise, thus the factor being adjusted is the standard deviation of the Gaussian distribution.

When picking the noise level for a given task, the important factor is that the noise be proportional to the relative range of the Q values (not their absolute magnitude). For example, if all the Q values are close together then even small noise levels will result in very random action selection, whereas if the Q values are widely spaced then insufficient noise might result in purely greedy action selection. It is also possible to adjust the noise level over the course of learning. This could be used, for example, to mimic the common soft-max technique of decreasing the temperature parameter over time.

The other important variable in the action selection system is a scaling factor on the input to the basal ganglia. Scaling is necessary because the basal ganglia is only designed to operate over values in a certain range; specifically, the basal ganglia works best if its inputs are in the range 0.5–1.5. Therefore the Q values need to be scaled from their true values down into that range for the $\arg\max$ calculation.

Applying a constant scale to a value is easy to do in the NEF (see Equation 2.36). However, this does assume that the approximate range of Q values is known for a given task, so that the appropriate scaling constant can be determined. The basal ganglia still

[7]Note that this expression has been slightly simplified, in that it only compares the given action to the maximum value action, and not all other actions.

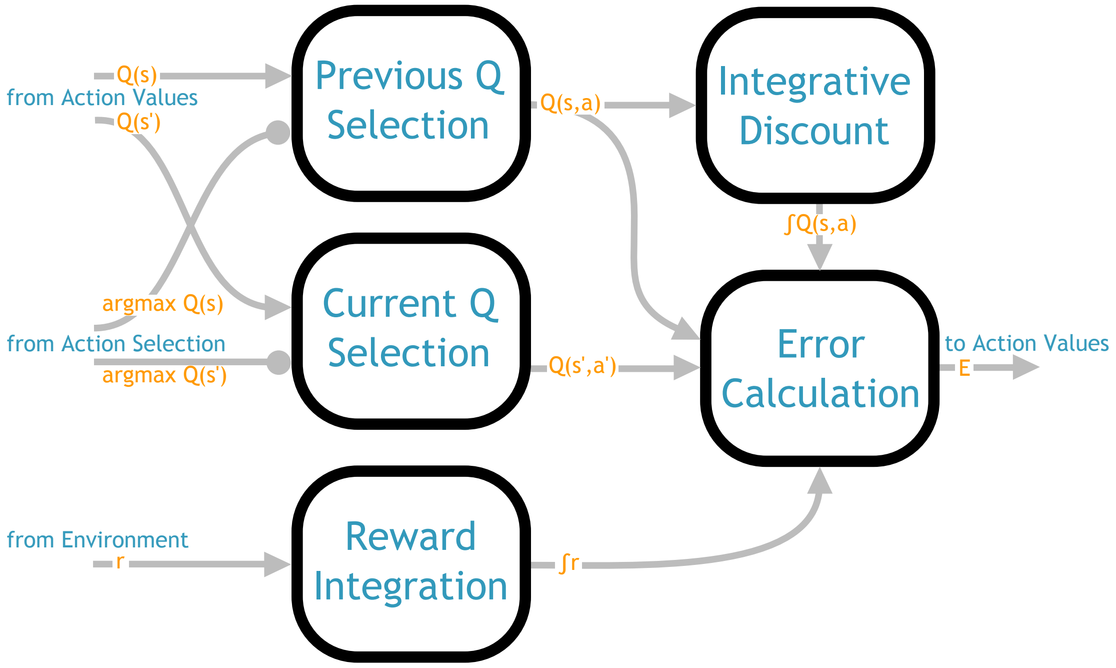


Figure 4.5: Architecture of error calculation network, implementing the SMDP TD error calculation of Equation 4.9. The selection functions are performed by the network shown in Figure 4.6, and the integration by the network shown in Figure 4.3. The final output is used as the error signal modifying the Q function representation in the action values component.

operates well for somewhat smaller or larger ranges, so the scale does not need to be finely tuned; it just needs to be in the right ballpark to scale the Q values down to a magnitude around 1. It is possible to do automatic normalization via neurons (Eliasmith and Martens, 2011), which could be added in to this model to update the scaling dynamically. However, that was not the focus of this work, and so in the results presented here we just use a constant scale.

4.3 Error calculation

The purpose of the error calculation component is to calculate the SMDP TD prediction error (see Equation 2.18):

$$\delta(s,a) = \sum_{t=0}^{\tau-1} \gamma^t r_t + \gamma^\tau Q(s',a') - Q(s,a) \tag{4.8}$$

There are four basic elements that go into this computation: the values of the current and previously selected action, the discount, and the reward. In this section we will discuss how each of those elements are computed, and how they are combined to calculate the TD error.

4.3.1 Selecting Q values

The action values for the previous and current state, $Q(s)$ and $Q(s')$ are already computed in the action values component. However, those vectors contain the values of all the actions available in a state, while what is needed for Equation 4.8 is the specific value of the selected action $Q(s,a)$ and $Q(s',a')$. Thus what is needed is to separate out the element of that vector corresponding to the selected action.

The selected action is represented by the output of the action selection component as a vector of zeros, with one for the selected action. Thus computing the inner product of that vector and $Q(s')$ would, roughly speaking, give the desired value $Q(s',a')$. However, it is difficult to do precise multiplication using the linear approximation of the decoder calculations, particularly when the multiplication involves values of very different magnitude (such as multiplying a Q value by zero). It is important that the Q values be as accurate as possible, as the TD error needs to capture subtle improvements or decreases in the action values, so multiplication is not ideal. A more accurate selection method can be found by noting that the particular case of multiplying by zero is equivalent to inhibition. However, inhibition is more accurate, because when neurons are inactive their output is exactly zero, rather than the approximate zero obtained by multiplying non-zero activities by the decoders.

Inhibition can be used to select one element of $Q(s')$ by constructing a small circuit containing n populations, where each population receives one element of $Q(s')$ as input (see Figure 4.6). This routing is accomplished via a simple linear transform. For example, to route the first element to a population the transformation matrix would be $[1, 0, 0, \ldots, 0]$; if

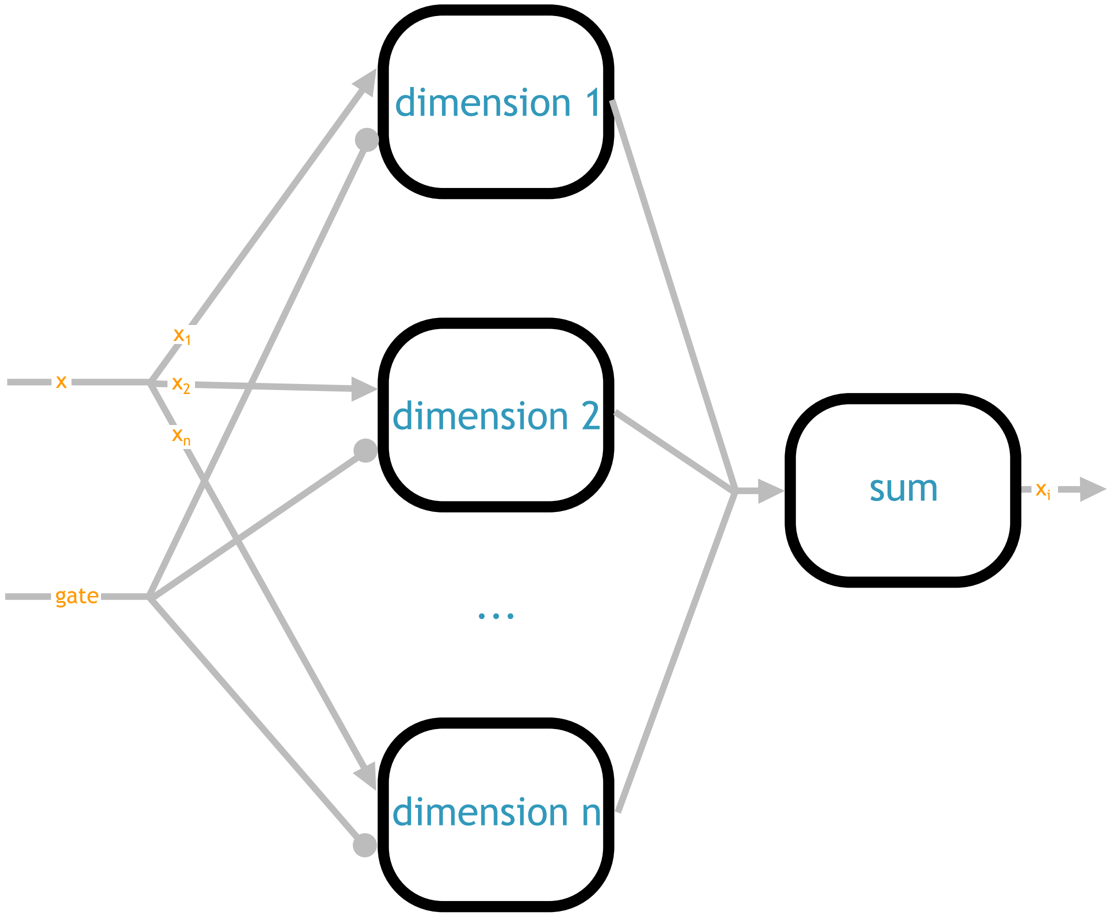


Figure 4.6: Circuit used to select one element in a vector. The vector is first divided into populations representing each of its individual elements. The gating signal is similarly divided into its individual elements, each inhibiting the non-aligned populations. Finally the output of all the individual populations is summed together, the result of which will be equal to the selected element since all other outputs are zero.

this transformation is used to calculate the connection weights as in Equation 2.36, this will result in only the specified element of the input being represented in the target population. The output of the action selection component is then connected to each population via inhibitory weights. This means that if, e.g., the first action was selected, then all the populations except the first will be inhibited. The output of the n populations is then summed in a final population. Since the output of all except the selected population is zero, the sum will be equal to the Q value of the selected action.

The error calculation also requires the value of the previous state, $Q(s,a)$. Fortunately, the previous values are already being conveniently represented by the dual training system in the action values component (see Section 4.1.2). Thus the same inhibitory selection network can be used, but using the previous Q values as input rather than the current values. Similarly, the inhibition should not be driven by the currently selected action, a', but by the action selected in the previous state, a. Again however this information is already available in the action selection component, which, as discussed, saves the previously selected action. Using that saved action as the inhibitory signal will result in $Q(s,a)$ being selected.

One might wonder if it would be more efficient to just directly store the selected Q value from the previous state, $Q(s,a)$), rather than storing a and then repeating the inhibitory selection to find $Q(s,a)$. However, this approach can lead to a positive feedback problem. Imagine the agent arrives in some state s, selects an action a, and stores the value of that action $Q(s,a)$. Suppose for simplicity's sake that a causes the agent to remain in state s. The dual training system then updates the current state Q representation based on the previous state Q representation, and suppose this causes the current state Q representation to increase. If the TD error is then calculated, this will look like a positive prediction error between the stored $Q(s,a)$ and the current $Q(s,a)$. The TD update will then be applied, increasing the value of the previous state Q representation in the dual training system, which will again cause an erroneous prediction error when the dual training occurs, and this will continue indefinitely. The TD error should reflect the actual difference between state values, not the difference between values before and after the dual training update. That is why the model uses the system described above, so that the TD error is always based on the most recent output of the Q functions, not the past output.

One final note of importance is that the $Q(s,a)$ and $Q(s',a')$ calculated by this component are based on the action selected by the agent, rather than for example the strict max Q value used in Q learning. This means that when these values are used to calculate a prediction error the model will be computing on-policy learning updates (as in the SARSA RL algorithm, Equation 2.5). This is done largely for practical reasons; the selection process is already being done once to pick an action, and the result of that process can just be reused to determine the Q values rather than having a separate max calculation. However, this also has the benefit that RL with linear function approximation has more general convergence guarantees with on-policy updates (Tsitsiklis and Van Roy, 1997).

4.3.2 Discounting

The next element of Equation 4.8 is the discount factor, γ. Expressed in continuous terms, γ is an exponentially decaying signal that is multiplied by incoming rewards across the SMDP delay period, as well as the value of the next action $Q(s', a')$ at the end of the delay period. We will discuss two different approaches to calculating this discount.

The first approach is to represent the exponentially decaying signal via a leaky integrator. This is a population of neurons connected back to itself with a scale less than one (see Section 2.3.2). This will cause the represented value to decay over time, at a rate determined by the scale. This value can then be multiplied by the incoming rewards and current state value using the technique discussed in Section 2.3.2 in order to implement Equation 4.8. However, this approach relies heavily on multiplication, which, as mentioned above, is difficult to perform accurately, particularly when the values being multiplied are of very different magnitude (as they are likely to be here, depending on the range of the Q values). Again, this inaccuracy can be problematic, because the TD error calculation often involves very slight differences in value that we do not want to disappear in the multiplicative noise. This motivated a second approach that would be less dependent on multiplication.

Before explaining the second approach it is helpful to step back and examine the purpose of the discount. TD methods rely on the notion of a state or state-action value. A Q value represents the expected rewards an agent will accumulate for selecting action a in state s, including any immediate rewards and all rewards in the future resulting from that choice. The problem is that in the general case where the agent continues moving around the environment indefinitely, the sum of all future rewards is infinite. This would lead to unhelpful Q values; if $Q(s, a) = \infty \; \forall a$, then the Q values are not useful for action selection. This is why a discount is needed—it reduces the impact of rewards in the future so that the sum has a finite value.

This is not only a problem for an abstract theoretical situation, it has practical importance for model construction. Namely, without a discount the Q values will grow indefinitely. The values need not reach infinity before they will fall outside the representational range of the model, at which point the model will fail in various ways. Thus a different way to phrase the function of the discount is that it keeps the Q values within the representational range of the model. In addition, it is important that, in doing so, the discount preserves the relative differences between the Q values, so that they remain useful for action selection (for example, a “discount” that simply thresholded all the values at some limit would not be helpful).

In the second approach to discounting we calculate the discount by integrating the value of the previous action. The advantage of this approach is that this discount factor can simply be subtracted from the TD error, rather than combined multiplicatively:[8]

$$\delta(s,a) = \sum_{t=0}^{\tau-1} r_t + Q(s',a') - Q(s,a) - \sum_{t=0}^{\tau-1} \gamma Q(s,a) \tag{4.9}$$

(we express things in a discrete form here for continuity with previous equations—in practice the summations are replaced with continuous integrals).

Clearly this form of discounting is not mathematically equivalent to the multiplicative discount. However, it captures the two basic properties of the multiplicative discount—namely, that the discount ($\sum_{t=0}^{\tau-1} \gamma Q(s,a)$) scales with time and with the magnitude of the Q values. In addition, it accomplishes the practical purpose of the discount of keeping the Q values within the representational range of the model.

For example, imagine an agent with one state and action and a fixed τ, receiving constant reward. Initially, if $Q(s,a)$ is zero, the discount will be zero, so the prediction error will just be equal to the reward. As $Q(s,a)$ gets larger, so too will the discount, which will result in a steadily decreasing prediction error. Eventually an equilibrium point will be reached; when $\sum_{t=0}^{\tau-1} r_t = \sum_{t=0}^{\tau-1} \gamma Q(s,a)$ the prediction error will be zero, and the Q values will stop growing. In a more realistic environment where the agent is moving between different states, the equilibrium point will be reached when $\sum_{t=0}^{\tau-1} r_t + Q(s,a) - Q(s',a') = \sum_{t=0}^{\tau-1} \gamma Q(s,a)$. The modeller can adjust where the equilibrium point will be by adjusting γ (this is helpful for fitting the values to the representational range of a given model). Importantly, the equilibrium point depends on the values received after selecting action a (the immediate rewards and the improvement in the state value). That is, actions that result in a better outcome will have a higher equilibrium point. Thus this discounting scheme fulfills the key requirement of preserving relative differences between actions.

The main difference between the multiplicative and integrative discount is that the integrative discount grows continuously over time. This means that if the delay period is long enough Equation 4.9 can result in a negative prediction error, even if the undiscounted

[8]In machine learning terms we can think of the discount factor as a regularization term, limiting the magnitude of the Q values.

prediction error would be positive. In some cases this can be interesting behaviour to include in the model—essentially it is an automatic penalty applied to actions that take too long to complete. However, if the modeller wants the discount to operate in the standard way, allowing it to change the magnitude of a prediction error but not its valence, this can be achieved by applying a threshold to the discount, so that it cannot grow larger than the undiscounted prediction error.

In terms of implementing this discount neurally, we have already discussed in Section 2.3.2 how to implement an integrator using the NEF. Connecting the output of the population representing $Q(s, a)$ to the integrator population with a scale of γ will result in the desired computation. The only complication here is that the integrator needs to be reset to zero every time an action terminates. This can be achieved by placing this integrator within the memory circuit used elsewhere in this model (Figure 4.3), with zero as the target value. The storage update is triggered by the same signal used to trigger the storage of the previous state in Section 4.1.2. This will cause the value in the integrator to be set to zero whenever a new action is selected, and allow it to integrate normally at other times.

There is one final technical challenge related to discounting in this model. Discounting only operates on the action values of the selected action, a, by pushing that value towards zero. This places an upper limit on the Q values, because whatever value is highest will be pushed downwards to its equilibrium point, or until another action has a higher value. This is the desired behaviour, and keeps the Q values within the representational range of the model. However, this process does not work for negative values. Imagine two values $Q(s, a_1) < Q(s, a_2) < 0$. a_2 has a higher value, so it will be selected, and the discount will push its value upwards (in the absence of any negative prediction error). However, a_1 will never be selected, because it will continue to have a lower value than a_2; a_2 will just keep being pushed upwards, while a_1 remains the same. In other words, the discount factor applies a negative feedback loop to the selection of positive values, but a positive feedback loop to the selection of negative values (note that this is true for both the multiplicative and integrative discount schemes).

Of course if a_1 is never selected there should be no prediction errors pushing it downwards either. Thus the occasional random exploration of a_1 will result in the discount being applied, which should eventually bring its value back up. However, in practice the noisiness of a neural model can cause some drift even in non-selected values, and we found that random exploration was not enough to correct this. This led to a tendency for the model to get stuck in local minima when the action values were all negative, in which it would just continue to repeatedly select the same action. Therefore we developed a neural mechanism to explicitly apply an upwards force to negative values, to fill the role the discount plays for positive values. Thus this component does not represent an important part of the model from an algorithmic perspective, but we include a description here because it is an example of the practical problems of a neural implementation.

The easiest solution would be to apply a fixed threshold to the Q values, so that they cannot go below a certain point. However, this is not the best solution, for the same reason that a fixed upper limit is not a good positive discount mechanic—it eliminates the relative differences between values. If two action values are drifting out of range we would like to correct that while still preserving their relative values as much as possible. A more complex method would be to compute a pseudo-discount for all the actions (e.g., by integrating their value); however, this would be fairly elaborate to implement. We opted instead to provide a small fixed positive bias to all the negative values. This strikes a balance between the two extremes; it will preserve the relative difference between values, but we only need to apply a fixed value rather than computing a signal for each action. Since all that this mechanism is intended to do is speed up the corrections that would normally happen through random exploration, it is not necessary for the correction to be extremely precise.

The bias can be computed by a population that constantly represents a vector, where each element of the vector is a small positive value. However, the bias should only be applied to negative values (the same effect is already being implemented by the discount for positive values). Positive values can be detected by a population that receives $Q(s)$ as input and computes the threshold signal

$$f(x) = \bigoplus_a \begin{cases} 1 & \text{if } Q(s,a) > 0 \\ 0 & \text{else} \end{cases} \tag{4.10}$$

As in the state distance thresholding used in Section 4.1.2, the neuron properties can be used to help compute the thresholding by aligning all the neuron firing thresholds to the desired function threshold. The output of this population is then used as an inhibitory signal on the populations representing the positive bias, so that the bias for any dimensions with a positive value will be set to zero (similar to Figure 4.6). The output of that population is then added to the regular TD error, meaning that any negative values will have a small positive bias applied to them, even if they were not selected. This has the desired effect of more quickly moving the agent out of local minima for negative values.

4.3.3 TD error calculation

With $Q(s, a)$, $Q(s', a')$, and the discount all computed, the only remaining calculation in Equation 4.9 is to integrate the reward and then put all of these components together.

The reward can be integrated using exactly the same circuit as was used to compute the integrative discount (that is, an integrator with an extra mechanism attached to reset the integrator to zero when a new action is selected). The only difference is that the input to the integrator is the reward from the environment in this case.

With all of the values computed, the TD error is easy to calculate. As it is a simple linear combination of the four components, it can be computed by a single population that takes as input the output of the populations representing each value, with a transform of -1 applied to $Q(s, a)$ and the discount. The output of this population then represents the TD error function described in Equation 4.9.

Note that the output of this population will be a continuous signal across the delay period, whereas we only want to update $Q(s, a)$ when the action a terminates. This can be achieved by inhibiting the above population, so that the output will be zero except when we want to apply the TD update. The timing of this signal is based on the same signal described previously for saving the selected action and resetting the discount and reward. The two are just slightly offset from each other, so that the learning update is applied just before the network is reset for the next action.

The last step is to transform the scalar output of the above population into the n-dimension error signal required by the learning rule in Equation 2.40. n is the dimensionality of the decoders to which the learning rule is being applied, which in this case is the population representing $Q(s)$. In other words, we need to compute an error for each action.

Fortunately this is relatively simple; the TD update should only be applied to the selected action, so the error should just be zero for all the non-selected actions.[9] Thus the n-dimensional error could be computed by multiplying the TD error by the vector of zeros and one output from the action selection network. However, as mentioned above, the special case of multiplying by zero can be implemented more accurately using inhibition. We create an n-dimensional population where each element represents the same TD error value, and then inhibit all the dimensions corresponding to the non-selected actions. What we want to accomplish here is essentially the same as how $Q(s, a)$ and $Q(s', a')$ were computed. There we were taking $Q(s)$, inhibiting all but the selected action, and then summing the result.

[9]One could imagine a non-zero error for multiple dimensions in the continuous action case, where the agent selects some weighting over the basis actions as its output. This possibility is discussed in Section 6.2.3.

Here we just omit the summation at the end, so that we are left with an n-dimensional vector with all zeros except for the selected action. This value is the final output from this network; using it as the error signal in the decoder learning rule on the output of the population representing $Q(s)$ will implement the TD update of Equation 2.8.

4.3.4 Important parameters

The important tunable parameter in the error calculation component is the discount factor, γ. As mentioned, the key role of the discount factor in this model is to keep the Q values within the representational range of the model. The higher the value of γ, the lower the equilibrium point of the Q values will be. Thus the particular setting of γ will depend on where the modeller wants the Q values to settle for a given task. This is fairly arbitrary, so in general the modeller can just decide what the range of the Q values should be and then adjust γ accordingly.

There is one consideration that slightly complicates this process. In addition to affecting the maximum Q values, the discount also affects the slope of the Q value gradient outwards from the rewarded states. For higher discount factors the Q values will decrease more rapidly as the agent moves away from the rewarded state, as that decrease is driven by the discount. This can be an important consideration, as when the Q values become too small they cannot be accurately distinguished by the action selection system (we will see an example of this in Section 5.3).

As a result, in general it is desirable to keep γ small in order to propagate reward information as far as possible throughout the state space. Thus when trying to adjust the range of Q values, it may be better to lower the reward values from the environment (e.g., by applying a constant scale < 1), rather than increasing γ. The effect will be the same, lowering the equilibrium point, but this will create a shallower slope in the reward gradient.

4.4 Hierarchical composition

At this point all the components of a flat reinforcement learning model are implemented. As shown in Figure 4.1, this system implements the full RL loop. It takes a state from the environment, determines the values of the available actions in that state, picks an action based on the Q values, waits until the action terminates, observes the resulting state and any rewards from the environment, computes the SMDP TD error, updates the Q values accordingly, and repeats it all in the next state.

When extending this basic model to a hierarchical setting, we can think of this cycle as the operation of a single layer in the hierarchy. The only difference is that the delay until action termination represents the activity of a lower level, rather than a delay imposed by the environment. But from the perspective of the rest of the system, learning action values, selecting actions, and computing the TD error all proceeds in the same way. Thus all that remains is to show how these basic elements can be composed into a multi-level hierarchical system.

4.4.1 Hierarchical structure

In the design of this system we have tried to make the hierarchy as modular as possible. The goal is to be able to construct the hierarchy out of the basic pieces described above without modifying the internal operation of those pieces. There are a number of advantages to this approach. First, from a practical perspective it simplifies the design and use of the system; rather than redesigning the model whenever a new layer is added, we can just repeat the same basic structure. In addition, each interaction between layers introduces more potential points of failure, and also makes those failures more difficult to understand. Thus by minimizing the interactions between layers we can make the system more robust. This is related to another advantage of the modular approach, namely scaling. When there are minimal interactions between layers, this means that there is minimal overhead as more layers are added. If the elements of two layers are densely interconnected then the number of connections in the model would increase polynomially as new layers are added, rather than linearly. Yet another advantage of this approach is biological plausibility. Just as all the features described above (ease of implementation, robustness, and scaling) are useful from a computational perspective, they are also useful in biological systems. Thus the brain also seems to have a tendency to construct more complex systems out of simple repeated elements, and we follow that design in the structure of this model (this is discussed in more detail in Section 4.5).

In practice what this modularity means is that the only interaction between layers is via the normal inputs and outputs of the RL process. That is, the layers interact via the same inputs and outputs with which the flat system interacts with the environment (state and reward inputs, and action output)—there are no new connections between the internal components of different layers.

In other words, from the perspective of a single layer, the layers above and below it

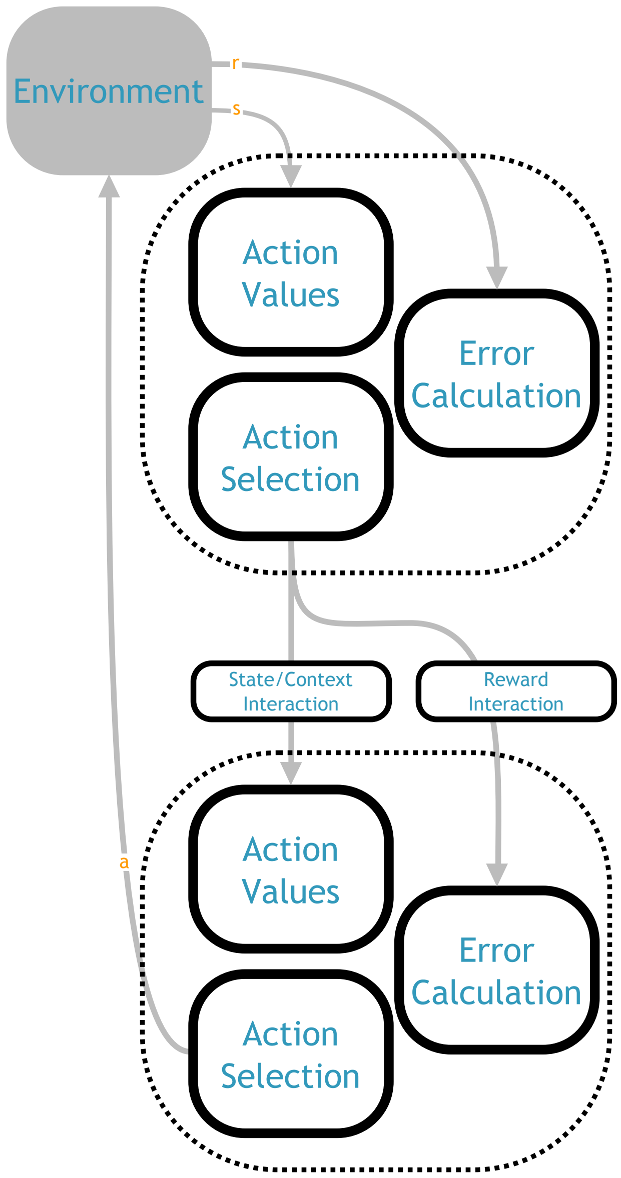


Figure 4.7: Hierarchical composition of the basic SMDP architecture (from Figure 4.1). The output of the higher level can modify either the state input to the action values component of the lower level (a state or context interaction), or the reward input to the error calculation component (reward interaction). Shown here with a two layer hierarchy, but this same approach can be repeated to the desired depth.

are just a part of the environmental black box. The layer sends out some action vector, which through some unknown mechanism (the operation of other layers) results in some rewards being received as input and a new state when the action terminates. The individual layers have no “knowledge” that they are embedded in a hierarchy; each of them is just working independently on their own SMDP reinforcement learning problems, trying to pick whichever actions will maximize their reward. Only from an outside perspective can it be seen that the RL problems of the different layers are connected to one another.

For example, imagine a system trying to navigate around a house, where one layer of the hierarchy picks which room to go to and a lower level takes care of navigating to the target room. The overall task is to move to the kitchen, which requires the agent to move through the dining room. The top level does not need to know that when it selects the “go to the dining room” action a lower level is going to select “go forward” and “turn left” actions that actually bring it to the dining room. The top level just learns that when it outputs the “go to the dining room” vector, after some delay period it ends up in the dining room state. When it arrives in the dining room it can evaluate whether that choice was a good one or not, based just on the elapsed time and the value of the resulting state. Thus the values of the different actions, and the process for learning those values, is a flat SMDP problem, with no reference to a hierarchy. Similarly, the lower level does not need to know that there is some higher level process trying to navigate to the kitchen. It just receives some input which it has learned is associated with going to the dining room, and it goes about completing that task (perhaps with the aid of still lower levels).

To reiterate then, the important constraint on the hierarchical structure of this model is that the only interaction between layers is through the standard inputs (states and rewards) and outputs (actions). However, that still leaves open a wide range of possibilities as to what those inputs and outputs look like, and this model does not prescribe a specific implementation. As discussed previously, the goal here is to describe a generic model that can be applied to many different tasks. Different tasks will require different types of hierarchical interaction, depending on the hierarchical structure of the task. In the results section we will give examples of specific tasks and discuss how the interactions were implemented in those cases. Here we will speak more generally about the different styles of interaction and the functionality they afford.

Note that in this work we also take the hierarchical interaction to be defined by the modeller. The question of how these structures could be learned automatically is an important and interesting one; we will discuss work in this field in Section 6.2.6. However, this is very much an open problem in the computational field of HRL, not to mention neural implementations. The question of how to implement a *given* hierarchical structure is a precursor to any question of learning that structure, so it is on the former that we focus our efforts here.

We divide the ways in which layers can interact into three different categories. There are only two inputs to a layer, the reward and the state, thus hierarchical interactions must flow over one of those two channels. However, we divide the latter into two different categories—"context" and "state" interactions.

Context interaction

In a context interaction the higher layer adds some new state information to the input of the lower layer. It is termed a context interaction because this extra state can be thought of as a context signal that modifies how the lower level responds to the normal environmental state. For example, in the house navigating task suppose the environmental state is the GPS location of the agent. The goal of navigating to the dining room can then be represented by appending a vector to the state input of the lower level. The content of that vector will be task dependent; it could be the GPS location of the dining room, or a random vector label. From the perspective of the lower level it does not matter how the vector was generated, it is all the same black box state input.

Concatenation is just a linear operation, and so can be implemented by a single population that receives both the environmental state and the output of the higher level as input, concatenates them, and then connects its output to the state input of the lower layer. The lower level then has as its input both the normal state s and an indication of the goal, or context, c. Recall that the lower level is not aware that it is part of a hierarchy, and so is unaware that its input is a composition of these two elements. It just sees some state input $\tilde{s}$, and it tries to learn the values of its available actions in each state.

The important feature that the context interaction allows is that the same environmental state s is now represented at multiple different points in the concatenated state space of $S \times C$. This means that the agent can learn different action values for the same environmental state, depending on the context. That is, the agent can learn one value for $Q(s \oplus c_1, a)$ and another for $Q(s \oplus c_2, a)$. In the case of the house navigating agent, c_1 might represent "go to the dining room" and c_2 might represent "go to the bathroom", thus the agent can learn that the same action ("turn left") in the same location can have different values depending on the context.

Since we are using function approximation, there will be some relation between $Q(s \oplus c_1)$ and $Q(s \oplus c_2)$. For example, if c_1 and c_2 are very similar, then changing $Q(s \oplus c_1)$ will also affect $Q(s \oplus c_2)$. This is the same issue already discussed in reference to state encoding

(Section 4.1.1), and the same points apply here. For example, the degree of interaction can be adjusted by changing the sparsity of the $\tilde{s}$ state encoding.

To summarize, the context interaction allows the lower level to learn different sets of Q values, and thereby different policies, and it allows the high level to change which policy is being followed by picking different output actions. Importantly, from the perspective of both layers they just have a set of Q values that assign a value to each action in the current state, and they can learn those values using the standard SMDP TD error. Thus each layer operates independently in exactly the way described in the previous sections, but because they are connected via this context interaction the end result will be a hierarchical learning system.

State interaction

In state interactions the higher level modifies the environmental state for the lower level, rather than appending new information. The primary use case for this is what is known as "state abstraction". State abstraction is based on the idea that not all aspects of the state may be relevant for a given task. Thus when the agent learns Q values for the whole state space, large amounts of effort are being wasted learning the value of the Q function over different states that represent the same effective point in the problem space. If the irrelevant aspects of the state can be eliminated ahead of time then the size of the state space becomes exponentially smaller, making the learning much more efficient.

Recognizing this, state abstraction has been of interest in RL almost since its inception; however, it has had particular traction in the field of HRL (e.g., Dietterich, 2000; Andre and Russell, 2002; Provost et al., 2007; Barto et al., 2013). When an overall task is broken up into subtasks, it is often the case that aspects of the state that are relevant for the overall task are not relevant in a particular subtask. For example, the overall task of making breakfast might require as state the location of many different cooking implements. However, if that task is broken up into subtasks for making cereal, toast, and coffee, then each one of those subtasks only requires a subset of the available kitchen tools. Thus while it may be difficult to find a useful state abstraction for the overall task, it can be more feasible to find state abstractions in each of the different subtasks.

This kind of operation is enabled by the "state interaction" hierarchical structure. For example, when the high level outputs a vector a_1 representing the "make cereal" action, that vector can be used as a gating signal on the environmental state to block out the features irrelevant to that subtask. This can be implemented via, e.g., the same inhibitory gating scheme used to block different parts of $Q(s)$ based on the selected action (Figure 4.6).

The output of that gating network then represents a new state $\tilde{s}$ that belongs to a subspace $\tilde{S} \subseteq S$. The lower level then only needs to learn a Q function over the smaller state space of $\tilde{S}$.

Note that state interaction need not be limited to state abstraction, it could map s to an arbitrary new state space. For example, instead of simply eliminating irrelevant aspects of the state space, the interaction could be defined as a type of context-dependent PCA that rotates the state into a form more useful for learning in the lower level task. However, state abstraction is the simplest and most lightweight type of state interaction, and its usage has been explored in several other examples of HRL. Thus it is likely to be the most commonly applicable type of state interaction in this model.

Reward interaction

Reward interaction involves the higher level modifying the reward input of the lower level. The primary use case of this is to implement theories of pseudoreward, for example as used in both options and MAXQ (see Sections 2.2.2 and 2.2.4). Recall that pseudoreward is reward administered for completing a subtask. Without pseudoreward, a layer can only learn to choose actions that maximize the external reward associated with an overall task. If the modeller wants the layer to learn to achieve a subgoal independent of the overall task, then there needs to be pseudoreward associated with that subgoal in order to guide the RL process.

For example, in the house navigating agent recall that the desired functional architecture is that the lower level navigates to the target set by the high level, while the high level picks targets to complete the overall task. However, if the low level only receives environmental rewards, then it will learn to select actions that complete the overall task, which may or may not involve going to the target set by the high level. In order for the low level to learn a policy that goes to the specified target, it needs to be rewarded for reaching that target.

More generally, the low level should be rewarded for completing the goal set by the high level. Again it is difficult to specify the specific implementation of this mechanism in a task-independent fashion, as the notion of a goal will differ from task to task. However, the most common type of goal is likely to involve reaching some target state s_0 (as in the above example). In this case the output a of the high level defines the target state s_0 (i.e., there is some one-to-one mapping $A \mapsto S$). The pseudoreward signal can then be computed via, e.g.,

$$r_t = \begin{cases} 1 & \text{if } \|s - s_0\| < \theta \\ 0 & \text{if } \|s - s_0\| \geq \theta \end{cases} \tag{4.11}$$

Note that this function has the same form as the one computed when calculating the state distance in the dual training system (Equation 4.4). Thus the same circuit used to compute that function can be reused here to compute the pseudoreward.

The pseudoreward signal could completely replace the environmental reward, or it could be added to the environmental reward with some relative weighting. The former will implement the pure subgoal learning of options/MAXQ, while the latter is more akin to reward shaping. Reward shaping involves a higher level that is used to help train a lower level by providing "hints" via the reward signal (e.g., Randlov and Alstrom, 1998). The pure subgoal reward can be seen as an extreme case of reward shaping. Again, either approach is compatible with this framework, depending on the task.

4.4.2 Relation to other HRL approaches

It is helpful to compare the hierarchical mechanisms of this model to the computational theories described in Section 2.2, as another way of elucidating the features of this model. Firstly, it is important to note that this model is not an exact recreation of any of those theories—the process of neural implementation requires many adaptations and modifications, as we have seen in previous sections. However, the model does stay quite close to these systems functionally, in particular to the options framework.[10]

The key aspect of the options framework is that it redefines actions as policy objects called options. That is, when the agent selects an action, what it is doing is putting a particular policy into place, so that future decisions will be made according to that policy. This can be simulated in the model through the mechanism of context interactions. When the agent selects an option, that can be modelled by appending a vector to the state of a lower level. This is like loading a particular set of Q values—a different mapping from actions to values, depending on the context. Thus, effectively this action has put a new policy into place that will guide future action selection, which is the basic mechanism of the options framework.

[10]We will focus primarily on the options and MAXQ frameworks here, as they lie closest to this approach. HAMs are linked to this work by their use of SMDPs to encapsulate temporal abstractions, but the key feature of HAMs—the reduction of the task to choice points in finite state machines—does not really have close parallels in this work.

The SMDP TD update mechanism used in this model is also essentially the same one employed in the options framework (compare Equation 4.9 and the formula in Section 3.2 of Sutton et al. 1999). The options framework employs the standard multiplicative discount, a Q-learning rather than SARSA style update, and is formulated for discrete time and space, but the overall form of the SMDP TD learning update is the same; the Q values from when the abstract action was selected and when the action terminates are subtracted, plus the reward, and that prediction error is used to update the Q values from the beginning of the action period. This is an important feature that separates this approach from the MAXQ framework, where the learning updates are based on the hierarchical decomposition of the value function.

One important way in which this model differs from the options framework, and is more similar to MAXQ, is that the actions are placed into a fixed hierarchical structure. In the options framework an option's policy is defined over all the available actions in a given state. In this model the actions are spread across different layers, and each policy is only defined across the actions of that layer. For example, the lower level might contain all the primitive actions for interacting with the environment, and the higher level could contain the abstract actions (representing different policies for the lower level); this would mean that the low level policy was only defined over primitive actions, and the higher level policy was only defined over abstract actions—it would not be possible for the high level to select a primitive action.

Of course the hierarchical structure could be defined differently, for example including all the abstract and primitive actions at each level. But even the concept of different levels is not a feature of the options framework. In the options framework there are just a pool of options available in a given state. After one is selected the same pool is still there, there is no notion of moving through some pre-existing hierarchical structure. For example, the options framework allows for an infinite recursion of option selection; if option o is selected in state s, the policy for o could also specify that option o be selected in state s, and so on. This would not be possible in this model or MAXQ, because when o is selected that means moving down one layer in a given hierarchical structure, and there are a finite number of layers. We will discuss more in Section 6.1.3 why the decision was made to modify this aspect of the options framework; for now we just highlight it as an important difference.

It should also be noted that, as discussed above, the hierarchical structure we have defined here allows for a lot of flexibility. A given model could be pushed more in the direction of options or MAXQ depending on the choices of the modeller. For example, the use of context interaction is more options-like, as described above, while state interactions are more MAXQ-like (state abstraction is a key feature and motivation in the MAXQ framework Dietterich, 2000). In addition, depending on how the reward interactions are constructed the results of this system could be hierarchically (MAXQ) or recursively (options) optimal.

4.4.3 Summary

Context and state interactions are based on changing the input of the Q function. The former acts by appending new information, allowing the agent to select different actions in the same environmental state. In other words, context interaction increases the size of the state space in order to increase the decision making power. State interaction is most likely used for implementing state abstraction, wherein irrelevant aspects of the state space are removed. This is the inverse of the context effect; the state space is made smaller, leading to a decrease in the complexity of the policy. Reward interaction acts differently, by changing the target of the Q function. This is key to the notion of subgoals, where the aim is for the layer to learn to achieve some target that is different than the overall task goal.

Although we have presented the different interaction types independently, in practice useful hierarchical structures often combine all three. For example, if state interaction is combined with context interaction, then the efficiency gains of the state abstraction can help balance the state space enlargement of the added context information. And it is often the chunking of the overall task into simpler subgoals, made possible by pseudoreward, that motivates the added context information, and allows for the simplifications of state abstraction.

The important feature of all three interaction types is that they operate only through the normal RL inputs and outputs. All that changes are the vector values of those signals, based on relatively simple transformations as described above. There are no changes made to, e.g., the internal action selection or TD error calculation of the components. This makes it easy to define different hierarchical interactions for different tasks, as it is not necessary to make changes to the fundamental implementation of the SMDP RL mechanisms.

4.5 Neuroanatomical mapping

In previous sections we discussed how the various model components can be implemented via the low-level features of the brain, namely neural activity and synaptic connections. Now we turn to how this model can be implemented by the macro-level structures of the brain. That is, where do the various components of this model lie physically in the brain? The majority of empirical evidence lies at this macro-level, thus such a mapping is critical

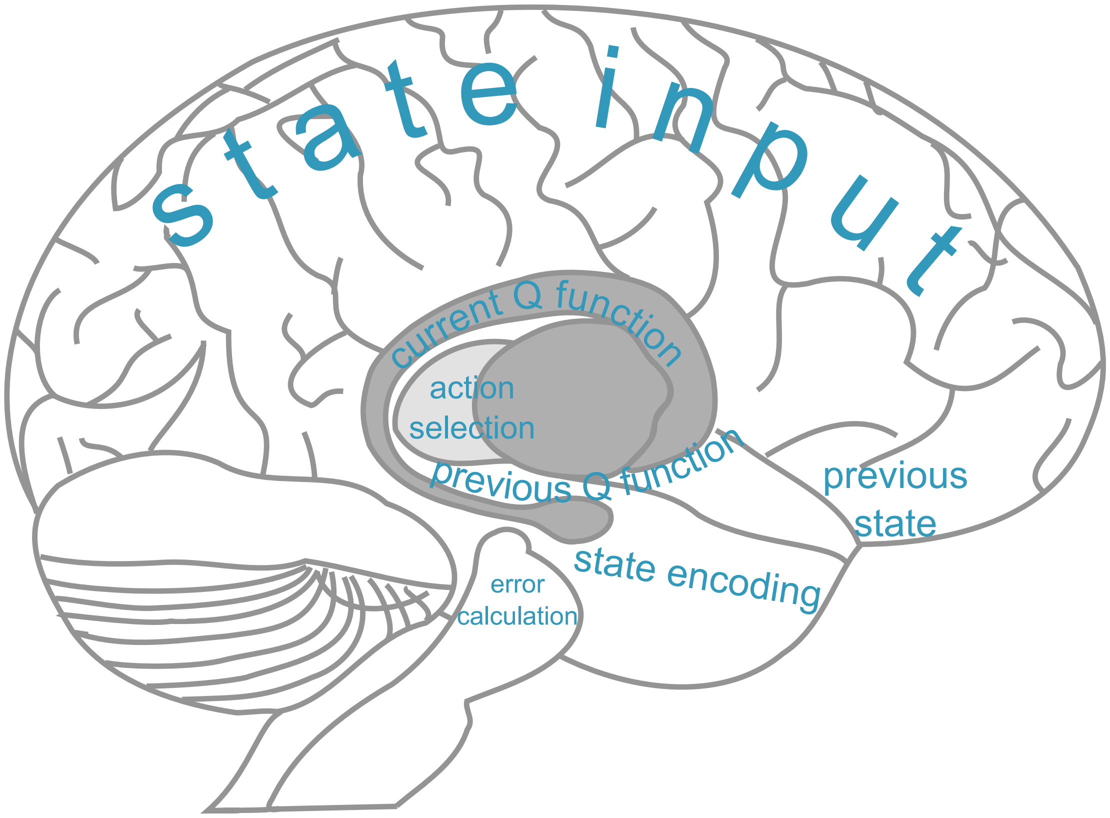


Figure 4.8: Neuroanatomical mapping of the model. State input could be taken from almost any brain region, depending on the task (hippocampus may provide a useful state encoding in many cases). Q values are represented in the striatum (dark grey), divided into dorsal and ventral for the current and previous Q functions, respectively. Orbitofrontal cortex provides the input to the previous Q function. Action selection is performed by basal ganglia/thalamus (light grey). The error signals used to update the Q functions are output from the ventral tegmental area/substantia nigra. *Brain image adapted from image by John Henkel (Wikimedia Commons/Public Domain).*

in connecting this work to experimental data. This mapping also helps to establish the biological plausibility of the model, by demonstrating that the components and communication in this model are supported by the neuroanatomy.

It is important to note that this section is largely a review of previous work. Extensive effort has gone into the search for neurophysiological correlates of RL processes (see Dayan and Niv 2008; Niv 2009; Botvinick et al. 2009 for reviews). The main purpose of this section is to summarize those results, and connect the components of this model to that data. In other words, our purpose here is not to propose a novel neuroanatomical mapping; rather, it is primarily to show that the model we have developed is consistent with the existing proposals.

The structure of this section follows the description of the model, focusing on action values, action selection, error calculation, and hierarchical composition in turn. A coarse overview of the mapping is shown in Figure 4.8.

4.5.1 Action values

The basic feature of this component is the neural representation of the Q function. Thus the question is, where are action values represented in the brain? Since action selection is performed by the basal ganglia in our model, it would be expected that the action values be represented in the major input channel of the basal ganglia, the striatum. And, indeed, there is strong evidence that this is the case.

For example, Samejima et al. (2005) recorded from 504 neurons in the dorsolateral striatum of macaque monkeys. The task was a simple two-choice RL task; the monkey holds a lever, and then when prompted moves it to either the left or right, and receives some corresponding reward. The experimenters then manipulated the probability of reward associated with moving left or right in different blocks of trials.

They found that 142 of the recorded neurons showed some responsiveness to the task, and of those, half showed a correlation with the reward probabilities of the actions. For example, a "left neuron" would fire more when the left action had a high probability of reward and less when the left action had a low probability.

The challenge in this investigation is to distinguish action values from action selection. That is, does the firing of the "left neuron" represent the value of left, or does it just indicate that the monkey has decided to move left? Samejima et al. (2005) showed that these neurons were actually representing Q values by manipulating the reward probabilities independently of the selected action. For example, in one block of trials right would be rewarded with 90% probability and left with 10%. In the next block, right would again be rewarded with 90% and left with 50%. In either case the selected action is predominantly right, and yet Samejima et al. showed that the "left" neurons still showed a differential response in the two trials. This can also be demonstrated by comparing one block where left is rewarded 50% and right is rewarded 90% versus another where left is rewarded 50% and right 10%. In this case the value of the left action is constant, but action selection is reversed in the two trials. Samejima et al. (2005) showed that the activity of the "left neurons" was constant in these two conditions, again indicating that they encode action values, not action selection. Overall, one third of the task-sensitive neurons showed this Q value encoding, which constituted 60% of the neurons that showed any correlation with reward probabilities.

In addition, Samejima et al. (2005) found that the activity of the Q value neurons followed the time course of the Q values in a simple TD RL model trained using the same reward history as the monkey. This suggests that not only do the neurons represent Q values, those values are learned in a similar manner to that predicted by the TD approach. A similar result was found by Pessiglione et al. (2006) in a human fMRI experiment, but in the ventral rather than dorsal striatum. This type of model-matching evidence is not as strong, as the correlation of observed signals with an action-value model is only suggestive of the fact that the observed signal represents action values (as opposed to, e.g., state values). However, it is more evidence consistent with the hypothesis that the action values component is located in the striatum.

It is also important to note that the striatum receives very widespread afferent projections. This is a desirable feature of any action values component, as the afferent information in this case represents the state. The state in a decision making task could involve visual areas, somatosensory, motor, linguistic, working memory, and so on. Thus it is important that all that information could potentially be available as the input to the action value function. The striatum fulfils that requirement, further supporting the proposed location of this component.

Place cells

When discussing the action values component, we pointed out that it is often useful to have a distributed, sparse, and local state representation, using Gaussian radial basis functions as an example (see Section 4.1.1). The state representation is a task-specific feature, and so we will not go in depth into it here. However, it is interesting to mention that this kind of representation is exactly that provided by hippocampal "place cells" (O'Keefe and Dostrovsky, 1971). These neurons were discovered in rodent spatial navigation tasks, where it was noticed that certain neurons would fire whenever the rat was in some specific location in the environment, firing more rapidly the closer the rat was to the centre of the place cell field. Although the original discovery of place cells was based on spatial representations, later work has shown that the hippocampus provides the same style of sparse encoding across non-spatial state information as well (Eichenbaum et al., 1999). The hippocampus also projects to both the dorsal and ventral striatum (Groenewegen et al., 1987). Thus the hippocampus represents a plausible candidate for where one might find the kinds of state encoding described in Section 4.1.1.

Note that we do not suggest that all state information in this model originates in the hippocampus. Different tasks could require many different inputs, from various locations throughout the brain. Rather, we just intend this to show that a particular style of state representation which we pointed out as useful from a computational perspective is in fact available in the brain.

Dual training

Another important feature of the action values component is the dual training system (Section 4.1.2). This is a novel feature of this model, and so not something that has been directly investigated experimentally. Thus we must search for evidence of this component indirectly, looking for suggestions of its plausibility.

The first issue to be addressed is where the previous state information is stored. One suggestion comes from the work of Schultz et al. (2000). They review a collection of previous results based on a delayed response task. A monkey is presented with a cue that indicates the task for that trial. After a delay period, it then selects an action based on that task. Then, after another delay period, it is rewarded if it picked the correct action. Under the interpretation of the dual training system, this would mean that the monkey needs to retain information about those previous states across the delay periods, so that the values of the Q functions in those states can be updated once the next state is reached.

Schultz et al. (2000) found that neurons in the orbitofrontal cortex (OFC) consistently showed sustained activity following each state change (i.e., spanning the delay period). Importantly, this OFC activity was only, or predominantly, present when the monkey was in a rewarded, reinforcement learning environment. In a passive task where the monkey did not have to make any decisions to get a reward this persistent OFC activity was not present. This provides further evidence that this OFC activity is not just a generic memory system, but memory specifically tied to the reinforcement learning process.

The next key aspect of the dual training system is the internally generated error signal based on the difference between the output of the two Q value functions. This requires a particular form of connectivity, where the output of the two functions is combined and then fed back to just one of the functions. Interestingly, this is exactly the structure of the connectivity between ventral and dorsal striatum via the dopaminergic nuclei, reviewed in Joel et al. (2002). They were discussing this in the context of an actor-critic architecture, pointing out that it was problematic for some models that map the actor and critic onto the dorsal and ventral striatum. But if we interpret the same data from the perspective of the dual training system, it fits quite well.

First of all, the dopaminergic nuclei (ventral tegmental area and substantia nigra) have long been associated with error signals in reinforcement learning. This will be discussed in more detail in Section 4.5.3, for now we just point out that this makes it a good candidate for the production of the error signal in the dual training system. The interesting observation of Joel et al. (2002) is that there is an asymmetry in the connections between the dorsal/ventral striatum and the dopaminergic nuclei. The ventral striatum projects broadly throughout the dopaminergic nuclei, while the dorsal striatum only projects to a subset of the dopamine neurons. Both receive projections back from the dopamine neurons, but the result of this connectivity pattern is that only the dopamine neurons projecting to the dorsal striatum receive input from both ventral and dorsal striatum; in other words, those neurons could conceivably compute the difference between the dorsal and ventral output, and project that difference back to the dorsal striatum. Another important point is that the OFC projects to ventral, but not dorsal, striatum. Thus the previous state information would be available to the ventral striatum.

Together, this data supports the following mapping of the dual training system. We have already proposed that the striatum is the basis of the value function representation; now we divide that into dorsal and ventral striatum, corresponding to the current and previous Q value components of the dual training system, respectively. The previous state information is stored in the OFC, which projects to the ventral striatum (the previous Q function). The dorsal and ventral striatum then both project to the dopaminergic nuclei, where the difference is computed and fed back to the dorsal striatum to update the current Q function.

It is important to emphasize that this section of the neuroanatomical mapping is quite speculative. We have assembled various different results into a coherent story, but none of these results were intended to address the proposed system. Although the mapping we have established here is plausible, it is entirely possible that new data could invalidate this proposal, or create a more convincing mapping elsewhere. We will discuss in Section 6.3 some of the specific predictions that arise from the dual training system, and how they could be used to experimentally investigate the mapping we propose here.

Action termination signal

One final question in the action values component is the origin of the action termination signal. This signal is used to trigger the transfer of the current state into the previous state memory (see Section 4.1.2), and is also used in subsequent components to control, e.g., the resetting of the integrated reward.

Interesting data in this regard comes from a study by Fujii and Graybiel (2003). They gave monkeys a task in which they had to make a sequence of eye movements in order to receive reward. Recording from neurons in the PFC, they observed a burst of activity when the sequence was complete. This burst was not related to reward activity, as delaying the reward, changing its value, or omitting it entirely did not affect the signal. The signal was also robust to changes in the properties of the sequence, such as changing the delay between saccades or the number of saccades; this indicates that it is not a simple timing signal, but actually indicates the end of the sequence.

Interestingly, if the monkey was given an external cue indicating the end of the sequence then the self-generated signal was reduced. This suggests that the monkey is able to make use of other termination signals if they are available, and further reinforces the role of this neural activity as a termination signal.

In summary, it appears that the brain does generate signals marking the termination of temporally extended actions. These signals could then fulfil the role required by the action value and error calculation components in this model.

4.5.2 Action selection

The neuroanatomical mapping of the action selection component is relatively straightforward, as it consists largely of the basal ganglia model of Gurney et al. (2001) and Stewart et al. (2010). We have discussed how this model maps onto the internal neuroanatomical structure of the basal ganglia and thalamus in Section 2.3.4, and this is presented in much greater detail in Gurney et al. (2001). The only remaining issue is how the inputs and outputs of this component connect to the rest of the model.

The only input to the action selection component is the action values. As mentioned, the striatum is the primary input channel for the basal ganglia, thus given the mapping of the previous section the action values are readily available to the action selection component.

The thalamic output goes to two places. One projection goes towards generating the action vector for the environment. This aspect we do not take to be a realistic part of the model that has a neuroanatomical mapping. The action generation circuit we discuss in Section 4.2 is simply a placeholder for the complex mechanisms in the brain that transform thalamic output into an action, which we do not include as part of this model. However, see Eliasmith et al. (2012) for an example of work that does link the basal ganglia to a model of motor output, with a corresponding neuroanatomical mapping. Thus the model we present here is consistent with such a connection, but it is not implemented.

The other target for the thalamic output is the error calculation network, where it is used to gate the error calculation according to the selected action. Evidence for this interaction comes from the work of Parsons et al. (2007), recording dopaminergic input in the ventral striatum. Recall that the ventral striatum contains the previous state Q function, which is where we would expect the error calculation component to project. Parsons et al. (2007) found that although the dopamine input in the ventral striatum comes from the ventral tegmental area, it could be modulated by stimulation administered in the thalamus. Although it is difficult to establish the actual function of that modulation, it does demonstrate that the thalamic output interacts with the error signal calculation, as proposed in this model.

4.5.3 Error calculation

The basic function of this third component is to compute the TD prediction error. This is one of the most well-studied neural correlates of reinforcement learning; work by Schultz (1998) showing that dopamine neuron activity corresponded to reward prediction error (rather than just reward, as was previously thought) was what sparked much of the interest in applying computational theories of reinforcement learning to the brain. This result has since been replicated many times, including in humans (e.g., O'Doherty et al., 2004, see Niv, 2009 for a review).

The two main nuclei of dopamine neurons are the ventral tegmental area (VTA) and substantia nigra pars compacta (SNc). In general, the former targets the ventral striatum while the latter targets the dorsal striatum. Given the mapping of the previous state Q representation onto the ventral striatum, we would therefore expect the TD error signal of the model to come from the VTA.

More recent studies have even attempted to differentiate the particular type of prediction error observed in the brain. The two main hypotheses are the action-value based errors (Q learning and SARSA) and the actor-critic architecture (based on the state values). That is, the error in the first case is based on $Q(s', a') - Q(s, a)$, and $V(s') - V(s)$ in the actor critic case. Morris et al. (2006) attempted to differentiate these two possibilities using a task where monkeys were presented with two stimuli, each with its own reward probability, and had to pick one. The monkey then receives a reward, and Morris et al. can analyze whether the resulting prediction error observation correlates with the state or action value. For example, imagine the monkey is in a state with one high value choice and one low value choice, picks the low value option, and gets no reward. The actor-critic hypothesis would predict a large negative prediction error in this case, as the monkey was in a relatively high value state and ended up with no reward. SARSA would predict a small prediction error, because the monkey picked an option with a low estimated value and that estimate was proven correct. Morris et al. (2006) find that the observed prediction errors correlate with the action value approach. This is replicated in work by Roesch et al. (2007) using rodents, although they found a mixture of on-policy and off-policy errors. Together these studies give relatively strong support to the proposal that the error signal used in this model is carried by the dopaminergic neurons.

Inputs and outputs

There are three inputs to the error calculation component: action values, the identity of the selected action, and reward. We have already discussed in the action value and action selection mapping that their output is connected to the dopaminergic nuclei, again supporting the plausibility of the overall mapping. The only remaining input is reward.

In this model the reward signal is just a direct input from the environment. Obviously real brains do not receive a direct reward signal; they receive some sensory data as input, which is then translated through internal mechanisms into reward. This is why money can be used as a reward in human reinforcement learning experiments, even though it has no direct stimulative value. These mechanisms are not included in this model; as with the action output, the reward circuitry in this model is just a placeholder for complex processes occurring elsewhere in the brain. Thus we do not include these components as part of the neuroanatomical mapping in this model. However, Lammel et al. (2012) found that they could simulate reward or punishment by optogenetically stimulating different inputs to the VTA, supporting the idea that the VTA does have an input channel for reward (wherever it arises from), as suggested in this model.

The output of the error calculation component projects to the action values component, where it is used to modify the previous state Q function. Given the mapping so far, this implies that the dopamine signal should project to the (ventral) striatum, and modulate the learning of synaptic weights there. This is another well studied feature of the reinforcement learning circuitry. Evidence for dopaminergic modulation of synaptic plasticity in the striatum was first demonstrated by Wickens et al. (1996), and has since been replicated in various paradigms, including human experiments (Reynolds et al., 2001; Pessiglione et al., 2006). Thus there is good evidence that this interaction between the error calculation and action value component is consistent with the neurophysiology.

Discounting

There are two final features to discuss related to error calculation in this model. Both are related to discounting, namely the integrative discount and the positive bias mechanism. As was the case for the dual training system, these are novel features of this model and therefore have not been directly investigated by any experimental work. Therefore we must adopt a similar speculative approach, searching for data that could be consistent with these hypotheses.

With respect to the integrative discount, what immediately appears relevant is data investigating how people estimate the value of future rewards (Critchfield and Kollins, 2001). For example, subjects are asked questions like "would you like \$10 now or \$100 in 6 months?", and their response can be used to estimate the discount they apply to a 6 month delay. Some have suggested that the discount factor in RL could be used to explain these results, which has led to debate into whether neural reinforcement learning models should use, e.g., a hyperbolic or exponential discount function (Dayan and Niv, 2008).

However, these results are not as relevant to the current question as they may appear. They refer to what we might call "behavioural discounting". This is different than the discount factor γ used in the TD update equation (Equation 2.8). As discussed in Section 4.3.2, the TD discount factor was introduced for practical reasons, to prevent Q values from constantly increasing. It was only later, after the parallels between TD prediction errors and dopamine were discovered, that it was suggested that γ might also explain behavioural discounting.

It is likely that relatively complex reasoning goes into the calculation of behavioural discounting, based on factors such as present needs, future risk/uncertainty, and so on. Thus while researchers such as Kim et al. (2008) have found that activity in dorsolateral prefrontal cortex (DLPFC) correlates with the behaviourally discounted values, we hesitate to apply that conclusion to the integrative discounting of this model.

In other words, when we say that the model uses integrative discounting, we are referring to the implementational question of how the discount is used to compute a prediction error signal, which may or may not be equivalent to the observed behavioural discounting. For example, if a constant signal is fed into the integrator the result would be linear discounting, if a linearly increasing signal were fed into the integrator the result would be polynomial discounting, and if some complex function computed in DLPFC were fed into the integrator the result might look like hyperbolic discounting. Thus it is difficult to infer the implementational mechanism based on behavioural data.

In theory, we would expect the integrative discounting to be located along with the rest of the error calculation process in the dopaminergic nuclei, barring evidence to the contrary. The problem is that this kind of detailed implementation question is very low level, involving a small internal process in the error calculation regions. It is difficult to observe or differentiate computational variables at that level, particularly when experimenters are not looking for them. Thus we are not aware of any data either for or against this hypothesis. In Section 6.3 we discuss a prediction that could be used to generate such evidence.

Unfortunately the mapping is similarly inconclusive with respect to the positive bias mechanism. Again this is a low level implementational mechanism, for which it is difficult to find evidence either for or against. In addition, it would not be surprising to find that the positive bias mechanism is not well supported. Recall that the reason the positive bias was introduced was to supplement the relatively weak exploration mechanism in this model. Real brains have much more complex exploration mechanisms, and if such a mechanism were included in this model (for example, an exploration component that could notice that no random exploration had occurred for a long time), then it might be possible to remove the positive bias mechanism entirely. In other words, as with some of the action and reward components, we can think of the positive bias circuit as a placeholder for more complex exploratory networks, and not something we would necessarily expect to find a neuroanatomical correlate for.

4.5.4 Hierarchical composition

Empirical data on hierarchical reinforcement learning is more difficult to come by than standard RL. One reason is that HRL is simply a younger field, and has had less time to accumulate data; the neural correlates of reinforcement learning have been under investigation since at least the mid-90s, whereas data on HRL has only started to appear within the last 5–7 years. Another challenge is that HRL tasks are required to be more complex, in order to provide a hierarchical structure. This makes them difficult to apply to animal models like rodents, or even primates, without simplifying the hierarchies to the point of triviality. Thus much of the HRL data comes from human studies, using non-invasive methods such as fMRI. This means that often we can only speak of general activity levels in relatively large brain regions, rather than looking at detailed neural signals as we were able to do in previous sections. However, we can still find evidence to support the general hierarchical structure we suggested in Section 4.4.

The first question to be addressed is whether there are different hierarchical levels in the brain's RL processes, corresponding to the different levels in the model. Specifically, this model suggests that the different processing levels are represented by physically distinct neural networks, and that they interact in a relatively one-way fashion with higher levels influencing lower. There is indeed neuroanatomical evidence supporting this type of structure. Although we have been speaking previously of the basal ganglia/striatum as unitary objects, in fact the basal ganglia is divided into many relatively independent loops (see Figure 4.9). That is, a particular area of cortex will project to a particular striatal region, which will make its way through the basal ganglia to a particular region of the thalamus, and then back to the same cortical region (Redgrave et al., 1999). Thus there are multiple repeated action value/action selection circuits, which could implement the different levels of a hierarchy.

In addition, more recent results have shown that these loops are actually more of a spiral, with information from "higher level" cortical areas spiralling through the basal ganglia down into "lower level" areas. For example, Haruno and Kawato (2006) gave subjects a two-choice task, where they were shown an image and had to press one of two buttons based on the stimulus. Each stimulus was associated with different reward probabilities for the two buttons. As expected, they found activity in the striatum correlated with the theoretical action values and prediction error. The important observation from our perspective is that as they increased the task difficulty, different regions of the basal ganglia became more active; specifically, the activity related to action values moved in a caudal–rostral direction within the striatum.

Badre et al. (2010) found a similar result in a different task that gives more insight into the specific link to hierarchical processing. In this study the setup was similar—subjects were shown a stimulus, and had to press one of three buttons. The stimuli were objects that could vary in shape, colour, and orientation. In one condition each stimulus was associated with a random button. In the other condition, the button associations were governed by a higher level rule. For example, a rule might be "if the stimulus is red, respond according to orientation" versus "if the stimulus is blue, respond according to shape". In the first condition the learning process is simple reinforcement learning. The subject sees a stimulus, presses a button, and receives a reward which they can use to update the value of that button for that stimulus. In the second condition subjects can learn at two different levels.

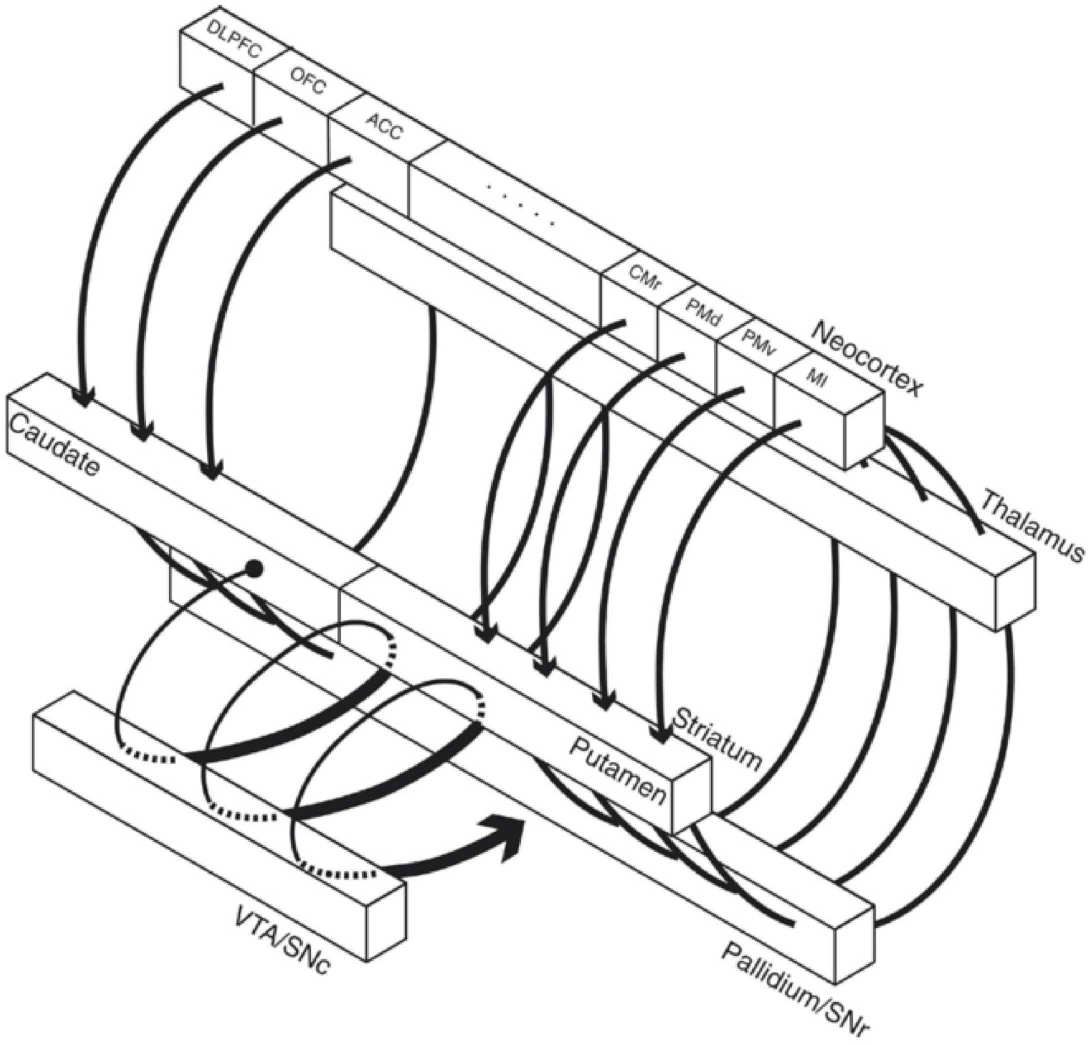


Figure 4.9: Visualization of cortical–striatal–thalamic loop/spiral structure, illustrating multiple parallel systems and caudal–rostral gradient. Figure from Haruno and Kawato (2006).

At the high level they can learn the abstract rule; that is, when they see a red stimulus they can respond according to orientation, and if they are rewarded then they can update the value of that rule. At a lower level they need to learn the implementation of the rule (i.e., given that the rule is "respond according to orientation", which button should they press for the given stimulus). Similar to Haruno and Kawato (2006), Badre et al. (2010) found that the hierarchical condition activated more anterior striatal regions.

These caudal–rostral gradients in the striatum parallel a similar pattern in the cortex, corresponding to the inputs/outputs of the basal ganglia loops. That is, the involvement of more complex/hierarchical tasks tends to involve increasingly anterior cortical regions, moving from the unimodal sensory/motor regions into the more abstract associative prefrontal cortex (Badre et al., 2010). For example, Golde et al. (2010) found that as the number of rules increased in a pattern completion based intelligence test, activity moved more and more anterior in the prefrontal cortex.

In summary, there is relatively strong evidence that there are multiple distinct cortico–striatal–thalamic processing circuits, as suggested by the repeated RL circuits that form the hierarchical structure of this model. In addition, these loops interact in a top-down fashion, with higher level circuits projecting information to lower levels, again as employed in the model. Finally, we can place these multiple levels in a roughly caudal–rostral gradient, with higher levels of the hierarchy occupying increasingly anterior regions of striatum/cortex.

However, it is important to note that although the simple repetition of this mapping is appealing, as we move to increasingly high level RL processing there may be qualitative differences in the neural circuitry. For example, although we have proposed that the basic representation of action values occurs in the striatum, there is evidence that in more complex tasks prefrontal regions such as anterior cingulate cortex or OFC may be involved in representing action values (Botvinick et al., 2009; Holroyd and Yeung, 2012). These signals are still relayed through the striatum, thus we would still expect to see striatal activity correlated with action values. The difference is in where those values are being computed. In the basic story, action values are computed by the connection weights from cortex to striatum. This is essentially a one-hidden-layer function approximation model. However, in more abstract contexts, such as deciding the next step in repairing a bicycle, it seems unlikely that the values of the various options are computed directly in cortico–striatal weights. Rather, complex circuitry involving memory, planning, and more would all go into determining the values of the different available options. Those values can then be relayed through the striatum in order to continue the normal decision making/learning process. The resulting neuroanatomical mapping will depend on the task-specific implementation; here we only want to highlight the fact that such an extension is not incompatible with the rest of the mapping presented here.

Hierarchical interactions

The next issue is the neuroanatomical correlates of the three different hierarchical interactions we defined (context, state, and reward, see Section 4.4). The model does not describe specific mechanisms for these interactions; rather, they represent general categories, which could be implemented in various different ways. Thus we will not focus here on establishing a specific neuroanatomical mapping for these interactions, but instead focus on their plausibility. That is, we will review evidence that the brain does employ these different interaction types in hierarchical RL environments.

The context interactions are perhaps the most well established. Botvinick et al. (2009) summarize data from several different studies, in an effort to extend the standard actor-critic neuroanatomical mapping to HRL. Our model does not employ the actor-critic architecture, but the underlying features of interest are the same in this case. Botvinick et al. (2009) show first that contextual information is represented in prefrontal areas (which we would expect given the above mapping of higher level layers onto more anterior regions), and second that those representations modulate activity in striatum (i.e., the representation of action values). Thus the mechanisms are in place to implement the context interaction type; a high level action can modify the context representation in some prefrontal region, and that context can then drive different action values (and resulting policies) in lower level RL processes.

State interactions have been investigated in the work of O'Reilly and Frank (2006) and Frank and Badre (2012). The former presents a model of RL-based gating, where basal ganglia output selectively inhibits different aspects of state represented in cortex. This is an example of state abstraction, where the basal ganglia is learning which aspects of the state are relevant in a given task. Frank and Badre (2012) extend this model to a hierarchical context, where the output of one basal ganglia loop gates the input to the next (these models are described in more detail in Sections 3.1.3 and 3.2.2). These results are somewhat indirect evidence, as the existence of a model does not imply a correspondence in real brains. However, these models are well-based in neuroanatomical evidence showing that the basal ganglia does have these kinds of gating abilities (see Frank et al. 2001 for a review). In addition, the authors go to extensive effort to compare the results of the model to experimental data (e.g., Badre and Frank, 2012). At the least this work demonstrates that state interaction is plausibly supported by the neuroanatomy, and more generously it provides a specific neural mechanism for that interaction.

The last question is the plausibility of reward interactions; in other words, is there neural evidence for the use of pseudoreward in the reinforcement learning circuitry? Excellent data in this regard comes from recent work by Ribas-Fernandes et al. (2011). They created a simulated navigation task, where subjects had to move a cursor to a package, pick it up, and then drop it off at a target location. In order to search for evidence of pseudoreward, they manipulated the task by shifting the location of the package while the subject was en route. If the overall distance to the target location were made longer we would expect this to induce a negative prediction error, because the agent is suddenly in a worse state than they expected. The interesting result comes when the package is shifted to a point that does not change the overall distance to the target, but does change the distance to the package. In this case the distance to the reward is unchanged, so in a basic RL framework there would be no prediction error. However, Ribas-Fernandes et al. observe a negative prediction error in the anterior cingulate cortex when the shift occurs. This suggests that there is pseudoreward associated with reaching the package; the new state is farther from that pseudoreward, even if it is no farther from the task reward, which leads to a negative prediction error.

This result demonstrates the presence of subgoals, but it does not necessarily demonstrate a hierarchical relationship. The key feature of reward interaction, as we define it, is that it is based on the output of a higher layer. In this task subjects could just be representing the problem as a sequence of two independent tasks, without any driving influence from a higher level. Ribas-Fernandes et al. (2011) investigate this possibility in a follow-up experiment, where at the beginning of the trial subjects are shown two different packages and asked which one they would like to deliver. If they are representing the package as an independent goal they should reliably pick the closer package (assuming the overall distance is the same).[11] However, Ribas-Fernandes et al. found no significant relationship between package distance and subject choice, despite the reliable presence of pseudoreward prediction errors. This shows that subjects do not innately value a short package distance, they only value it within the context of the overall task. In other words, the pseudoreward is driven by the higher level problem solving process. This means that the observed prediction errors are due to true pseudorewards, in the manner defined by reward interactions.

4.5.5 Summary

This completes the neuroanatomical mapping for all the major components of the model. For some there is strong experimental evidence and a fairly confident neural mapping, such as the association of prediction error with the dopaminergic nuclei. In others we have put together evidence from various sources to support a mapping that is plausible but unproven

[11]Or the farther package, if they were trying to optimize the delivery part of the task.

(e.g., the mapping of the dual training system onto dorsal and ventral striatum). And in a few cases we have pointed out model features for which there is little evidence either for or against, such as integrative discounting.

The first purpose of this mapping is to support the biological plausibility of the model. We seek to show that this model is a plausible neural account of hierarchical reinforcement learning, not just at the level of neural function but in terms of the macro-level neuroanatomical features of the brain. The other purpose of the mapping is to assist in producing testable claims that can be used to investigate the model experimentally. For example, this mapping now represents a specific prediction that dorsal and ventral striatum are implementing the computations of the dual training system. If that prediction is born out then the model has provided new insight into a key aspect of basal ganglia anatomy. If it is not, then that motivates a re-evaluation of the model's implementation, and the search for a revised neuroanatomical mapping or a new method of propagating error updates back in time. In either case, the usefulness of the model is improved by the presence of a detailed neuroanatomical mapping.

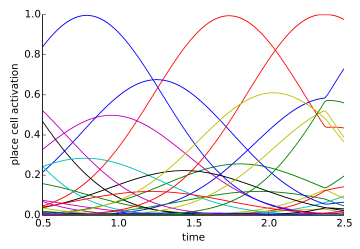
Chapter 5

Results

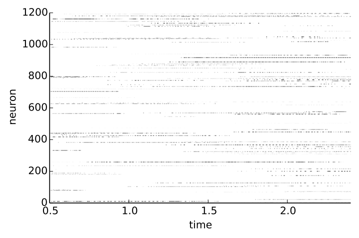
With the description of the model complete, we now move on to demonstrations of the model's performance. The goals of this chapter are threefold. First and foremost, the purpose of these results is to demonstrate that the model works—that the action values, action selection, and error calculation components all perform the functions described in the previous chapter, and that together they are able to carry out the hierarchical reinforcement learning process. The second goal is to demonstrate that the model's performance is consistent with neurophysiological data, in order to further support its biological plausibility. And third, we seek to demonstrate the advantages of hierarchical learning, in order to show the importance of including these features in models of decision making.

Note that it is not our goal to maximize the model's performance on some specific task. Achieving the highest performance on a given task requires building a model highly customized to that task, in order to take advantage of all the particular quirks of that environment. Although one could pursue that goal using our methods, that is not the focus of the work we present here. We seek to understand the brain's general reinforcement abilities; thus we consider it more important to demonstrate how the model can be flexibly applied to a range of tasks without modifying its basic structure, rather than showing how to build a model for a specific task.

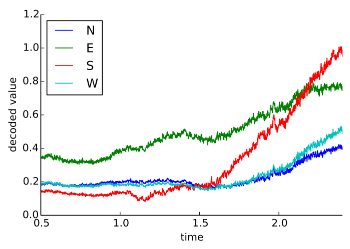
To that end, we have chosen the tasks we present here in order to showcase the different features of the model and demonstrate the range of its problem solving ability. In each section we will present the task in more detail, describe how the model is applied to that task, and then explore its performance.



(a)



(b)



(c)

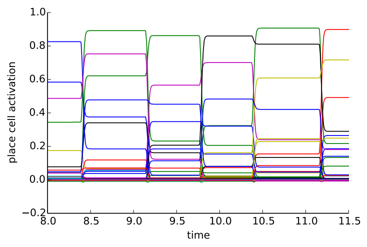
Figure 5.1: Example input and output for the action values component. The input (a) consists of simulated place cell activations, and the output (showing (b) spikes and (c) decoded values) indicates the value of moving in the four cardinal directions in a spatial navigation task (the target is to the south east of the agent).

5.1 Individual component performance

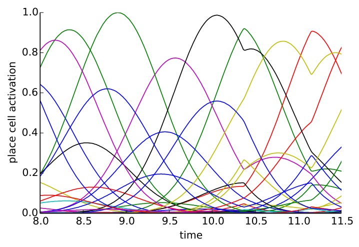
Before moving on to the tasks themselves, we begin by demonstrating the performance of the individual components of the model. In the previous chapter we described the desired function of each component, and the implementation designed to carry out that function. Here we seek to demonstrate that those implementations are successful and carry out the specified function. These examples are intended to be illustrative, rather than rigorous proofs. The quantitative performance of the model is examined in the context of the specific tasks in upcoming sections.

Figure 5.1 shows the output of the action values component. At the left we can see the input state; in this case, these are simulated place cell activations (Gaussian RBFs, see Section 4.1.1), with each line representing the activity of one cell. As the agent moves through the state space different cells become active. The middle figure shows the activity of the neurons representing the current Q function, which receive that state as input; each row corresponds to one neuron, and each dot is a spike from that neuron. On the far right is the vector $Q(s)$ that is decoded from the output of those neurons. We include the neural activities just to emphasize that all of the functions we describe here are being performed via neural computations. In future figures we will just show the decoded values, as they contain the meaningful, interpretable information.

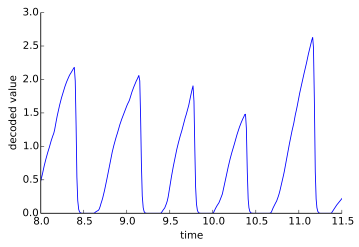
In this case the agent is approaching a rewarded state so, as expected, the action values are ramping up. The target is to the south east of the agent, so those actions have the two highest values, and we can see that as the agent moves eastward eventually the value of moving south becomes dominant. The actions that move away from the target (north



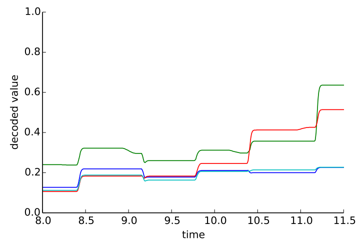
(a) Previous state



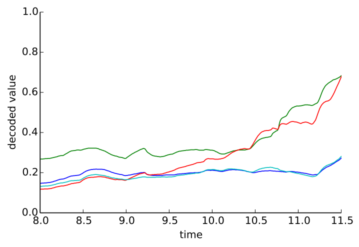
(b) Current state



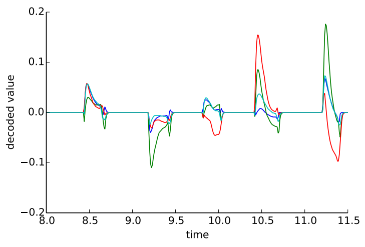
(c) State difference



(d) Previous state Q values



(e) Current state Q values



(f) Value difference

Figure 5.2: Computations of the dual training system. The previous and current state are shown in (a) and (b), respectively. (c) shows the output of the population computing the thresholded distance between those states. (d) and (e) show the Q values of the previous and current state. (f) shows the difference between those values, which is inhibited by (c) and used as the training signal for (e). See Section 4.1.2 for details.

and west) continue to have a low value throughout. In other words, the component output matches the Q values we would expect in the given state.

Figure 5.2 demonstrates the activity of the dual training system. At the top left we can see the previous and current state. To the right of that is the output of the population computing the thresholded distance between the two states; note that whenever the previous state is updated there is a brief window when the current and previous state are close to each other, and then they progressively diverge, as we would expect. The principle of the dual training system is that whenever the states are the same the output of the Q functions should be the same. Below the states we can see the output of the previous and current Q function, and the output of the population computing the difference between them. This population is inhibited by the state difference, and we can see that its output is zero whenever the state distance exceeds the threshold. The computed value difference is then used to update the current Q function, which can be observed in the deviations in the (e) signals (although this is complicated by the fact that the input state is also changing).

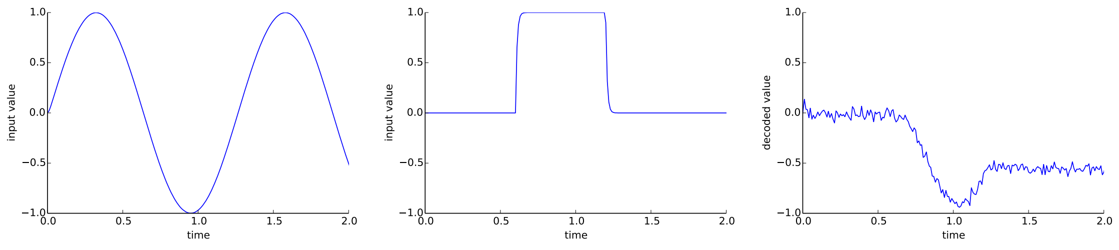

(a) (b) (c)

Figure 5.3: Demonstration of the memory circuit. (a) shows the target value, and (b) contains the gating signal that controls when that target is loaded into memory. It can be seen in (c) that the stored value is initially zero, then when the gating signal activates it begins to track the target value, and when the gating signal returns to zero the memory retains the last value.

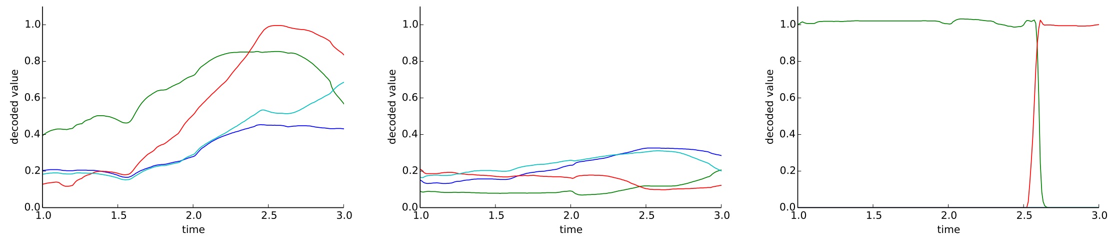
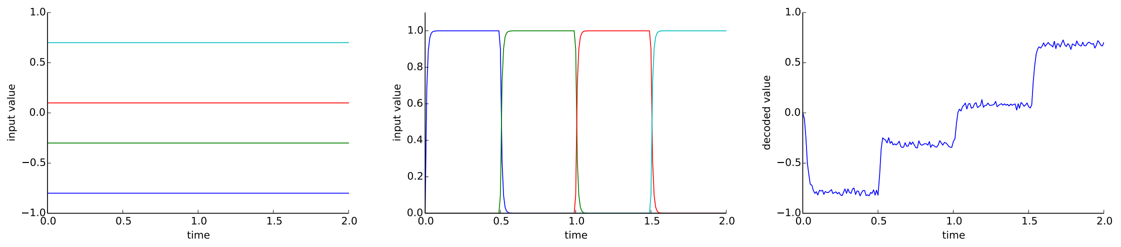

(a) (b) (c)

Figure 5.4: Operation of the action selection component. Showing (a) input action values, (b) output of the basal ganglia circuit, and (c) the final thalamic output, indicating 1 for the selected action and 0 elsewhere.

Figure 5.3 demonstrates the performance of the memory circuit that is used in several places throughout the model. On the left we can see the two inputs to the circuit, the target value and the signal controlling when the target value is loaded into the memory. To the right we see the output of the memory. At the beginning there is nothing stored in the memory so the output is zero. At $t = 0.6$ the timing signal activates, and the target value is loaded into memory. At $t = 1.2$ the timing signal turns off, after which the memory retains the last value loaded in. If we imagine that the target function is the current state and the timing signal is the action termination signal, it can be seen that this results in the memory network preserving the previous state.

Figure 5.4 demonstrates the performance of the action selection component. To the left



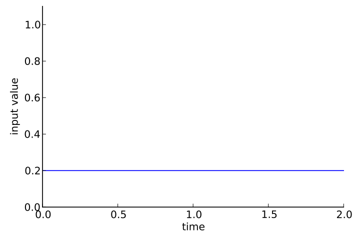
(a) (b) (c)

Figure 5.5: Demonstration of the selection circuit. (a) shows the input signal, (b) contains the signal indicating the selected dimension, and (c) shows the output of the selection circuit, which takes on the value of the selected dimension.

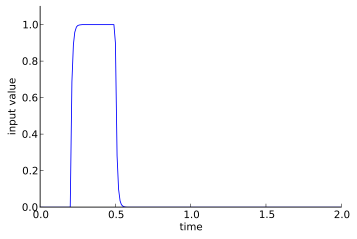
we can see the input to the component, a set of action values $Q(s)$. In the middle is the output of the basal ganglia network. Note that it has selected the highest valued action, but the selection is somewhat noisy and the output is inverted (the selected element has the lowest value). The rightmost figure shows the output of the thalamus, which has the the same action selected but with a much cleaner output of 1 for the selected action and 0 elsewhere.

Figure 5.5 shows the behaviour of the circuit used in several places throughout the model in order to select one element of a vector (for example, in order to isolate the Q value of the selected action). To the left we can see the input vector (e.g., $Q(s)$), and in the middle the selection vector (1 for the desired element and 0 elsewhere). On the right is the output of the selection circuit, and it can be seen that as the selection vector changes different elements of the input vector are produced as output.

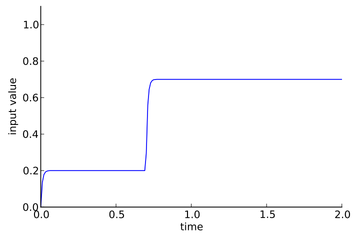
Figure 5.6 shows the SMDP TD error calculation process. The inputs of the error calculation are shown on the top: the previous action value ($Q(s, a)$), reward (r), and current action value ($Q(s', a')$). The bottom left shows the integrative discount, which is the integrated value of the previous action. Since $Q(s, a)$ is constant, the discount signal is a linear ramp. To the right is the integrated reward; it begins at zero, integrates the received reward from $t = 0.2$ to $t = 0.5$, and then remains at the resulting value since there is no more reward. In the bottom right is the calculated SMDP TD error, a continuous implementation of Equation 4.9. Initially the prediction error is zero, as the current and previous state have the same value and there is no reward. Then the agent receives some reward, which generates a positive prediction error. At $t = 0.7$ the current action value increases, further increasing the prediction error for the previous action value. We can also see the effect of the discount, reducing the future value over time.



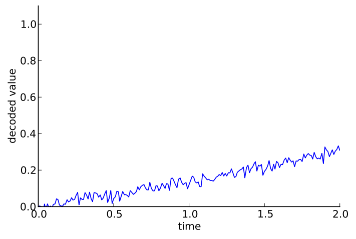
(a)



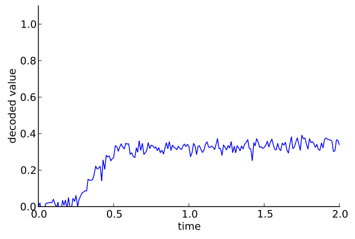
(b)



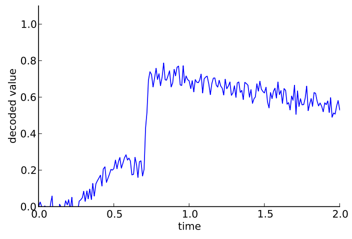
(c)



(d)

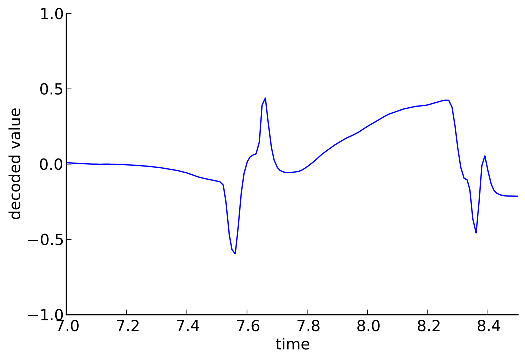


(e)

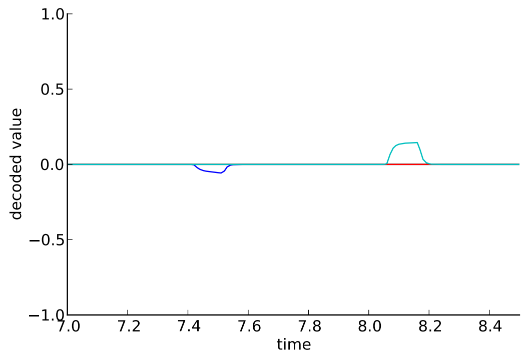


(f)

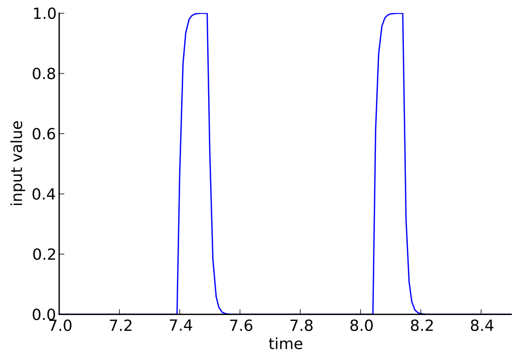
Figure 5.6: Computations of the error calculation network. Example input signals are shown in (a), (b), and (c) (the previous action value, reward, and current action value, respectively). (d) shows the integrative discount, (e) shows the accumulated reward, and (f) shows the overall output of the error calculation. Observe that (f) is equal to (c) + (e) - (d) - (a) (see Equation 4.9).



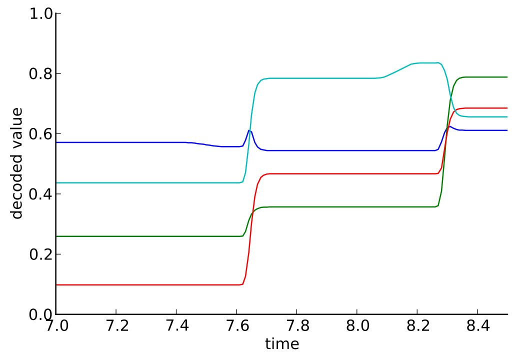
(a) Continuous error signal



(b) Gated error signal



(c) Termination signal



(d) $Q(s)$ values

Figure 5.7: Example application of the error signal to update Q values. (a) shows the error signal, analogous to that shown in Figure 5.6. (c) shows the action termination signal, which is used to gate the signal in (a), resulting in (b). (d) shows the Q values, demonstrating the impact of the error update on the represented values.

Finally, in Figure 5.7 we can see the calculated error signal being applied to the Q value representation. The continuous SMDP TD error signal is shown in the top left; this is the output signal from the network shown in Figure 5.6, but with realistic inputs. Recall that the SMDP TD error is then gated by the action termination signal and the selected action, in order to generate the learning update for the previous action value. The action termination signal is shown in the bottom left; note that after the action terminates the previous state is updated, hence the transition in the Q values at $t = 7.6$ and $t = 8.3$.[1] It can be seen in the top right that the error is gated by this termination signal—it is only non-zero when the action terminates.[2] The error signal is also gated by the selected action, so that the 1D error signal can be applied just to the selected action. The selected action is not shown, but it can be inferred based on the Q values, and we can see that the error update is correctly applied to the previously selected action at the end of the action period. As expected, when the error signal is applied the value of the previously selected action increases or decreases, proportionate to the error value. Thus this implements the TD update of Equation 2.18.

This completes the demonstration of all the major functional computations of the model. There are no dramatic conclusions to be drawn from these results, other than that the various neural components perform as intended, implementing the different functions required in this model. Of course many of these demonstrations have been toy examples, with hand-chosen inputs. In the upcoming sections we will see what happens when all of these components are interconnected in the complete model, and examine their performance on more interesting tasks.

5.2 SMDP navigation task

We begin by examining the performance of the basic model, without any hierarchy involved. As discussed in Section 3.1.5, the closest model to the one we present here is that of Potjans et al. (2009). Theirs is the only other biologically detailed model that includes TD error calculation in order to tackle temporally extended (i.e., non-associative) tasks. Thus in this section we will compare the performance of our model to the work of Potjans et al. (2009), in order to provide a baseline for the rest of the results we present.

[1]This state transition results in the spiky transients in the continuous error signal, as the previous action values are updated.

[2]There is also a cap placed on the magnitude of the error, in order to prevent outliers from radically altering the represented values. This is why the magnitude of the gated error signal at $t = 8.0$ is less than the magnitude of the continuous error signal.

5.2.1 Environment

The task used by Potjans et al. (2009) is a simple 5×5 grid navigation task. The agent begins in a random location, and must navigate to a fixed target location. The state is the x, y location of the agent. The model we present here is designed to work in continuous space (as opposed to the Potjans et al. 2009 model, where each grid cell has a dedicated neural state population), so operating in a discrete task like this is somewhat unnatural. However, we approximate the discrete environment by moving the agent only to fixed points in the continuous space, representing the centres of the grid cells.

The environment makes four actions available, each of which move the agent one square to the north, south, east, or west (unless the agent is at the edge of the grid in which case it stays in the same place). Recall that each action is represented by a vector; when the agent outputs a vector, the environment will move the agent in the corresponding direction. In the Potjans et al. (2009) model, state transitions occur instantaneously after the model makes a decision. In our model we use a random time delay of 600–900ms between state transitions, in order to demonstrate the ability of the model to perform in an SMDP environment. The agent receives a reward of 1 when it reaches the goal state, and 0 at all other times.

5.2.2 Results

Figure 5.8 shows the performance of the model on this task. A trial begins when the agent starts in a random grid cell, and ends when it reaches the target. The "latency" measure refers to the difference between how many steps the agent took to reach the target and the optimal number of steps (the Manhattan distance from the start location to the target). The "algorithmic" line shows the performance of a simple table-based Q learning implementation, for comparison.

It can be seen that all three implementations are able to learn the task. The Potjans et al. (2009) model is somewhat slower to learn, but within 100 trials all have achieved near-optimal performance. It is also important to note that the model we present here is operating on an SMDP version of the task, which could not be solved at all by the Potjans et al. (2009) model. However, the main conclusion to draw from this result is simply that the model's basic RL performance is at least as good as the nearest neural model, and we will now expand that performance into the domain of HRL.

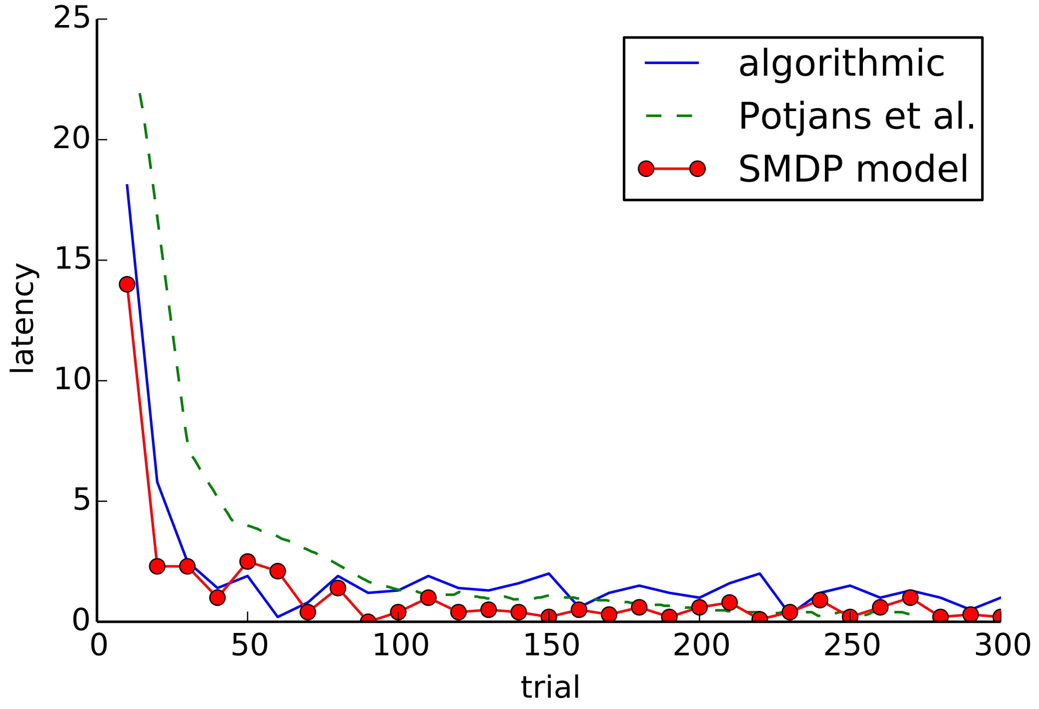


Figure 5.8: Performance of the model compared to a simple algorithmic Q-learning implementation and the model of Potjans et al. (2009).

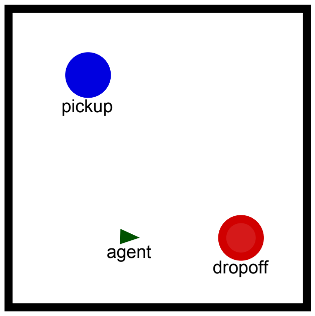


Figure 5.9: Schematic representation of the environment in the delivery task. The agent must navigate to the pickup location to retrieve the package and then move to the delivery location to receive reward.

5.3 Pick-up and delivery task

The first hierarchical task we will examine is a spatial navigation task, shown in Figure 5.9. The agent must move to one location to pick up a package, and then another to drop it off. This is a common task in HRL, used in both computational and experimental settings (Dietterich, 2000; Ribas-Fernandes et al., 2011). It is popular because it naturally breaks up into two different hierarchical levels; one level takes care of picking the target—managing the sequencing of the pick-up and delivery—while the lower level manages the physical movement to the target set by the higher level. It also represents a natural extension of the classic "gridworld" environments, an example of which we saw in the previous task, thus connecting to that work as well as showing how it can be extended via HRL.

5.3.1 Environment

As in the previous task, the environment provides four actions to the agent, corresponding to movement in the four cardinal directions. However, in this case the environment is represented continuously, so instead of moving through grid cells, the actions move the agent a short distance each simulation timestep. In all the tasks we present here the simulation timestep is one millisecond. If the agent attempts to move into a wall, it simply remains in the same location.

Rather than the raw x, y position, in this task the environment represents the agent's location using simulated place cell activations, as described in Section 4.1.1. Place cells are randomly distributed throughout the map, and each has a Gaussian activation based on the Euclidean distance from the agent to the centre of that place cell.

The package is represented by one of two vectors appended to the place cell activations, one vector representing that the agent has the package and the other indicating it is empty handed. The agent picks up the package automatically as soon as it enters the pick-up area. The d-dimensional vector of all the place cell activities and the package vector then forms the state input to the agent.

By default the agent receives a small negative reward of -0.05 as it moves throughout the environment, in order to encourage it to complete the task quickly. Only when it enters the delivery location with the package in hand does it receive a positive reward value of 1.5. After 600ms in the delivery location with the package, the package is reset and the agent is placed in a random location to begin the pick-up and delivery process again.

5.3.2 Hierarchical setup

In all tasks the basic structure of the model is exactly as described in Chapter 4. The only thing that is customized for a particular task is the hierarchical structure—that is, how several of the basic elements described in Chapter 4 are combined to solve the task hierarchically.

In this task the hierarchical structure consists of two layers. The lower layer has four actions, corresponding to the basic environmental actions (movement in the cardinal directions). These actions execute for a randomly determined time period (ranging between 600 and 900ms) and then terminate.[3] The higher level has two actions, representing "go to the pick-up location" and "go to the delivery location". These actions terminate when the agent reaches the specified location or after 30 seconds, whichever comes sooner. The timeout is to prevent the agent from becoming stuck selecting an action that never completes.

The layers interact via a context interaction. The output of the high level (e.g., "go to the pick-up location") is represented by a vector, which is appended to the state input of the lower level. Thus the low level has two contexts, a "pick-up" and "delivery" context. It learns a set of Q values for each action, describing a policy over the basic movements that will bring it to that location. The high level can switch between the different policies by changing its output action, thereby causing the agent to move to either the pick-up or delivery location via a single high level choice.

In order for the low level to learn the pick-up and delivery policies there needs to be pseudoreward associated with the targets set by the high level. Thus this task also involves

[3]The randomness was just added to make the task more complex, and demonstrate the ability of the model to perform in a stochastic SMDP environment.

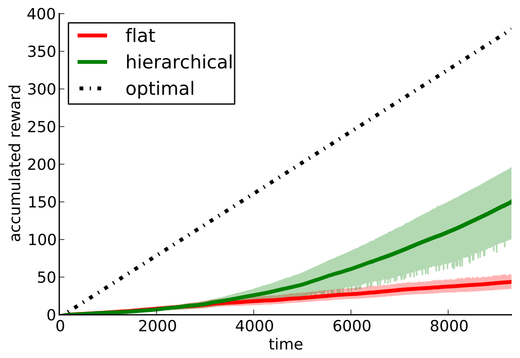


Figure 5.10: Performance of a flat versus hierarchical model on the pick-up and delivery task. Results are adjusted such that random performance corresponds to zero reward accumulation. The optimal line indicates the performance of an agent that always selects the action that takes it closest to the target.

a reward interaction. In this case the pseudoreward is 1.5 whenever the agent is in the location associated with the high level action (i.e. if the high level is outputting "pick-up" and the agent is in a pick-up state, the pseudoreward value is 1.5). At other times the pseudoreward is equal to a small negative penalty of -0.05. This penalty increases by -0.0001 every timestep that the agent attempts to move into a wall.[4] This pseudoreward signal completely replaces the reward signal for the lower level. Thus when the high level outputs "pick-up" the low level will learn to maximize the pseudoreward in that context, which means learning a policy that will bring it to the pick-up location.

This hierarchy is also an example of state interaction, because the vector indicating whether the agent has the package in hand or not is omitted for the low level (since it is irrelevant to the low level task). However, this abstraction is a fixed part of the hierarchical structure, not based on the output of the high level, so it is a fairly trivial example of state abstraction.

5.3.3 Results

In this task we seek to demonstrate the advantages of a hierarchical system. Figure 5.10 compares the performance of the hierarchical system described above to a flat model. The flat model consists of just one layer with the four basic actions, and the environmental

[4]This is a heuristic added to help prevent the agent from wasting long periods of time moving into walls, a common technique in these types of tasks.

state as input. The figure shows the total accumulated reward over time. Since this is the measure that the model seeks to maximize, the final point of this line indicates the agent's overall performance. Another useful measure is the slope of the line, which represents the rate of reward accumulation. This indicates how well the agent is performing at any point in time, independent of previous performance. In all figures the time axis indicates simulation time (in seconds), not real time; one simulation for this task takes 80–90 hours to run in real time.[5]

We run the model five times in each condition, using different randomly generated parameters each time (this includes variables such as neuron encoders, gains, and biases, place cell locations, and exploration noise). The shaded area shows 95% confidence intervals for the mean.

First, it is clear that the hierarchical model's overall performance is better than the flat model's, in that it accumulates much more reward. Interestingly, we can also see that the hierarchical model's instantaneous performance is higher at the end of the experiment (76% of optimal versus 12%). In other words, it is not just that the hierarchical model learns more quickly, it also learns a better policy than the flat model. The reasons for this are not obvious. The state input to the lower level contains the same information in either case, and the same environmental actions are available, so it would seem that the flat model should be able to learn an equivalent policy.

The reason for the better performance of the hierarchical system lies in the reward propagation. Due to the discount factor, the shape of the action value function is a gradient decreasing outwards from the rewarded states. In other words, the rewarded state has the highest value, the immediately surrounding states have a slightly lower value, further states have a slightly lower value, and so on. The discount goes to zero as the Q values approach zero, so the gradient will eventually flatten out. This means that the potential range of the Q values decreases the farther the agent moves from the target:

$$0 \leq Q(s,a) \leq \gamma(\|s - s_r\|)V(s_r) \tag{5.1}$$

where $V(s_r)$ is the value of the rewarded state and γ describes the shape of the discounting gradient. γ will change depending on the type and magnitude of discounting used, but it is always less than one and decreases as the distance to the rewarded state increases.

Equation 5.1 is significant for this model because the action selection component has limited precision. If the Q values are too close together then the basal ganglia model cannot tell them apart. In other words, if the agent finds itself too far from the target, it

[5]On a Dual Intel Xeon E5-2650 2GHz CPU.

will be unable to act effectively. It will just randomly wander until it happens to get close enough to begin to follow the Q gradient up to the reward.

The hierarchical framework helps to reduce this problem because, as discussed in Section 2.2, HRL reduces the reward propagation distance. This occurs in two ways. First, the high level actions effectively represent shortcuts across the state space. When the high level selects the "delivery" action, from its perspective it reaches the target location in a single state change. Thus the effective distance of the previous state from the reward is much shorter, and the gradient is less reduced. Second, the low level benefits from the pseudoreward interaction. The environmental reward is only administered in the delivery location, thus in the flat model the reward has to propagate from the delivery location, back through the pick-up location, and then outwards from there. The pseudoreward, on the other hand, is associated directly with both the pick-up and delivery locations, depending on the action selected by the high level. Thus in any given state the lower layer in the hierarchical system has a shorter distance to travel before it reaches a rewarding state. These two factors combine to allow the hierarchical model to perform more successfully than the flat model.

The interesting aspect of this result is that this advantage only appears in biological (or, more generally, noisy/imprecise) systems. A purely computational system can always distinguish the Q values, no matter how small their range may be. Thus in these systems the flat and hierarchical models would always converge to the same performance in the long run, the only difference would be the learning speed. This shows that when we incorporate HRL into neural models we do not just recreate the computational advantages, we can find important practical benefits specific to the constraints faced by neural systems.

As discussed in Section 2.2, another advantage of HRL is that it promotes the transfer of knowledge between related tasks. The abstract actions represent modular components that can be reused in different tasks, saving the effort that went into learning that subtask. We investigated this effect in this model by creating a simpler version of the delivery task where the agent is just rewarded for moving to a target location. There are several different locations where the target could be, and the environment indicates which context is active by appending a vector to the place cell activations. Note that this environment effectively recreates the subgoal structure of the lower layer in the delivery task; there are several different locations that the agent needs to learn to move to, depending on some context vector appended to the state. The only difference is that those contexts are randomly picked by the environment, rather than governed by the pick-up–delivery structure of the previous task.

We then trained a flat model in this environment. This model will therefore be learning

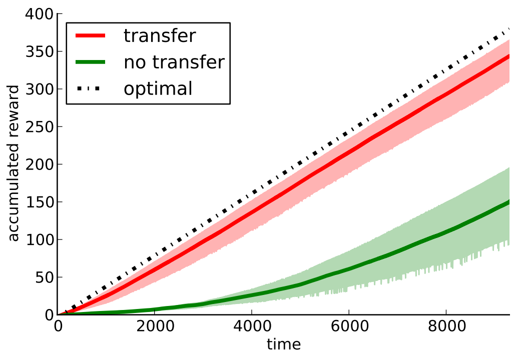


Figure 5.11: Performance of a hierarchical model where low level skills were transferred from previous learning.

the lower level policies needed in the delivery task—a set of $Q(s \oplus c, a)$ values that cause it to move to the target location associated with c—but not the high level structure of the delivery task. After 2000 seconds of training, we take the learned decoding weights from this model and use them to initialize the decoders of the lower level in the full delivery model.

Figure 5.11 shows the results of this knowledge transfer. The effect is dramatic—the model's performance is vastly improved on the delivery task. It is important to note that the improvement in performance goes beyond a simple 2000 second head start. The model with transfer very quickly reaches peak performance (96% of optimal), while the naive model is still learning at the end of the trial (76% of optimal).

This highlights an important advantage of transfer learning, namely that of incremental training. The high level's learning problem is significantly complicated if the low level does not reliably carry out its commands—from the high level perspective, this will look like an SMDP with a highly stochastic transition function. In the transfer case the high level is immediately presented with a reliable low level, significantly easing its learning problem. In other words, learning multiple layers simultaneously is more difficult than learning one layer at a time, and knowledge transfer enables the latter approach.

On the one hand, the result of Figure 5.11 is unsurprising; one could easily predict that a model benefiting from such pretraining would perform much better. However, what this simulation is intended to demonstrate is the ease of transferring knowledge between tasks. The knowledge transfer was possible because the two tasks were composed out of the same underlying subtasks, and the structure of the model captured that subtask structure. For example, it would not have been possible to load the previous knowledge from the simpler task into the flat model of the delivery task, because the flat model does not share any structure with the simpler task. From the perspective of the flat model, the solution to the delivery task is completely different from the simpler task, and it is not clear how that prior knowledge could be reused.

5.4 Hierarchical stimuli task

The second hierarchical test for the model is a recreation of the task from Badre et al. (2010). In this task subjects are shown a stimulus and must press one of three buttons. After pressing the button they receive feedback on whether they pressed the correct button or not. The stimuli are artificial shapes that vary in colour, shape, and orientation. There are two colours, three shapes, and three orientations, for a total of 18 stimuli, and therefore 18 stimuli–response mappings that need to be learned.

In order to investigate hierarchical processing, Badre et al. created two versions of the task. In the "flat" version each stimulus was mapped to an arbitrary button—there were no patterns in play, the subject just had to separately learn the correct button for each stimulus. In the "hierarchical" version of the task the button presses follow a rule. If the object has one colour, the button press is dependent on the shape (i.e., button A for shape 1, button B for shape 2, etc.) regardless of the orientation, and vice versa if the object has the other colour. In this case the task can be solved by learning 2 stimulus$\mapsto$rule mappings and 6 stimulus+rule$\mapsto$response mappings; in other words, the subject can use hierarchical learning to reduce the size of the problem space. The idea is that the difference in processing between the two tasks will reveal the effect of the brain's hierarchical processing components.

From a computational perspective, this task is not as interesting as the previous one. Each button press is independent, so there are no temporal sequences of actions involved; this is an associative RL task (see Section 3.1.2). The reason we chose this task is that it is one of the few hierarchical tasks that has experimental data at the neural level. This is important, because we do not just want to show that this model can perform HRL, we want to show that it is a plausible hypothesis for how the brain could perform HRL. Comparing the output of the model to experimental data is one way to do that. In addition, the closest model to the one we present here, the hierarchical PBWM model (Section 3.2.2) of Frank and Badre (2012), was applied to this task. Thus examining how our model performs on this task allows us to compare the performance of the two models.

In the previous two tasks we used the rate formulation for the LIF neurons that make up the model (Equation 2.33). For this task, since the goal is to compare as closely as possible to neural data, we use the spiking LIF model (Equation 2.32). Thus these results also demonstrate that the rate and spiking versions can be used interchangeably in this model.

5.4.1 Environment

The state in this task is the stimulus object. We follow the implementation of Frank and Badre (2012) in representing the stimulus via an 8 dimensional vector. The vector has one element for each possible value of the stimulus; the two colours are represented by the two vectors $[1, 0]$ and $[0, 1]$, the three shapes are $[1, 0, 0]$, $[0, 1, 0]$, and $[0, 0, 1]$, and so on. In the Frank and Badre (2012) model each attribute is represented as a separate input, but in our model we concatenate the three attribute vectors together to form the full stimulus representation (e.g. $[1, 0, 1, 0, 0, 1, 0, 0]$), which becomes the state input to the model.

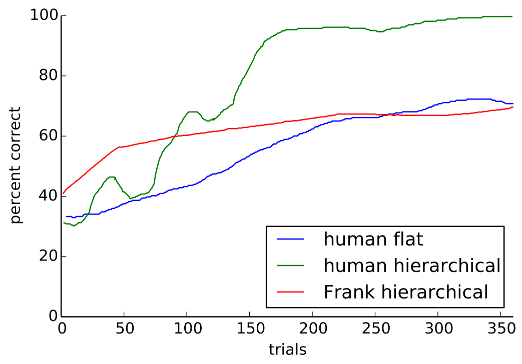
Each stimulus is presented for 500ms. During this time the model selects one of the three output actions (corresponding to the three buttons). After 500ms the environment checks the action to see if it is correct or not, and delivers a corresponding reward of ± 1.5 for 100ms. It then randomly picks a new stimulus and the process repeats.

5.4.2 Hierarchical setup

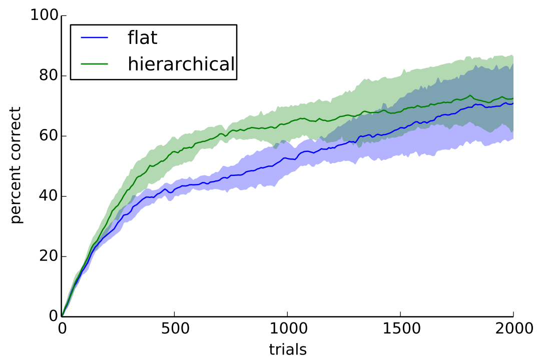
In this task the high level has two actions, corresponding to the two rules "respond according to shape" and "respond according to orientation", with a third option of producing no output, indicating no rule. The lower level has the three basic actions representing the three button presses. In this case all of these actions terminate after a fixed interval of 500ms, to align with the stimulus presentation described above.

The only hierarchical interaction in this task is state abstraction. When the high level selects the "shape" rule, what this means is that all the state information other than shape is irrelevant. In other words, the 8 dimensional state can be projected on to the 3 dimensional shape space. In the model we implement this by using the output vector of the high level to inhibit the irrelevant state elements in the input to the lower level (using a similar inhibitory selection circuit to that shown in Figure 4.6). The low level then learns a mapping from that reduced set of states to the output actions.

There is no context interaction in this case, as no new information is added to the state, and no reward interaction, as just the environmental reward is used to train both layers.



(a)



(b)

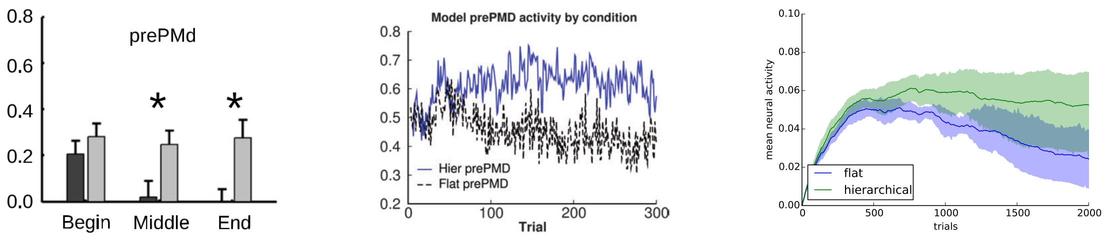
Figure 5.12: (a) Human data from Badre et al. (2010), as well as model data from Frank and Badre (2012). (b) Behavioural performance of the model in the flat and hierarchical conditions. Frank and Badre (2012) did not report performance of their model in the flat condition.

5.4.3 Results

Figure 5.12 shows the behavioural results from the model. The x axis shows the number of trials, where each trial is one stimulus presentation, and the y axis shows the percentage of trials answered correctly over time. The left figure shows the human data from Badre et al. (2010) on the flat and hierarchical conditions. We also show the data from Frank and Badre (2012) on the hierarchical task (performance on the flat condition was not reported). On the right we see results from our model on the flat and hierarchical tasks. Note that the model is the same in both conditions; in the previous experiment we kept the task the same and changed the model to be either flat or hierarchical, here we keep the model the same and change the task to be either flat or hierarchical.

The first observation is that neither our model nor that of Frank and Badre (2012) achieves human level performance. Performance peaks at around 70% accuracy, as opposed to the near-perfect performance of humans. However, these results do demonstrate the advantage of hierarchical processing, in that, as with humans, our model is able to learn more quickly on the hierarchical version of the task. In other words, the use of state abstraction does improve performance in tasks that allow for it. Thus we can see the benefit of incorporating hierarchical reasoning in neural models, while still observing that there is more work to be done.[6]

[6]Anecdotally, the reason for the model's limited performance seems to be that it over-generalizes the



(a) (b) (c)

Figure 5.13: (a) fMRI activation from Badre et al. (2010) in dorsal pre-premotor cortex (the region found to be associated with higher level hierarchical reasoning). Hierarchical and flat conditions are shown in light and dark grey, respectively. (b) Neural activity in output of higher level in Frank and Badre (2012) model. (c) Neural activity in output of higher level in the model presented here.

The main neural result from the Badre et al. (2010) study was that activity in anterior prefrontal regions was greater in the hierarchical versus the flat condition. This is consistent with the mapping we established in Section 4.5. However, Badre et al. found a particular temporal pattern of activation in those anterior regions, which provides a more detailed point of comparison. Namely, both conditions began with the same high activity level, and then the activity decreased in the flat condition. *A priori* it might seem equally likely that both conditions begin with a low activity level and then activity increases in the hierarchical condition; thus this result provides a unique, testable signature of hierarchical processing in the brain.

Figure 5.13 shows the comparison between the flat and hierarchical activation in the Badre et al. (2010) human data, Frank and Badre (2012) model, and our model. The Badre et al. (2010) figure shows fMRI data from the dorsal pre-premotor cortex, which was the area in PFC found to show a significant difference in activation in the flat versus hierarchical condition. What is being plotted is the change in activation from baseline at the beginning, middle, and end of learning. It can be seen that the difference in activation is due to a decrease in the flat condition, rather than an increase in the hierarchical condition. The Frank and Badre (2012) data shows the normalized firing rate in the component of their model mapped to that brain region. Our model data comes from the spiking activity of the population representing the action vector in the action selection component in the higher level layer. This is the activity that projects down to the input of the lower level, which is

high level rules. That is, once it determines that the "shape" rule is correct on some stimuli, it begins to apply the "shape" rule to all stimuli. This could perhaps be resolved by a different state encoding (greater sparsity would reduce the generalization).

what we would expect to drive the recorded prefrontal activity given the neuroanatomical mapping of Section 4.5.

It can be seen that the model shows initially equal activity in both conditions, followed by a decrease in the flat case. Quantitatively comparing this result to the human data is difficult, as they represent very different types of data. Similarly, we cannot say whether our model provides a better or worse fit than that of Frank and Badre (2012). The important point is that the model captures the observed trend of a decrease in the flat condition, rather than an increase in the hierarchical condition. In other words, it is at least as plausible as the Frank and Badre (2012) model in this respect, while providing all the functional advantages described previously (e.g., full temporally extended reinforcement learning).

One important thing to note is the scale on the x axis in each case; specifically, although the model we present here shows the same overall result, it takes longer to do so than the Frank and Badre (2012) model (or humans). Learning more slowly than humans is unsurprising, but why does the model learn more slowly than the Frank and Badre (2012) model? There could be several factors at play, but ultimately the answer is that this model has not been optimized for learning speed on this task. As mentioned in the introduction of this chapter, our goal in these results is to demonstrate the flexible performance of the same model across a range of tasks, rather than maximizing performance on one task. For example, even though this is a simple associative RL task, the model is still performing the full TD error calculation. In other words, the model is trying to find the button press that will not only get it the best immediate reward, but also maximize the reward of future button presses. Since in this task the trials are completely independent, this is a red herring, and will just serve to confuse the learning process. Humans can be instructed that the trials are independent (this is part of the experimental procedure in the Badre et al. 2010 task), and the Frank and Badre (2012) model is only able to perform associative RL and so has no potential to be confused. It is likely that if a custom version of this model were built for the Badre et al. (2010) task then it could achieve much faster learning than what we present here; however, we have not attempted to construct such a model, and so this is only speculation. An interesting project for future work would be to construct a system that was able to switch between associative and TD learning, either through learning or instruction.

The fact that the model is able to recreate this effect supports the plausibility of its explanation for HRL processing in the brain. However, one of the main advantages of creating functional neural models is that we can find mechanistic explanations for the observed phenomena, rather than simply matching the data. Thus it is interesting to explore *why* the model produces this activity pattern.

To begin, we can examine why the output activity is high or low. The activity levels are due to the output vectors associated with the different actions of the high level. Recall that these vectors are used to inhibit the state input to the lower level. In the flat condition no state abstraction is possible, so nothing should be inhibited, so the output vector should be zero. However, in the NEF there is no necessary connection between the represented value and the neural activity levels; that is, it is not obvious why a zero valued vector should result in low spiking activity. The key is that in these inhibitory circuits it is important that the inhibition signal be a very exact representation of zero (when not trying to inhibit). Otherwise small fluctuations around zero are magnified in the inhibited signal, introducing unwanted noise. As discussed in Section 4.1.2, a more accurate zero can be represented by aligning the firing thresholds of the neurons with the zero value. This has the functional effect of a cleaner inhibition signal, but it also has the neural effect of linking the represented value to the neural firing rate. Thus this functional optimization explains why we find lower activity in the flat condition.

Another question is why the initial tendency is to a high activity level, rather than a low one. In this case this is due to a specific change we made to the model in order to recreate this effect. We added a small positive bias to the reward for the high level when it chose one of the two rules as the output action, even if it was the wrong rule; specifically, the reward for the high level was ± 1.5 if it selected the null action, but -1.35/+1.65 if it selected the shape or orientation rule. This makes it more likely that the model will pick the non-null actions during the initial exploration phase, meaning it will have higher output activity. Eventually the model will learn the correct response (+1.5 is still better than -1.35), which will then lead to a decrease in the activity levels; the bias just serves as an initial nudge in the hierarchical direction.

Thus in this case the bump effect is something we explicitly added, rather than arising naturally out of the functional constraints as in the previous case. This is less satisfying, but it still provides interesting insight. Namely, this modification represents a specific prediction/hypothesis: the brain has a small positive reward bias associated with learning abstract rules. This is not an unreasonable prediction; one of the basic functions of the brain is to find patterns (rules) governing the world around it, thus it seems plausible that it would have a bias towards learning a rule-based account versus learning an arbitrary stimulus–response mapping. We will discuss how these predictions might be tested experimentally in Section 6.3.

Chapter 6

Discussion

We have divided the discussion chapter into three main sections. In the first we address, in no particular order, various questions that may arise regarding the choices that were made in the design of this model. In the next section we discuss how the model could be extended in various ways. These extensions range from short to long term research efforts; some are relatively small changes that would improve the existing function of the model, while others add extensive new functionality. We finish the chapter by recalling some of the predictions of the model, and outline how those predictions might be investigated experimentally.

6.1 Model questions

6.1.1 Why not use the actor-critic architecture?

Many biological neural models use an actor-critic architecture (e.g., O'Reilly and Frank, 2006; Potjans et al., 2009; Botvinick et al., 2009; Frémaux et al., 2013), thus the decision not to use that structure in this model may seem surprising. To understand why, it is helpful to examine the reasons for the popularity of actor-critic elsewhere.

From a functional perspective, the main advantage of actor-critic is its simplicity. Separating the state values and the policy into two relatively independent functions reduces the RL process to two function approximation problems. That is, instead of having to learn action values, find the highest value, and then convert that value into an output action, the problem is to learn two functions—one mapping states to values, the other mapping states to actions. The modeller does not need to worry about the internal computations involved in those mappings; from a modelling perspective they are just arbitrary input–output associations, to be learned by trial and error. In addition, both of the functions can be trained by the same error signal. Doing this kind of input–output function approximation is very well studied in neural networks, thus composing the RL process out of that basic operation is a natural approach to constructing a neural model of RL.

In contrast, recall all the different computational components described in Chapter 4: selection circuits, memory circuits, basal ganglia/thalamus models, and so on. And none of these circuits are independent; each is interconnected with other components, with the output of one forming the input to another. One might wonder then, why bother with this extra complexity when we could use actor-critic instead?

The first observation is that this complexity is greatly reduced by the methods of the NEF (Section 2.3). In traditional modelling approaches every aspect of the model has to be learned, because there is no other way to get the neural components to compute the desired function. Learning the various components described in Chapter 4 based only on the environmental reward signal would be impossible; there are too many degrees of freedom, interlinked by complex relationships, all of which need to work together to produce a correct response. In the best case scenario, the modeller could train each component independently via supervised methods. For example, the modeller could construct a neural network with the physical structure of the memory circuit, and then use training examples of desired inputs and outputs to try to teach it to compute the desired function. Then the modeller could assemble all the pretrained components together into the overall model and see if it works. This would be a lengthy and difficult process, and in any case it would be much more complex than simply training an actor-critic network.

However, as discussed in Section 2.3.2, the NEF allows us to specify the desired functions/transformations and then analytically determine the weights. When the desired computations of a component can be deterministically defined ahead of time, as is the case for the majority of the components in this model, there is no need to learn that component. For example, the performance of the action selection component is static, it does not need to change its function based on the reward signal; instead, the learning can be focused on the dynamic part of the RL algorithm, namely the action values. Thus the NEF makes it possible to build more complex structure into the model, by reducing the dependence on learning.

However, even given that the NEF allows us to build models with more complex structure, the question remains why we would want to if actor-critic is already working. The answer is that actor-critic has a number of functional disadvantages. One problem is that it is inefficient, from a learning perspective. An actor-critic model has to learn two functions, one for state values and one for the policy. In addition, those two functions are dependent on one another; for example, it is impossible to learn an effective actor function until the critic function has learned the approximate state values.

A Q function gives both those pieces of information in one function. That is, if you know the Q function you can determine the state values ($V(s) = \max_a Q(s, a)$) and the policy (via, e.g., Equation 2.7). Learning a Q function is no more complicated than learning an actor function; in fact, often the actor function looks almost exactly like a Q function, but the values are called "action strengths" instead of action values (Botvinick et al., 2009). So using a Q value approach it is possible to learn equivalent information, via a roughly equivalent learning update, but the system only needs to learn one function instead of two.[1]

This may contribute to the fact that while actor-critic is popular in the domain of neural modelling, it tends to be less used in purely computational RL. For example, the three main approaches to HRL, and all the computational work based on those approaches, use action value techniques rather than actor-critic. Thus another reason to use an action-value based approach is that it allows us to stay closer to the computational techniques we are attempting to model. While it is certainly possible to adapt these approaches to work in an actor-critic architecture (e.g., Botvinick et al., 2009), building models that are able to work directly with action values removes the need for any such adaptation.

Another reason for the popularity of actor-critic is that its components have been mapped onto known features of the neuroanatomy. With respect to the critic, the main evidence supporting its neuroanatomical mapping is the observation of dopamine neuron activity corresponding to TD prediction error, and the projection of those neurons to the striatum where dopamine modulates synaptic plasticity (Joel et al., 2002). However, while this is consistent with the actor-critic framework, it is also consistent with many other RL approaches that make use of a TD error (such as the model we present here, as discussed in Section 4.5). In addition, more recent work by Morris et al. (2006) and Roesch et al. (2007) has shown that the striatal dopamine signal is better characterized as an action value prediction error, rather than the state value prediction error employed in the critic.

Evidence for the actor is more diffuse, as the actor tends to be implemented differently depending on the action output required in a model. O'Doherty et al. (2004) supported

[1]This may bring to mind the two functions in the dual training system. However, those two functions address a different problem—how to apply the SMDP TD update using a local learning rule. This is still a problem in the actor-critic architecture, so it would also require something akin to the dual training system, for four functions total.

the actor-critic distinction by creating two versions of an RL task, one where the subjects observed two stimuli and then had to pick one, and the other where they simply observed one of the two stimuli and then were rewarded accordingly. The assumption is that in both cases subjects learn the value function (the critic), but only in the decision making paradigm do they need to learn a policy (the actor). O'Doherty et al. found activity in the ventral striatum in both cases but dorsal activity only in the decision making case, supporting a mapping of the critic to the former and actor to the latter. However, this type of data describing the overall activity level of broad anatomical areas is difficult to use to differentiate competing computational hypotheses. That is, the fact that the dorsal striatum is less active in one case does not tell us *why* it is less active. Using the neuroanatomical mapping established in Section 4.5, we could propose that the magnitude of the dual training signal is reduced in situations where the agent does not need to actively make decisions (since there is less urgency to update the Q function that drives action selection), thereby leading to reduced activity in dorsal striatum. Both explanations could cite the work of O'Doherty et al. (2004) as supporting evidence.

These difficulties are characteristic of many attempts to develop neuroanatomical mappings for computational models. Namely, while it can often be shown that a given model is consistent with neural data, it can be difficult to find data that distinguishes the specific implementation of one approach from other possibilities. The problem is that the implementational distinctions are often at a level of detail that is difficult to observe using our existing experimental methods. For example, we can observe the overall activity of dopaminergic nuclei and see that it correlates with a prediction error signal, but we cannot observe the internal computations that give rise to that signal. Note that this is not a critique unique to the actor-critic architecture—it applies equally to the neuroanatomical mapping of this model. What this data establishes is that the actor-critic architecture is consistent with the existing data, which is a good thing for any proposed biological neural model. The intent here is simply to highlight the fact that this consistency is not a reason to favour the actor-critic architecture over other, also consistent, approaches.

In the absence of experimental data that strongly distinguishes one approach from another, our general approach is to favour the one with greater functional ability. In this case action-value methods are more efficient and are more easily connected to the computational literature, which is why we adopted that approach in this model.

6.1.2 Why is the model based primarily on the options framework, rather than MAXQ/HAM?

First, it is important to note that this model is not a pure implementation of the options approach, and can combine aspects of the various HRL frameworks depending on how the hierarchical interactions are constructed (see Section 4.4.2). Rather than recreating any particular HRL approach, the goal in the hierarchical structure of this model was to make the minimal changes necessary to adapt the basic SMDP RL model to HRL. As it happens, that was also the goal of the authors of the options framework. As they put it, "What is the minimal extension of the reinforcement learning framework that allows a general treatment of temporally abstract knowledge and action?" (Sutton et al., 1999, pg. 182). That is why this model bears the greatest similarity to the options approach.

For example, in the options framework the learning update applied after selecting an abstract action is the same as that applied after selecting a basic action. In either case the error is based on the difference between the initial value of the selected action and the action value where the action terminates (plus any reward received while the action was executing). The error calculation is not affected by the presence of higher or lower levels, each layer operates independently. This is nice from a modelling perspective, as it means that the same learning process can be used at all the different levels of the hierarchy. Contrast this with, for example, the MAXQ framework. In this case, computing the error for an abstract action involves recursively computing action values based on the value functions of all the lower layers. In addition, each layer uses a different value function for its internal error calculation than it does for the recursive computations, each requiring its own error signal to learn.

As another example, in the options approach (and the implementation of this model) all states and actions are treated equally, none have a special privilege or effect. This means that all inputs and outputs can flow over the same channel, and the model does not require any built in information about the structure of the state space. Compare this to the HAM approach, where states/actions are divided into qualitatively different types, each requiring different behaviour by the system. For example, some states involve the fixed execution of a basic action (where no learning is required), while in other states the agent must make a decision and learn the value of the chosen action.

These kinds of features represent significant additional complications for a neural implementation, which is why we tended away from those frameworks in this model. This is not to say that we should avoid any complex computations in a neural model. These complications come with important computational benefits, such as the efficient reuse of value function representations in MAXQ. In the future it may be interesting to try to include these features in a neural model. However, at this point in the neural modelling of HRL we are still seeking to understand the most basic issues of a neural implementation. Thus we have tried to implement the minimal neural structure of HRL, which corresponds most closely to the options framework.

6.1.3 Why encode the hierarchy into the physical structure of the model?

This relates to the previous question, in that one of the primary features of the options framework is its lack of a fixed structure. As discussed in Section 4.4.2, any of the options available in a given state can be selected at any time; it is not the case that the available actions change depending on a previous (higher level) decision. Thus given the points just made about the similarity of this model to the options framework, it may seem strange that the model employs a fixed hierarchical structure.

It would be entirely possible to implement the flexible structure of the options framework using essentially the same basic SMDP model (see Figure 6.1). The key would be to recursively connect the output of the model to its own input; the hierarchical interactions would be the same (context, state, and reward), but instead of e.g. the output action in the context interaction modifying the state of a lower level, it would modify its own state input. This would allow the model to represent policies with potentially infinite hierarchical depth, as in the options framework.

However, this added flexibility brings with it a number of challenges. One is that we lose the advantage described in the previous section, where all actions in a given SMDP are treated equally. Now the different actions in a given layer would have different effects. For example, if the agent selects a "context interaction" action then that needs to be appended to its input state, a "state interaction" action would modify the input state, and a basic action would be sent to the environment. Thus the model would need different output channels flowing to different locations, rather than the single input/output links of this model.

Another challenge is that the recursive model needs to explicitly keep track of the "call stack"—the record of previously selected abstract actions. This is necessary so that when an action terminates the agent can be returned to the appropriate internal state. For example, imagine an agent in context A selects an action b that puts it in context B, and then selects a primitive action x. When x terminates, the system then needs to check if b has also terminated. If it has, the context needs to be changed back from B to A. There is no action that makes that change, it is just part of the internal bookkeeping of the HRL

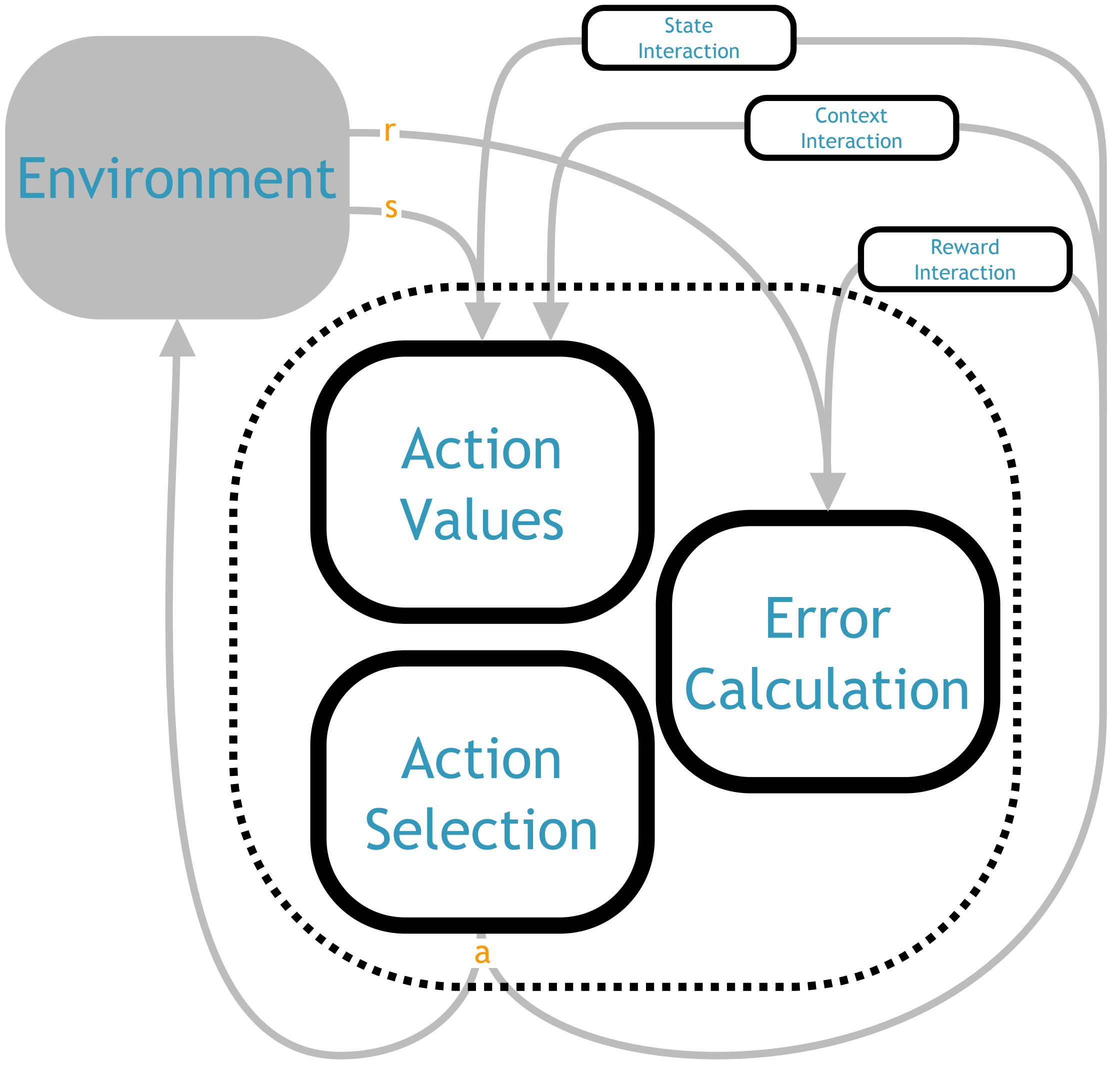


Figure 6.1: Example hierarchical architecture using a recursive structure (compare to Figure 4.7).

algorithm—so the model needs to implement some mechanism to keep track of all this information and execute the necessary changes.

In contrast, with a fixed hierarchical structure the "stack" is implicit in the hierarchy itself. For example, the above situation could be implemented with a two layer hierarchy. The top level is in context A. It selects action b, and its output then sets the lower level into context B; the lower level then selects action x, and the output of the lower level goes to the environment. Note that the top level continues to be in context A, and it continues to output the action b. The termination of b does not require any central mechanism, or any communication between layers. Each level just selects actions and waits for them to terminate, at which point it selects a new action.

Another way to phrase this is that in the recursive structure there can be multiple actions being executed simultaneously in a single layer. This requires more bookkeeping to keep track of which one is active and swap back and forth between them. With the hierarchical structure the actions are distributed throughout the hierarchy, so that only one action is being executed at a time in any given layer. This simplifies the hierarchical computations, and supports the modular structure we strive for.

Another advantage is that the distributed hierarchical structure allows computations to be performed in parallel, rather than serially. For example, in the above scenario the recursive approach has to first process the termination of x (computing the TD error and applying the learning update), and then do the same for action b. If we imagine that each termination cycle takes 100ms, then it would take 200ms to process the end of action x. In the hierarchical approach, error calculation proceeds in parallel at each level of the hierarchy; when x terminates in some new state, the high level can begin processing the termination of b at the same time that the low level processes the termination of x, only taking 100ms total. This is true of all the different computations going on in a layer, not just error calculation. The fixed hierarchical structure repeats the computational elements in each layer, allowing each to proceed independently, while the recursive structure only has one channel that all computations must flow through in turn. Thus there are functional as well as implementational advantages to a fixed hierarchy.

Finally, the neuroanatomical evidence is consistent with structurally distinct layers. As discussed in Section 4.5, multiple avenues of investigation have found that different levels of processing activate different physical regions of the brain. For example, Ribas-Fernandes et al. (2011) observed multiple, simultaneous prediction error signals in a hierarchical task, corresponding to the termination of a primitive action x and abstract action b as in our example, and Badre et al. (2010) found that the use of higher level actions activated different regions of the brain than simpler actions. These results show that a) the brain has multiple regions of RL processing operating in parallel and b) different actions are associated with different regions, which is consistent with a fixed hierarchical structure.

However, it is important to note that these options are not mutually exclusive. The brain could, and almost certainly does, employ a mixture of recursive and hierarchical approaches. In particular, the disadvantage of a fixed hierarchy is that it imposes a fixed limit on the hierarchical depth—if a system has two layers, then it can only learn at two different levels of abstraction. Clearly this is not a desirable feature in a hierarchical system. For example, imagine the depth of the hierarchy that connects the abstract decision to get into university with the immediate action of which icon to click on a computer screen. For one, it would require a huge amount of neural resources to repeat the whole RL circuitry the hundreds of times necessary to provide realistic hierarchical depths. In addition, such a model would exhibit rather strange behaviour, where it would work fine up to a certain depth and then completely fail for a depth of $n + 1$.

One reasonable hypothesis would be that lower levels of RL processing are distributed across separate hierarchical layers, while the top levels of the hierarchy contain the recursive connections that allow for processing of arbitrary hierarchical depth. The slower, long term reasoning of the higher levels would be more suited to the serial processing, while the quick low level decisions could proceed in parallel. This would also allow for a smooth degradation of performance as the hierarchical depth increases, as the bookkeeping mechanisms begin to break down (for example, the system begins to forget the contents of the stack). It would be an interesting project in the future to try to build such a hybrid system.

6.1.4 What does this model contribute to the computational study of HRL?

Understanding how hierarchical reinforcement learning might be implemented in the brain is certainly the main goal of this research. However, this effort can also provide value to the purely computational study of HRL, as it has several desirable implementational features that are independent of the goal of biological plausibility.

One such feature is that the model operates in continuous time and space. As discussed in Section 2.2, the majority of HRL work has been in discrete environments; the model of Konidaris and Barto (2009) is the only other continuous implementation of the options framework, and it is limited to a particular type of hierarchical structure. Many of the interesting real-world problems that researchers hope HRL will be able to address involve continuous environments, thus expanding the theory into those domains is an important step.

The key aspect to the continuous spatial implementation is the use of neural populations to approximate the value function. This is not a new insight, as neural networks have been used to approximate the value function in many previous models of RL. However, here we show that those same ideas can be extended to the hierarchical domain, where multiple value functions are being represented at different levels of abstraction. In addition, the context/state interactions show how information from different layers can make its way into these function approximations.

With respect to continuous time, the main issues to solve are how to calculate errors and apply learning updates over time rather than instantaneously. The error calculation component described in Section 4.3 addresses the former, demonstrating how to calculate a continuous version of the SMDP TD error, and how that signal can be gated to provide the error at the appropriate points in time. The local learning rule (Equation 2.40) and dual training system address the latter, demonstrating how to apply that error via a continuous learning rule in a purely online fashion (i.e., where the system does not have access to the past neural activations).

Finally, the results in Chapter 5 demonstrate that these ideas work in practice—that this model can solve tasks that involve continuous time and state spaces. While these tasks are still rather simple, they demonstrate the new possibilities that are opened up by a continuous HRL implementation.

Another interesting feature of this implementation is that it deals naturally with noise, imprecision, and heterogeneity in its components. Noise and heterogeneity are innate features of the brain, thus brain-like implementations must incorporate solutions to these problems from the ground up. For example, in our model every neuron has different randomly generated properties, and therefore responds differently to a given input. The NEF methods show how to combine these diverse activities in order to decode a reliable output signal. Similarly, the output of these neurons is imprecise, and may be noisy/fluctuating. We saw an example of this in the pick-up and delivery task, where the flat model was unable to solve the task due to imprecision in the propagated reward values. There we saw how HRL can be used to help address these problems, a feature that is not apparent in implementations with perfect precision in the Q values.

These problems of noise and heterogeneity are challenges that tend to arise when computational theories are applied in the real world, for example on robotic platforms. Often it is difficult to make the move from simulation to physical environments, because algorithms can have subtle assumptions/requirements on the reliability of processing components that are not apparent until they are violated. Implementations that incorporate these constraints from the ground up will be easier to transfer to real-world domains.

A third interesting feature of a neural implementation appears in connection with the development of neuromorphic hardware. These are custom computing platforms that use neuron-like elements as their basic computational elements (Merolla and Arthur, 2007; Khan et al., 2008). These platforms are able to simulate neural models much faster than conventional hardware, and often require a fraction of the power. This makes them well suited to mobile/robotic applications, where low power consumption can reduce weight and prolong battery life. However, the software that runs on these platforms must be implemented neurally, in order to program the neural hardware. Thus while a traditional HRL implementation could not be run on these platforms, a model such as the one we present here could be.

Neuromorphic hardware is still in the development stages, so we have not been able to try a model with the complexity of the one we present here on these chips. However, previous work has demonstrated that models built using the NEF can be run on these neuromorphic platforms (Galluppi et al., 2012, 2014). Therefore it is likely that the model we present here could be run on a neuromorphic chip in the near future, and combine the computational advantages of HRL with the strengths of neuromorphic hardware.

An important feature to keep in mind when examining the computational advantages of this implementation of HRL is that our simulation software (see Section 2.3.5) is designed to make it easy to adjust the level of simulation detail. In this work we are trying to demonstrate that all aspects of the HRL process could be implemented in the brain, so we implement all the components neurally. However, in a purely computational application someone might just be interested in using the function approximation of the action values component, without caring about using a model of the basal ganglia to do action selection. The model is designed to support this, allowing different components to be swapped in and out without affecting the rest of the model. That is, as long as the action selection component performs the same function, working with the same vector inputs and outputs, it does not matter how it is implemented internally. Thus a researcher need not commit to a full neural implementation in order to make use of this model.

Exploring in detail the computational advantages we have described here would be an in-depth research project, and we do not take this work to have proven these benefits. The purpose of this section is just to point towards these issues, and suggest how this model could be used as a starting point in that direction.

6.2 Model extensions

We now turn to a discussion of how this model could be modified, improved, or expanded. We have arranged these changes in a rough order of complexity, beginning with relatively small modifications and ending with large functional changes to the model. Also note that this is not an exhaustive list, but represents what we consider to be the most promising or important changes.

6.2.1 Improved exploration

In this model we have used a very simple model of exploration, where noise with a constant variance is added to the action values (see Section 4.2). However, this simplicity can be problematic. For example, in the pick-up and delivery task we were constantly struggling to prevent the model from becoming stuck in local minima. Exploration is key to this problem, but the simple exploration of this model did not provide much functionality; increasing the noise could help move it out of the local minima, but then the noise would prevent it from distinguishing the small differences in Q values in regions distant from reward. These experiences have led us to consider a number of ways in which the model's exploration could be improved.

One approach would be to modify the variance of the noise according to a fixed schedule over time. This is a relatively common practice in RL, often implemented via an adjustment of the temperature parameter in the softmax policy (Equation 2.7). The idea is that at the beginning of learning the agent should explore broadly in order to learn the general structure of its environment. Its policy is likely incorrect anyway, so there is no point attempting to stick to it rigidly. Over time the agent will learn a better policy, so in order to maximize reward it should begin to listen to that policy more and reduce its exploration.

A further improvement would be to base the variance of the noise on the performance of the model. A problem with the fixed scheme above is that the modeller has to guess the appropriate exploration schedule. Often this is not obvious, and an incorrect schedule will result in the agent missing out on reward due to over-exploring, or converging its policy to an inaccurate Q function. In addition, the appropriate level of exploration may not change monotonically over the course of learning—it could fluctuate as the agent learns different areas of the state space. A better approach is to try to infer how much exploration is needed dynamically over the course of learning. For example, one approach would be to base the exploration noise on the average prediction error magnitude. The more accurate the Q function, the smaller the prediction errors. Thus at the beginning of learning the agent would have large prediction errors and therefore more exploration, as desired. Over the course of the trial it would reduce its exploration in proportion to its learning progress, until in theory it has an optimal Q function, with no prediction errors and therefore no exploration.

An even more advanced mechanism would be to use directed rather than random exploration. That is, instead of the agent making accidental explorations and hoping to wander into interesting areas, it could actively choose where to explore. One way to go about this would be to track the action frequency. For example, the model could keep track of which actions had not been selected in a while, and gradually increase the probability of selecting those actions. Alternatively, the agent could track the state frequency. That is, it could track how often it visits different regions of the state space, and then direct the exploration towards unvisited regions. These model-based exploration schemes are more powerful, but would require significant internal processing of their own.

The different exploration schemes could all be added to the model without making any significant modifications to its existing structure. They would be implemented by a component that simply takes the place of the current random noise generator. It would then interact with the rest of the system in the same way, by outputting a vector that is added to the action values before they are input to the action selection component.

6.2.2 Separate positive and negative RL mechanisms

In this model we treat all environmental feedback the same, regardless of its valence. However, there is an abundance of evidence that the brain treats positive and negative outcomes qualitatively differently. For example, fMRI has revealed different areas of activation in ventral striatum depending on whether the prediction error was positive or negative, and different cellular mechanisms may underlie increases in synaptic weight versus decreases (see Dayan and Niv 2008 for a review).

These features could be modelled by various different changes to the model. For example, the positive and negative components of the Q functions could be represented separately, and then added together to form the overall output of the action values component. Or the model could have separate error calculation networks, with independent inputs for positive and negative rewards. At a lower level, the model could use different learning rules/learning rates for positive versus negative errors.

It is not clear at the moment whether there are functional advantages to separating positive and negative mechanisms, or whether this is just a biological spandrel. Perhaps this division could be used to help resolve the discounting problem described in Section 4.3.2, where the discount has a negative feedback interaction with positive actions but a positive feedback with negative actions. If there were separate representations for positive and negative Q values, then both could be represented via positive values (just weighting the output of the latter by -1); this would allow the same negative feedback discounting to be applied in both cases, removing the need for the somewhat *ad hoc* positive bias mechanism. In any case, building a model that incorporates these features is a good way to explore their functional impact.

Even if it turns out that this distinction is not important functionally, recreating it would help support the biological plausibility of this model. For example, if it turned out that certain features of the model rely on positive and negative reward being combined in the same mechanism, then that would raise questions about whether those are a reasonable hypothesis for the mechanisms used by the brain. In theory there is no reason why this should be the case for any of the components of this model, but actually creating a working implementation would be the best way to show that.

6.2.3 Continuous action

Although this model operates in continuous time and space, it is still limited to discrete actions. For example, in the delivery task the model can only move in the four cardinal directions, rather than the full 360 degree range of motion. Many interesting tasks involve a continuous action space, thus it would be useful to expand the model in this direction.

The main challenge when working in a continuous action space is how to represent the Q values. In this model we use the vector $Q(s)$, where each element corresponds to the value of one of the available actions. However, in this case we no longer have a discrete set of actions to choose from; $Q(s)$ needs to represent a continuous action space, rather than a vector.

This is the same problem encountered when moving from a discrete to continuous state space, and the solution is also the same—function approximation. The action space can be spanned by a set of basis actions, and then the continuous action space represented as a weighting over those bases. For example, in the delivery task the four cardinal movements could be the basis actions, and then different combinations of those movements would give all the possible 2D movements.

One nice feature of this approach is that it leaves the structure of the rest of the model relatively unchanged. From an implementation perspective there are still a finite set of actions to represent, but now they represent basis actions. For example, the output of the action values component would still be a vector $Q(s)$, but now that vector would represent the value of each basis action.

The main change is that the action selection output would need to be more nuanced. Instead of outputting a binary vector selecting just one action, it would need to output the basis function weights. For example, the basal ganglia/thalamus could be adapted to compute something like the softmax function (Equation 2.7). This would take the Q values as input, and output a vector where each basis action is weighted proportional to its Q value. Note that the basal ganglia output is naturally a soft selection like this (see Figure 5.4), so it would certainly be feasible for it to compute such a weighting. Multiplying the basis action vectors by that weighting vector would then give the overall action vector. Again, this is essentially what is already occurring in the model, but in the current implementation the weighting vector is binary so the result of the multiplication is always equal to one of the action vectors (see Section 4.2).

The error calculation can also be adapted to the new system without any major structural changes. In the current model the error signal is already gated by the output of the action selection component (so that the error is non-zero for the selected action and zero elsewhere). For continuous actions we can do the same thing, multiplying the TD error by the action selection output, but in this case the action selection output is the weighting vector. This means that the value of each basis action will be modified proportional to how much that basis action contributed to the output action, which is what we would want intuitively.

In summary, the model could be adapted to work in a continuous action space with surprisingly limited changes to its existing structure. Of course this is all theoretical, and the question remains whether the action selection component could compute something like a softmax, or whether the weighted error update would be effective. But it would certainly be possible to explore those questions using the current model.

Note that the system as described here would operate with actions that are continuous in space, but still discrete in time. That is, the actions have a distinct beginning and end. We discussed in Section 2.1.5 how Doya (2000) developed a version of RL for continuous time actions, where the agent outputs a continuously changing signal, which one might think of adapting for this case. However, it is difficult to imagine how continuous time actions could be applied to an SMDP framework. The basic premise of an SMDP is that actions have temporal extent, which requires that there is a point when the action begins and a point when it ends. The notion of subgoals and termination conditions in HRL is similarly bound up with the concept of actions that begin and end. Thus while it seems likely that this model could be adapted to work with continuous space actions, continuous time actions are at odds with the basic principles on which the model is based, and are unlikely to fit within this framework.

6.2.4 Average reward

Another interesting possibility would be to change the model to work with average reward Q values rather than discounted reward. In the standard TD framework, $Q(s, a)$ represents the total discounted sum of reward to be expected after selecting action a in state s, with time going to infinity. In the average reward framework, $Q(s, a)$ represents the average reward to be expected, rather than the sum. The nice thing about this approach is that the average reward is constant over time, so there is no need for a discount factor to keep Q values from going to infinity. The discount factor introduces a lot of potential problems in this model, such as the vanishing gradient problem described in Section 5.3, or the need to readjust the value of γ to keep the Q values in the correct range for the basal ganglia. Thus removing the discount might simplify the rest of the model.

There are different approaches to average reward reinforcement learning, but they all follow a similar structure. The method we will describe here is based on the work of Singh (1994), as it is closest to the standard Q learning approach. The main change when moving to an average reward framework is that state/action values are all relative to the overall average reward. Therefore the first addition is that the model needs to keep track of the average reward over time, r_{avg}. This can be implemented by a network similar to the memory circuit (Figure 4.3), where an integrator stores the average value and the stored value is updated based on the incoming samples (with some small scale applied).

With r_{avg} computed, the TD update is then changed to

$$\delta(s, a) = \alpha[Q(s', a') + r - r_{avg} - Q(s, a)] \tag{6.1}$$

Note that this is essentially the same TD update as in Equation 2.8, except $Q(s, a) = Q(s', a') + r(s, a) - r_{avg}$. That is, instead of a Q value representing the immediate reward plus the value of the next state, it represents whether the immediate reward is better or worse than average (plus the value of the next state). Since the average will converge to some fixed point, the Q values will also converge around that fixed point, without the need for a discount. Incorporating average reward into this model would be relatively easy; the population computing the average reward would just take the place of the integrative discount in the error calculation network (Figure 4.5).

The downside of the average reward approach is that it is less well developed than discounted reward methods. One important problem is that while average discount methods are guaranteed to find a policy that maximizes the average reward, r_{avg}, they are not guaranteed to find a policy that maximizes $Q(s, a)$ (Mahadevan, 1996). For example, imagine a task where the agent must navigate to a fixed target location to receive reward, and receives a small punishment in all other states. Given enough time the agent will learn to reach the rewarded state and stay there; this means that the average reward will converge to the reward in the target state, regardless of the path the agent takes to get there. Clearly a path that goes directly to the target is better than one that takes a long time, but average reward methods are not guaranteed to find that solution.

However, in practice average reward methods have been shown to find good solutions to many problems. It would be interesting to see how they perform in this model, and whether the benefits of removing the discount outweigh any complications associated with the average reward approach.

6.2.5 Model-based reinforcement learning

In Section 2.1 we briefly touched on model-based approaches to RL, but all the methods used in this work have been model-free. Model-based methods have many potential strengths, thus it is interesting to explore how they might be integrated into this work.

Recall that in model-based approaches the agent attempts to construct an explicit representation of its environment (generally consisting of the reward and transition functions). That model can then be used in several different ways. One approach is to use the model to generate extra TD learning updates. The only way the current model can update its Q function is to execute an action and then wait for the new state and reward from the environment. If the agent has a model of the environment, it can instead imagine selecting an action a, simulate the outcome of that action using its internal model, and then use that simulated outcome to update the value of a. This can often be done much more quickly than waiting for the environment to execute each action, allowing the agent to perform more learning updates in the same amount of time.

A more complex use of the internal model is to move away from Q value based approaches entirely. The purpose of a Q value is to estimate the future value of an action, so that the agent can pick the action that has good long-term outcomes without knowing what those outcomes will be. However, if the agent has a model of the environment it can internally simulate the long term outcomes of the currently available actions, and use that to pick the best action instead.

The main advantage of these model-based approaches is a more efficient use of sample data. In a model-free approach, the information observed from the environment after selecting an action a (the reward and new state) only updates one Q value, $Q(s, a)$ (or a few previous states when using eligibility traces). In a model-based approach the sample is instead used to update the model of the environment and reward functions. This can then be used to rapidly update many different Q values using the sampling approach, or it will affect all simulated paths that pass through the updated state in the planning approach. In either case, the single sample can have far-reaching effects on the agent's policy.

For example, imagine an agent that has been exploring around in an environment for a while but has not yet located the reward. If it has learned a model of the environment, as soon as it finds the reward it will then be able to plan a path to that state from any of its well-learned areas of the environment—the reward information is immediately propagated throughout the agent's policy. Contrast that to a model-free approach, where reward would only be propagated to the states immediately preceding the rewarded state, and the agent would need to reach the reward many more times in order to propagate that information throughout its policy.

Model-based approaches to HRL are relatively unexplored—the majority of HRL work is model-free. However, the efficiency benefits of model-based methods also apply in the hierarchical case. Sutton et al. (1999) discuss how model-based methods can be applied in the options framework, although their points apply in general to any SMDP-based approach. The key is to learn a model of each option, rather than modelling the overall transition/reward function. An option model encodes the reward and terminal state to be expected when selecting that option in any given state; that is, it consists of two functions $\hat{P}_o : S \times S \mapsto \mathbb{R}$ and $\hat{R}_o : S \mapsto \mathbb{R}$. These option models can be learned in an analogous fashion to standard RL models. When the agent selects option o in state s, it can observe the resulting reward r and terminal state s'. It can use that information to update the models via, e.g., $\Delta\hat{R}_o(s) = \alpha[r - \hat{R}_o(s)]$. Sutton et al. (1999) also describe some model learning techniques that take advantage of the internal structure of the abstract actions, in order to improve upon this basic model learning.

The option model can then be employed in the same sample or planning techniques. That is, the agent can use the option model to internally simulate the effect of selecting an option, either to generate a simulated TD update or to compute the long term value of o. In fact, the benefits of a model-based approach may be even greater in HRL than standard RL. Abstract actions tend to take a long time to execute, because the agent has to move through the environmental execution of several lower level actions. Thus when an agent uses a model of an abstract action to generate samples it is saving even more time than it would when simulating a primitive action. And from the perspective of planning, abstract action models represent larger steps through the state space. This reduces the depth of the plan required to reach the goal, thereby reducing the complexity of the planning process.

However, despite these theoretical advantages, as mentioned the majority of HRL work has been model-free. Sutton et al. (1999) provided some basic demonstrations of model-based option learning, but this has not been explored in much detail since then. Several methods have used a model-based approach to learn options (to be discussed in Section 6.2.6), but then those options are used in a model-free way. Jong and Stone (2009) combined a model-based approach with the MAXQ framework. However, in this case the model was just used to guide exploration, as we suggested in Section 6.2.1 (directing exploration towards unexplored parts of the state space); the reinforcement learning was still model-free. Cao and Ray (2012) used a Bayesian approach to learn a model of the environment and pseudorewards, and then used that model to generate learning updates in the MAXQ framework. This is a hybrid approach, but still showed promising performance. In summary, there is good reason to believe that model-based HRL would have some important functional advantages, and some initial suggestions of how to go about it, but the problem is still largely unexplored.

It is also clear that humans (and other animals) employ model based methods in addition to model-free. For example, Daw et al. (2011) created a task with the structure shown in Figure 6.2. Imagine a subject has been performing this task for a while, and so has learned to perform well. Now suppose they select action 1, get the unlikely outcome of transitioning to state B, and get a higher than expected reward. Model-free RL predicts that this should make them more likely to pick action 1, because they just selected action 1 and got a positive prediction error. Model-based RL predicts that this should make them more likely to pick action 2, because they have an internal model of the environment that says that action 2 is more likely to lead to state B, which is the unexpectedly rewarding state. Daw et al. (2011) found a mixture of both results; different subjects showed different propensities towards model-based versus model-free RL, and various manipulations to the task could also influence the outcome. For example, adding a distractor task induced almost entirely model-free behaviour, presumably because subjects did not have the cognitive resources available for the more complex model-based decision making. Daw et al. (2011) also observed both model-based and model-free prediction errors in the ventral striatum, proportional to the subjects' choice behaviour. See Dolan and Dayan (2013) for a review exploring neural evidence for both model-based and model-free RL. In summary, there are not only computational reasons to be interested in model-based methods; we also need to explore those methods if we want to understand reinforcement learning in the brain.

The downside of model-based approaches is that they trade data efficiency for computation. For example, imagine an agent trying to pick between three available actions. In order to make an optimal choice the agent needs to search through all the possible outcomes of each action (recursively searching through all future actions), calculate the value of each

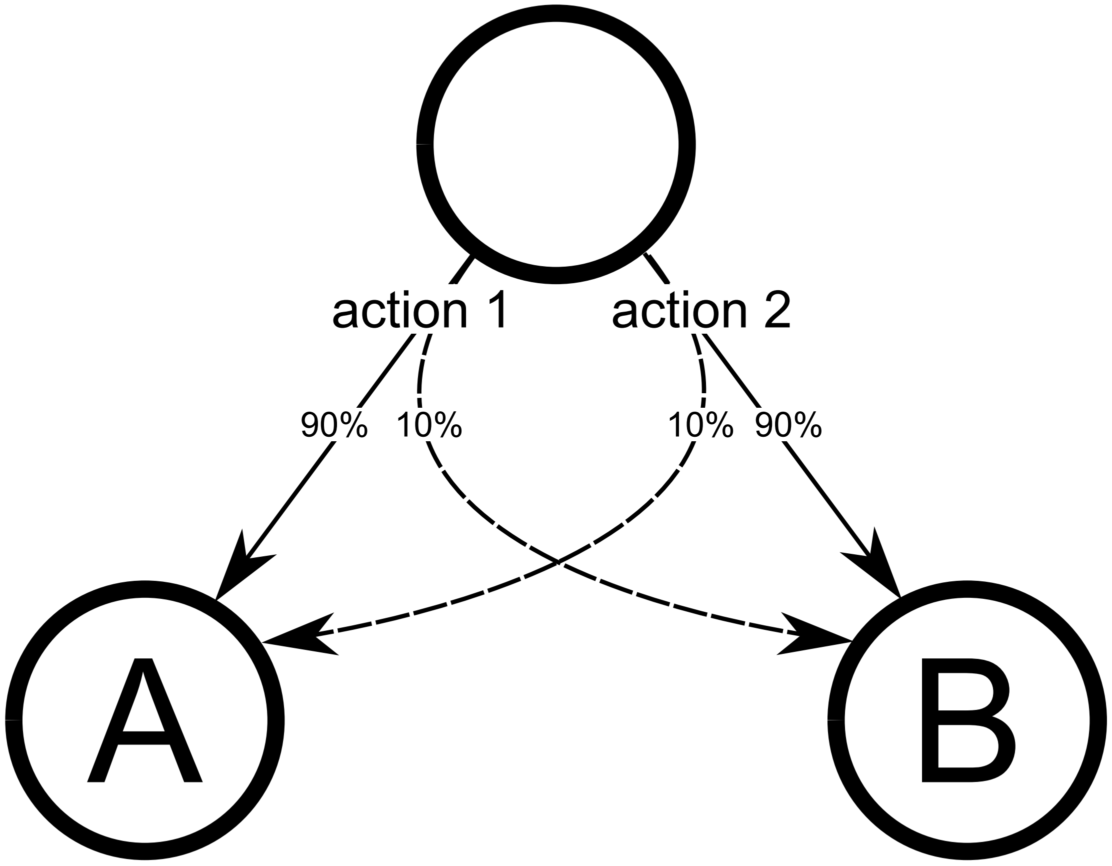


Figure 6.2: Task from Daw et al. (2011). Each action leads to one state with high probability, and the opposite state with low probability.

path, and choose accordingly. Contrast that with a model-free approach, where each of the three actions has a Q value and the optimal behaviour is simply to pick the action with the highest value. In other words, model-based methods make use of data more efficiently by encoding it into these complex representational structures, but decision making then becomes more complex. Model-free approaches put more effort into the learning process, in order to simplify the decision process.

This largely explains why neural models of model-based RL are so few and far between. Learning is relatively well understood in neural networks, so modellers are happy to put the burden on the learning process in order to simplify decision making. In contrast, the complex cognitive control involved in planning is much less understood.

One can imagine representing the transition/reward functions using the standard neural network function approximation methods. For example, we could create a neural population that takes two vectors representing a state and action as input, and outputs a vector representing the predicted future state. These could then be trained using an error signal such as[2]

$$\delta\hat{P}(s,a) = \alpha[s' - \hat{P}(s,a)] \tag{6.2}$$

fed into the standard NEF learning rule (Equation 2.40).

In order to implement the sampling approach, the learned model would essentially take the place of the environment in the current system. The output of the action selection component would be input to the transition/reward function, and the function would output a new state/reward (which would then be used to calculate a prediction error in the normal way). Thus the sampling approach could be implemented without dramatic changes to the existing model.

The real challenge appears when it comes to using the learned model to do planning. How would a neural network implement the complex dynamics of a search process? For example, to search just one path it would have to input a state and action to the population representing the transition function, store the output, feed that stored value back into the transition function at the next stage of the search, and so on. Then it would also need to keep track of the branching tree structure of the search as it moves along the different possible paths, and compute/store the value of each path. The manipulation of complex knowledge structures and careful timing would be key throughout this process, both of which are traditional weaknesses of neural networks.

However, there are NEF models demonstrating similar styles of processing in non-RL domains. In Crawford et al. (2013) a neural model was used to encode a complex linguistic

[2]Note that we have used the deterministic formulation of the transition function here.

knowledge structure, containing 100000 concepts linked by various relations. For example, this model can represent information such as $dog \overrightarrow{isa}\ mammal$ and $mammal \overrightarrow{has}\ fur$. The model can then search through this knowledge structure, in order to find the answer to questions such as “does a dog have fur?”. Clearly there is some distance between this and model-based RL, but it demonstrates that it is possible to control a search process, which could be extended to control a planning process.

In summary, model-based methods are an important but poorly understood aspect of neural RL modelling. Incorporating something like option model learning/planning into this work would be an extensive research project, but there are good reasons to believe that it would be feasible and productive.

6.2.6 Learning hierarchical structure

As mentioned in Section 4.4, in this model we have assumed that the hierarchical structure of the model is defined by the modeller. That is, the modeller decides how the problem is broken down into subtasks, and how the different hierarchical levels interact (by defining the context/state/reward interactions). However, an important goal in reinforcement learning is always to minimize the prior knowledge in the system. Therefore it would be interesting to explore how this model could incorporate more autonomous methods for constructing the hierarchical structure. This is one of the major open problems in HRL, and is an active area of research, especially in regards to understanding how this could occur neurally (Dayan and Niv, 2008; Niv, 2009; Botvinick et al., 2009; Diuk et al., 2012).

One approach is to focus on determining how to automatically generate subgoals. If the subgoals are known then it is fairly easy to generate a template hierarchical structure: we create an abstract action for each subgoal, controlled by a higher level via context interactions that select between subgoals. The options and MAXQ frameworks both specify how to learn the policy for those abstract actions, by using the subgoals to generate pseudoreward (see Section 2.2). The model in the pick-up and delivery task is an example of this kind of structure; in this context the problem is how the model could automatically learn that the pick-up and delivery locations should be two subgoals.

The key to these subgoal based approaches is to build up a representation of the state space, and then analyze that state space to identify useful subgoals. An example of this is the work of McGovern and Barto (2001). Their technique analyzed the agent’s paths through the state space to try to identify bottlenecks—states that the agent has to go through to successfully complete the task. In the pick-up and delivery task the pick-up location would be an example of a bottleneck, because all successful paths have to pass through that state. Their system then created new options aimed at just reaching the bottlenecks, which resulted in faster learning of the original task and better transfer to related tasks. Similar approaches, based on searching for bottlenecks, were pursued by Menache et al. (2002), Mannor et al. (2004), and Simsek and Barto (2004). Note that this relates back to the model-based discussion of the previous section, as learning a model of the environment is key to these techniques.

A different approach is to analyze the policy space rather than the state space. Pickett and Barto (2002) developed a technique that searches a set of optimal policies generated on related problems (tasks with the same state space but different reward function), and attempts to identify commonalities across the different policies. These commonalities are inferred to be useful sub-actions, and they are extracted out and made available to the system as options.

A third group of techniques move away from analyzing the state/policy space, and instead directly search over different hierarchical task structures. For example, Elfwing et al. (2007) used an evolutionary algorithm to explore different hierarchical structures in the MAXQ framework, to find the one that led to the most successful performance. Marthi et al. (2007) used a similar "generate and evaluate" approach to find useful task structures.

The work of Singh et al. (2005) focuses on how to improve the subpolicy learning process once the subgoals are identified. Their technique is based on the concept of intrinsic reward—reward generated internally by the agent, independent of the task reward. Singh et al. (2005) propose that this intrinsic reward is based on novelty; agents receive intrinsic reward when they arrive in a new or unexpected state. The basic idea is that when a novel state is encountered a new option is created with that state as its subgoal (meaning the option has pseudoreward associated with that state). The intrinsic reward is then used to focus exploration around that novel state, allowing the agent to quickly learn the policy for that option. As it learns the policy the target state will cease to be novel; this will cause the intrinsic reward signal to disappear, thereby releasing the agent to move on to learning a new subtask. This again connects to the previous model-based discussion, as the notion of an unexpected state requires that the agent build up an internal model of the expected outcome of its actions.

In order to avoid creating a new option for every new state encountered, this system requires the modeller to predefine certain events as "salient". This is defining what kinds of state change are interesting or important. For example, a movement to a new location in the same room is not usually interesting, even if it is technically a new state. However, if a light comes on when the agent arrives in that location, that is a more important state change. Singh et al. (2005) assume that there is some external system that indicates to the agent when a salient event has occurred. The previous approaches, such as the bottleneck search of McGovern and Barto (2001), can be thought of as methods to automatically define salient states.

The intrinsic reward framework also connects well with neurophysiological data showing that dopamine neurons respond to novelty, independent of any external reward (Horvitz, 2000). This has led to debate over whether the dopamine signal indicates TD error (as described in Section 4.5) or novelty. However, the intrinsic reward framework nicely integrates both of these findings (Mirolli et al., 2013). If novel states are rewarding, then we would expect to see a TD error in those states; thus dopamine neurons can respond to novelty, without that invalidating the TD error hypothesis.

It should be noted that all of these theories are designed for discrete time and space. Some work on continuous subtask learning has been developed by Mugan and Kuipers (2009) and Konidaris and Barto (2009). The work of Mugan and Kuipers is a hybrid approach; they assume that although the agent is operating in continuous space, there is an underlying discrete structure to the problem. Their algorithm focuses on learning a discrete representation while operating in the continuous space, and then once they have found that representation they apply standard discrete reinforcement learning techniques to find a policy. Konidaris and Barto stay in the continuous domain, but their algorithm assumes that the problem can be solved via a sequence of subtasks, where each subtask is responsible for moving the agent through one section of the state space and then passing control off to the next subtask. While they can automatically learn the subtasks in this chain, this technique would not work for learning a full hierarchical structure, where tasks can be nested within one another (their implementation can be thought of as a hierarchy with a maximum depth of two).

None of these techniques have been implemented in a neural model. The most immediate targets for a neural implementation are the subgoal identification approaches, such as the work of McGovern and Barto (2001).[3] The challenge in this case would largely be one of knowledge representation—how to build up a model of the state space, how to track the frequency of state visits/paths, and how to analyze that knowledge in order to identify states with certain characteristics. Note that these issues are very similar to those involved in model-based RL, as described in the previous section. Thus the pursuit of the former might solve many of the problems involved in the latter.

Clearly the description of this implementation is quite abstract. Our purpose in this

[3]The intrinsic motivation work of Singh et al. (2005) is also a very interesting target for a neural implementation, but as mentioned above it requires something like the subgoal identification process as a prerequisite.

section is mainly to point out that automatically learning hierarchical structure is an important open problem, and highlight some of the existing work. Implementing a neural model based on that work would greatly expand the explanatory depth of this model, providing a more complete picture of hierarchical reinforcement learning in the brain. Developing such a model would be the most elaborate, but also one of the most valuable, of the extensions we suggest here.

6.2.7 Improving simulation speed

This extension is not directly related to this model, but refers to neural simulation in general. We include it here because it was a significant constraint on this work, and sheds light on some of the design decisions.

A key challenge in neural modelling is that simulating these biologically detailed models is computationally intensive. For example, the pick-up and delivery model runs at around 1/30th of real time.[4] In practice what this means is that completing just one of the 2.5 hour simulation runs takes 3-4 days of computation. This fact presents a number of challenges for developing these kinds of models. One is that it significantly slows down the design process. Model development is an iterative process, involving trying out different ideas, finding problems, fixing problems, tuning parameters, and so on. When each one of those steps requires many hours of simulation, it slows down the whole development cycle.

These computational limitations also restrict the complexity of the model. The computation cost scales with the number of neurons and connections, so every new feature added to the model slows the model down. Thus the reason for not adding a new neural component is often not that it could not be implemented, but that it would require too many neurons. For example, separating the positive and negative feedback pathways (Section 6.2.2) would primarily involve the duplication of several components; this is not difficult from an implementation perspective, but would dramatically slow down the model.

Similarly, these constraints can prevent the model from being applied to more complex tasks. For example, the pick-up and delivery task is played out in a fairly small environment. Performing the same task in a larger, multi-room environment could be done without changing the model's implementation, but it would require more place cells and therefore more neurons in the state population. As another example, the slow simulation speed prevents the model from being applied in real-time tasks, such as robotic applications; it is difficult to control a robot when the control is occurring at 1/30th of real time.

[4]On a Dual Intel Xeon E5-2650 2GHz CPU.

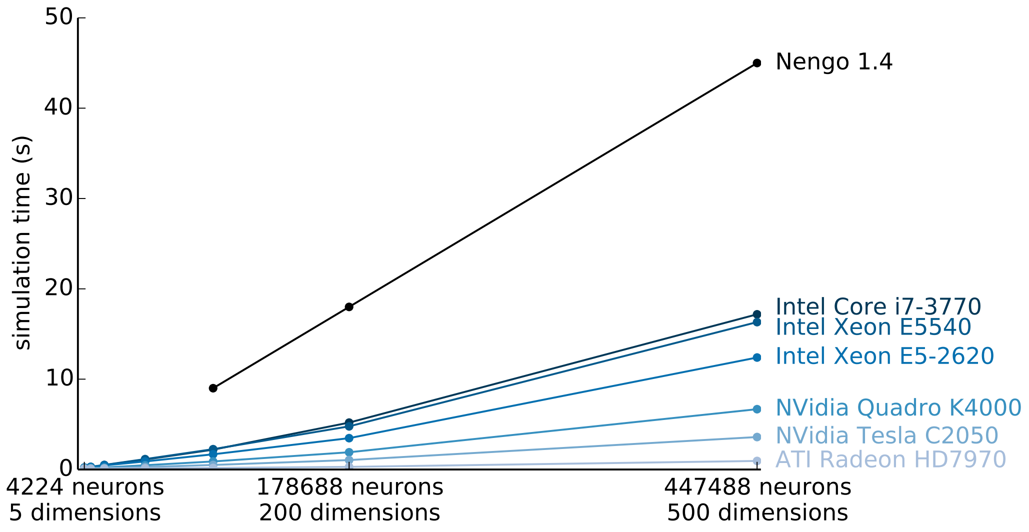


Figure 6.3: Simulation speed comparison between Nengo 1.4 and Nengo 2.0 running on various different platforms. The model in this case involved a discrete Fourier transform and an element-wise multiplication, run for 1 second of simulated time. Figure from Bekolay et al. (2014).

As mentioned in Section 2.3.5, the model is simulated in a software suite called Nengo. Recognizing these difficulties, improving the simulation speed of Nengo has been an important research focus. As we developed the model presented here, we have also put extensive effort into rewriting Nengo to support more efficient simulation of large models.

The basic idea is to allow Nengo models to run on a wide range of high performance platforms, such as neuromorphic hardware, Blue Gene supercomputers, and GPU clusters. Supporting this functionality required rewriting Nengo essentially from the ground up, but the effort has already shown promise (Bekolay et al., 2014). Figure 6.3 compares the performance of a simple Nengo circuit on several different platforms, showing significant improvements in the new software. Note that Figure 6.3 shows results from the most standard computing platforms; even larger improvements are to be expected when simulating on Blue Gene or neuromorphic hardware, although those benchmarks are not yet available.

Unfortunately this redesign is still a work in progress, so the current project was not able to benefit from these improvements; all of the work presented here was carried out in Nengo 1.4. Thus an important next step is to complete the development of Nengo 2.0 and then rewrite this model in the new environment. Functionally the two code bases are equivalent, so none of the behavioural results will change; it is mainly a matter of changing the code over to the new syntax of Nengo 2.0. This will allow the model to take advantage of the improved simulation speed, as well as future improvements to Nengo. The further development of the model will be greatly eased by these improvements, as well as allowing it to scale up to increasingly complex problems.

6.3 Model predictions

A great advantage of building mechanistic neural models is that they can be used to generate a wealth of comparisons to experimental data. There are two main approaches to these comparisons. One is to look at existing data and try to reproduce it with the model (as in Section 5.4). This can verify the biological plausibility of the model, or generate a new explanation for previously unexplained results (as in the classic dopamine/TD error work of Schultz 1998).

The other approach is to generate predictions from the model; in this case the modeller is using the analogy between the model and the modelled system to anticipate data that does not yet exist. Prediction serves the same basic purpose as data matching: it supports the biological plausibility of the model (if the prediction is born out), and it can generate new understanding of the modelled system. However, prediction can be much more effective in both these regards than data matching. A verified prediction is more convincing evidence of the biological plausibility of a model, as the prediction is out of the modellers hands; they cannot adjust the model to fit the data, they must fix the model and commit to the result. Similarly, when a modelling prediction is investigated experimentally it has the potential to drive research in an entirely new direction, which can have more impact than providing a new explanation for existing data.

However, along with these potential advantages comes the condition that generating useful predictions from a model is more difficult than data matching. The caveat of *useful* predictions is an important one. It is easy to generate predictions; for example, we could take any of the neural populations in this model and "predict" that there is a corresponding neural population in the brain. There are several conditions to a useful prediction. One is that the prediction should be testable; there needs to be a way to collect the predicted data using existing methods, and quantitatively compare that data to the model's prediction. For example, how would we find the set of neurons corresponding to one of the predicted populations, and how would we verify that it did correspond to the model population even if we did? It should also be the case that the prediction is not trivially true. It is quite likely that we could find some group of neurons whose activity would correlate with some neurons in the model, but that is not a convincing demonstration of their equivalence. In other words, the prediction should distinguish something unique to the model, and not be a result that would also follow from a broad range of competing hypotheses.

The final challenge of predictions is that someone needs to actually conduct the experimental investigation. It is often the fate of modelling predictions that they are cast out by the modeller and then never taken up by an experimentalist, in which case they are not of much use. Of course this is often out of the modeller's hands; unless the modeller has the experimental apparatus themselves all they can do is make the prediction as compelling as possible, in the hope of convincing an experimentalist to invest their effort. Often this can be an important benefit of data matching, as a model that has already been shown to match existing data has more support for the plausibility of its predictions.

In this section we present some of the predictions that can be generated from this model. This is not an exhaustive list, but these are the predictions that best meet the criteria outlined above.

6.3.1 Testing the dual training system

In Section 4.5 we described a hypothesized neuroanatomical mapping for the dual training system, and mentioned that while it was plausible it was also rather speculative and untested. It is useful then to discuss how one might investigate that theory experimentally.

The key prediction is that updates in the dorsal striatum (representing the current Q function) should be time delayed relative to those in the ventral striatum. For example, imagine a rodent in a T-maze, well-trained so that there are little to no prediction error signals. If an unexpected reward were then placed in one arm of the maze, we would expect a positive prediction error in ventral striatum, as normal. The unique prediction of the dual training system is that we would then see a similar positive prediction error in dorsal striatum, not concurrent with the first but closely following it.

The length of the time delay between the updates depends on how quickly the animal updates the data stored in orbitofrontal cortex (OFC). Recall that the current Q function can only be updated when the current and stored state are the same, which occurs immediately after the stored state is updated. Thus we would expect to see the following sequence: reward received, prediction error in ventral striatum, activity change in OFC (not necessarily correlated with prediction error), and finally prediction error in dorsal striatum. A more dramatic delay could be observed if the animal were moved out of the rewarded state before the OFC update occurred. Then we would not expect a prediction error in dorsal striatum until the next time the animal reaches the rewarded state. However, given the rapidity of the OFC updates, it would likely be difficult to interrupt the update sequence in this way.

The rapidity of these updates would also make it difficult to detect these timings with fMRI, due to its lower temporal resolution. Thus this experiment would be best applied in an animal model as described here, with direct electrophysiological recording. However, the challenge in that case would be to record from those diverse areas (dorsal/ventral striatum and OFC) in the same animal.

6.3.2 Integrative discount

In Section 4.5.3 we pointed out that there was no strong evidence either for or against the integrative discount mechanism. It is helpful then to imagine what convincing evidence would look like, and how it might be found.

The key signature of the integrative discount would be neural activity that increases (or decreases) over the course of action execution. Note that even though the integrated value increases, the NEF demonstrates how this can be represented neurally by a decrease in firing rate. Either way, the important point is that the activity changes monotonically over time, even in the absence of any external changes.

More specifically, the model predicts that the rate of change should be proportional to the value of the previously selected action. We have already seen that researchers can find neurons representing action values (e.g., Samejima et al., 2005), so those could be used to get a measure of the previous value. The next step would be to search for neurons whose activity changes monotonically over time, as described above. The final test would then be to look for a correlation between the two—as the action value goes up or down, the rate of change should go up or down accordingly.

The main problem with this prediction is that we do not really know where to look for these integrative discount neurons. The basic prediction is just that there are neurons somewhere in the brain with these properties, which is not very helpful for an experimentalist. In addition, the integrative discount can be computed by a fairly small population, which may be difficult to detect with methods such as fMRI. Thus testing this prediction may require electrophysiological recording, which is even more difficult to use in a broad search.

However, it seems likely that these neurons will be closely associated with the dopaminergic nuclei, since their output is a key factor in the TD error calculation. Thus we would expect them to be either inside the dopaminergic nuclei or within one synapse. In addition, these neurons need to receive input from the neurons representing action values in ventral striatum. This narrows down the search range somewhat, but still represents a difficult challenge.

6.3.3 Rule bias

Another prediction of the model, and one that could be tested in human subjects, arose in relation to the Badre et al. (2010) hierarchical stimuli task. Namely, the model predicts that the initial propensity of the pre-premotor region to a high activity level is due to an internally generated bias applied to the reward signal.

Experimentally, this would appear as stronger positive prediction errors and weaker negative prediction errors on the hierarchical version of the task relative to the flat version. This could be used to create a measure of the bias in each subject, for example by calculating the ratio between the average positive prediction error in the hierarchical and flat scenarios.

The model would then predict that that bias measure would correlate with the magnitude of the high activity bump in the pre-premotor region in the flat condition. Specifically, the bias should be correlated with the width of the bump—the stronger the subject's bias, the longer they should persist in trying to find a hierarchical rule.

One nice feature of this prediction is that it can likely be tested simply by re-analyzing existing data, rather than requiring a new experiment to be conducted. However, the question will be whether the measurements are sensitive enough to detect the hypothesized difference in prediction errors. As described in Section 5.4, even a bias as small as 10% in the reward signal can induce the hierarchical bias, which may translate into an even smaller difference in prediction errors.

Chapter 7

Conclusion

In this work we have presented a neural model capable of performing hierarchical reinforcement learning. This model is able to leverage its hierarchical structure to achieve improved performance, and is consistent with neurophysiological data. It thus represents a new hypothesis for neural decision making, bringing us closer to understanding the brain's impressive ability to learn complex behaviour from sparse feedback.

The model implements all the major components of reinforcement learning via neural mechanisms, including the learning of action values (Section 4.1), action selection (Section 4.2), and TD error calculation (Section 4.3). In addition, all of these components are designed to operate in an SMDP environment, where actions have temporal extent. This allows us to extend the model to hierarchical domains (Section 4.4), where the abstract actions of HRL must extend across the execution of the underlying basic actions.

Throughout the design of this model an important goal has been to develop a generic reinforcement learning system that can be applied across a wide range of tasks. Thus a strong effort has been made to keep task-specific information/optimizations out of the model's basic components. The model treats the environment as a black box, only interacting with it via the state and action vectors (and with a scalar reward signal as the only feedback). This means that when the model is applied to a new task its internal structure is largely unchanged; all that needs to be modified are those input and output components.

The hierarchical structure of the model takes a similar approach; we have tried to define this structure in as general a fashion as possible, via the three context, state, and reward interactions. However, these are less general, in that the modeller must implement a specific instantiation of these abstract categories based on the hierarchical structure of the task. Automating this process is an important direction for future development of this model (as discussed in Section 6.2.6).

We presented a detailed neuroanatomical mapping for the model in Section 4.5. This serves two purposes. First, it lends support to the biological plausibility of the model, by demonstrating that the structure of the model is consistent with known neuroanatomical structure. Second, it makes it easier to connect the results of the model to neurophysiological data. This can be used to verify that the model matches existing data (e.g., Section 5.4), or to generate new predictions (e.g., Section 6.3).

This model builds on previous work in a number of ways. First we can consider the model's contributions from the perspective of standard RL (ignoring the hierarchical framework). Many previous models have been limited to "associative RL", meaning that they can learn to maximize immediate rewards but not the long term consequences of their decisions (Section 3.1). Other models move beyond immediate rewards, but are still limited to optimizing relatively short-term consequences. In this model we show how the full TD learning process can be implemented neurally, thereby allowing the model to solve problems involving long sequences of decision making. Although this is not the only model to do so, it is one of few (Section 3.1.5), and the only one able to operate in a continuous/SMDP environment. We also believe that the distributed representational structure of this model will scale better to complex problem domains, although a convincing demonstration of that scaling awaits improvements to model simulation speed (see Section 6.2.7).

With respect to hierarchical reinforcement learning, this is the first neural model to implement an HRL framework. Previous work has described how HRL could be neurally implemented in theory only, or has implemented a simplified, associative HRL structure (see Section 3.2). Thus this is the first model to demonstrate that the computational principles of HRL could be implemented in the brain. This is exciting, as it represents a new hypothesis for the brain's reinforcement learning mechanisms. This new hypothesis has greater functional power than flat RL techniques, thus explaining a broader range of the brain's performance.

The results we presented in Chapter 5 were designed to reinforce all the above points. We demonstrated the model's basic functional performance in Sections 5.1 and 5.2. Section 5.3 established the power of the hierarchical approach, by demonstrating the improved performance of the hierarchical versus flat model on the delivery task, and the ability of the model to easily transfer knowledge between tasks. The hierarchical stimuli task of Section 5.4 focused more on the comparison to experimental data; although such data is limited in HRL, the results showed that the model's output was consistent with human neurophysiological data when applied to the same task.

However, with all of the positive outcomes outlined above, it is important to emphasize that this model has many limitations that could be improved upon. For example, it has only been applied to relatively simple environments. These tasks may be complex with respect to existing neural modelling, but they are still a far cry from the tasks that humans can solve effortlessly, not to mention ones we find difficult. Understanding that advanced performance is one of the main motivations for building these kinds of models, thus being limited to simple domains is somewhat unsatisfying.

Beyond scaling, there are also important qualitative gaps in this model. One of the most critical is model-based processing (Section 6.2.5). This is undoubtedly a crucial aspect of human reinforcement learning, yet all of the processing in this work is model-free. Understanding how to implement model-based RL in a neural model, how to apply those techniques to HRL, and how to combine such a system with the one we present here are important and unanswered questions.

Another key gap is the autonomous learning of hierarchical structure. In this model the hierarchical structure (consisting of the various state/context/reward interactions, as well as the division of abstract actions and assignment of actions to different hierarchical levels) is all created by the modeller. One of the primary goals of RL is to begin from a *tabula rasa* state, thus this dependence on the modeller is unsatisfying. In addition, real neural systems do not have anyone to create the hierarchical structure for them, it must be learned dynamically; thus this model must be omitting important neural computations. As discussed in Section 6.2.6, techniques to autonomously learn hierarchical structure are an ongoing area of research in HRL, and incorporating these ideas into a neural model will be an important step in understanding hierarchical processing in the brain.

Thus the conclusions of this work are both optimistic and challenging. On the one hand, we have seen that HRL can be implemented in a neural model, and seen demonstrations of the added power that framework can bring to neural modelling. On the other hand, this work also highlights the large distance still separating our models from real neural systems. Our hope is that this model closes that distance slightly, bringing us a little closer to understanding and recreating the impressive learning abilities of the brain.

References

Andre, D. and Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *Proceedings of the 18th National Conference on Artificial Intelligence*, pages 119–125. AAAI Press.

Baddeley, B. (2008). Reinforcement learning in continuous time and space: interference and not ill conditioning is the main problem when using distributed function approximators. *IEEE Transactions on Systems, Man, and Cybernetics. Part B, Cybernetics*, 38(4):950–6.

Badre, D. and Frank, M. J. (2012). Mechanisms of hierarchical reinforcement learning in cortico-striatal circuits 2: evidence from fMRI. *Cerebral Cortex*, 22(3):527–36.

Badre, D., Kayser, A. S., and D'Esposito, M. (2010). Frontal cortex and the discovery of abstract action rules. *Neuron*, 66(2):315–26.

Bakker, B. and Schmidhuber, J. (2004). Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proceedings of the 80th Conference on Intelligent Autonomous Systems*, pages 438–445.

Baras, D. and Meir, R. (2007). Reinforcement learning, spike-time-dependent plasticity, and the BCM rule. *Neural Computation*, 19(8):2245–79.

Barto, A. G., Konidaris, G., and Vigorito, C. (2013). Behavioral Hierarchy: Exploration and Representation. In Baldassare, G. and Mirolli, M., editors, *Computational and Robotic Models of the Hierarchical Organization of Behavior*, chapter 3, pages 1–39. Springer-Verlag, Berlin.

Barto, A. G. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, pages 1–28.

Barto, A. G., Sutton, R. S., and Anderson, C. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13(5):834–846.

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., Choo, X., Voelker, A. R., and Eliasmith, C. (2014). Nengo: a Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7(48):1–13.

Bellman, R. E. (1957). *Dynamic Programming.* Princeton University Press, Princeton.

Botvinick, M. M., Niv, Y., and Barto, A. G. (2009). Hierarchically organized behavior and its neural foundations: a reinforcement learning perspective. *Cognition*, 113(3):262–80.

Bradtke, S. J. and Duff, M. O. (1995). Reinforcement learning methods for continuous-time Markov decision problems. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems*, San Mateo. Morgan Kaufmann.

Busoniu, L., Babuska, R., Schutter, B. D., and Ernst, D. (2010). *Reinforcement learning and dynamic programming using function approximators.* CRC Press.

Cao, F. and Ray, S. (2012). Bayesian Hierarchical Reinforcement Learning. In *Advances in Neural Information Processing Systems*, pages 1–9.

Choo, X. and Eliasmith, C. (2013). General Instruction Following in a Large-Scale Biologically Plausible Brain Model. In *Proceedings of the 35th Annual Conference of the Cognitive Science Society*, pages 322–327.

Crawford, E., Gingerich, M., and Eliasmith, C. (2013). Biologically plausible, human-scale knowledge representation. In *Proceedings of the 35th Annual Conference of the Cognitive Science Society*, pages 412–417, Austin. Cognitive Science Society.

Critchfield, T. S. and Kollins, S. H. (2001). Temporal discounting: basic research and the analysis of socially important behavior. *Journal of Applied Behavior Analysis*, 34(1):101–22.

Cuayáhuitl, H., Kruijff-Korbayová, I., and Dethlefs, N. (2012). Hierarchical Dialogue Policy Learning Using Flexible State Transitions and Linear Function Approximation. In *Proceedings of the International Conference on Computational Linguistics.*

Daw, N. D., Gershman, S. J., Seymour, B., Dayan, P., and Dolan, R. J. (2011). Model-based influences on humans' choices and striatal prediction errors. *Neuron*, 69(6):1204–15.

Dayan, P. and Hinton, G. (1993). Feudal reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 271–278.

Dayan, P. and Niv, Y. (2008). Reinforcement learning: the good, the bad and the ugly. *Current Opinion in Neurobiology*, 18(2):185–96.

DeWolf, T. and Eliasmith, C. (2013). A neural model of the development of expertise. In *Proceedings of the 12th International Conference on Cognitive Modelling*.

Dietterich, T. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

Diuk, C., Schapiro, A., Cordova, N., Niv, Y., and Botvinick, M. M. (2012). Divide and conquer: hierarchical reinforcement learning and task decomposition in humans. In Baldassare, G. and Mirolli, M., editors, *Computational and Robotic Models of the Hierarchical Organization of Behavior*, page in press. Springer-Verlag.

Dolan, R. J. and Dayan, P. (2013). Goals and Habits in the Brain. *Neuron*, 80(2):312–325.

Doya, K. (2000). Reinforcement learning in continuous time and space. *Neural Computation*, 12(1):219–45.

Eichenbaum, H., Dudchenko, P., Wood, E., Shapiro, M., and Tanila, H. (1999). The hippocampus, memory, and place cells: is it spatial memory or a memory space? *Neuron*, 23(2):209–26.

Elfwing, S., Uchibe, E., Doya, K., and Christensen, H. (2007). Evolutionary development of hierarchical learning structures. *IEEE Transactions on Evolutionary Computation*, 11(2):249–264.

Eliasmith, C. and Anderson, C. (2003). *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT Press, Cambridge.

Eliasmith, C. and Martens, J. (2011). Normalization for probabilistic inference with neurons. *Biological Cybernetics*, 104(4-5):251–62.

Eliasmith, C., Stewart, T. C., Choo, X., Bekolay, T., DeWolf, T., Tang, Y., and Rasmussen, D. (2012). A large-scale model of the functioning brain. *Science*, 338(6111):1202–1205.

Felleman, D. J. and Van Essen, D. C. (1991). Distributed hierarchical processing in the primate cerebral cortex. *Cerebral Cortex*, 1(1):1–47.

Florian, R. V. (2007). Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19(6):1468–502.

Frank, M. J. and Badre, D. (2012). Mechanisms of hierarchical reinforcement learning in corticostriatal circuits 1: computational analysis. *Cerebral Cortex*, 22(3):509–26.

Frank, M. J., Loughry, B., and O'Reilly, R. C. (2001). Interactions between frontal cortex and basal ganglia in working memory: a computational model. *Cognitive, Affective & Behavioral Neuroscience*, 1(2):137–60.

Frémaux, N., Sprekeler, H., and Gerstner, W. (2013). Reinforcement Learning Using a Continuous Time Actor-Critic Framework with Spiking Neurons. *PLoS Computational Biology*, 9(4):e1003024.

Friedrich, J., Urbanczik, R., and Senn, W. (2011). Spatio-temporal credit assignment in neuronal population learning. *PLoS Computational Biology*, 7(6):e1002092.

Fujii, N. and Graybiel, A. M. (2003). Representation of action sequence boundaries by macaque prefrontal cortical neurons. *Science*, 301(5637):1246–9.

Galluppi, F., Davies, S., Furber, S., Stewart, T., and Eliasmith, C. (2012). Real time on-chip implementation of dynamical systems with spiking neurons. *The 2012 International Joint Conference on Neural Networks*, pages 1–8.

Galluppi, F., Denk, C., Meiner, M., Stewart, T., Plana, L., Eliasmith, C., Furber, S., and Conradt, J. (2014). Event-based neural computing on an autonomous mobile platform. In *Proceedings of IEEE International Conference on Robotics and Automation*.

Ghavamzadeh, M. and Mahadevan, S. (2001). Continuous-time hierarchical reinforcement learning. In *Proceedings of the 18th International Conference on Machine Learning*, pages 186–193. Morgan Kaufmann.

Golde, M., von Cramon, D. Y., and Schubotz, R. I. (2010). Differential role of anterior prefrontal and premotor cortex in the processing of relational information. *NeuroImage*, 49(3):2890–900.

Groenewegen, H., der Zee, E., Kortschot, A. T., and Witter, M. (1987). Organization of the projections from the subiculum to the ventral striatum in the rat. A study using anterograde transport of Phaseolus vulgaris leucoagglutinin. *Neuroscience*, 23(I):103–120.

Gurney, K., Prescott, T. J., and Redgrave, P. (2001). A computational model of action selection in the basal ganglia. I. A new functional anatomy. *Biological Cybernetics*, 84(6):401–10.

Haruno, M. and Kawato, M. (2006). Heterarchical reinforcement-learning model for integration of multiple cortico-striatal loops: fMRI examination in stimulus-action-reward association learning. *Neural Networks*, 19(8):1242–54.

Hasselmo, M. (2005). A model of prefrontal cortical mechanisms for goal-directed behavior. *Journal of Cognitive Neuroscience*, 17(7):1115–1129.

Hawasly, M. and Ramamoorthy, S. (2013). Lifelong Learning of Structure in the Space of Policies. In *Proceedings of the AAAI Spring Symposium on Lifelong Machine Learning*.

Hinton, G. E., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554.

Holroyd, C. B. and Yeung, N. (2012). Motivation of extended behaviors by anterior cingulate cortex. *Trends in Cognitive Sciences*, 16(2):122–8.

Horvitz, J. (2000). Mesolimbocortical and nigrostriatal dopamine responses to salient nonreward events. *Neuroscience*, 96(4):651–656.

Howard, R. A. (1960). *Dynamic programming and Markov processes*. MIT Press, Cambridge.

Howard, R. A. (1971). *Dynamic Probabilistic Systems*. Dover Publications.

Izhikevich, E. M. (2007). Solving the distal reward problem through linkage of STDP and dopamine signaling. *Cerebral Cortex*, 17(10):2443–52.

Joel, D., Niv, Y., and Ruppin, E. (2002). Actor-critic models of the basal ganglia: new anatomical and computational perspectives. *Neural Networks*, 15(4-6):535–47.

Jong, N. and Stone, P. (2009). Compositional models for reinforcement learning. In *European Conference on Machine Learning*, Bled.

Joshi, M., Khobragade, R., Sarda, S., Deshpande, U., and Mohan, S. (2012). Hierarchical Action Selection for Reinforcement Learning in Infinite Mario. In Kersting, K. and Toussaint, M., editors, *Sixth Starting Artificial Intelligence Research Symposium*, pages 162–167. IOS Press.

Khan, M., Lester, D., and Plana, L. (2008). SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *IEEE Joint Conference on Neural Networks*, pages 2849–2856.

Kim, S., Hwang, J., and Lee, D. (2008). Prefrontal coding of temporally discounted values during intertemporal choice. *Neuron*, 59(1):161–72.

Koenig, S. and Simmons, R. G. (1993). Complexity Analysis of Real-Time Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 99–105.

Konidaris, G. and Barto, A. G. (2009). Skill discovery in continuous reinforcement learning domains using skill chaining. *Advances in Neural Information Processing Systems*, pages 1–9.

Lammel, S., Lim, B. K., Ran, C., Huang, K. W., Betley, M. J., Tye, K. M., Deisseroth, K., and Malenka, R. C. (2012). Input-specific control of reward and aversion in the ventral tegmental area. *Nature*, 491(7423):212–7.

Lapicque, L. (1907). Recherches quantitatives sur lexcitation électrique des nerfs traitée comme une polarisation. *Journal de Physiologie et de Pathologie Générale*, 9:620–635.

Lawson, C. L. and Hanson, R. J. (1974). *Solving least squares problems.* Prentice-Hall, Englewood Cliffs.

Liu, D. and Todorov, E. (2009). Hierarchical optimal control of a 7-DOF arm model. In *2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 50–57. Ieee.

MacNeil, D. and Eliasmith, C. (2011). Fine-tuning and the stability of recurrent neural networks. *PloS ONE*, 6(9):e22885.

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 38.

Mannor, S., Menache, I., Hoze, A., and Klein, U. (2004). Dynamic abstraction in reinforcement learning via clustering. *21st International Conference on Machine Learning*, page 71.

Markram, H., Lubke, J., Frotscher, M., and Sakmann, B. (1997). Regulation of Synaptic Efficacy by Coincidence of Postsynaptic APs and EPSPs. *Science*, 275(5297):213–215.

Marthi, B., Kaelbling, L., and Lozano-Perez, T. (2007). Learning hierarchical structure in policies. In *NIPS Workshop on Hierarchical Organization of Behavior*.

McGovern, A. and Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the 18th International Conference on Machine Learning*, pages 361–368. Morgan Kaufmann.

Menache, I., Mannor, S., and Shimkin, N. (2002). Q-cut — dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the 13th European Conference on Machine Learning*, pages 295–306, London. Springer-Verlag.

Menache, I., Mannor, S., and Shimkin, N. (2005). Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, 134(1):215–238.

Merolla, P. and Arthur, J. (2007). Expandable networks for neuromorphic chips. *IEEE Transactions on Circuits and Systems*, 54(2):301–311.

Millán, J., Posenato, D., and Dedieu, E. (2002). Continuous-action Q-learning. *Machine Learning*, pages 247–265.

Miller, E. K. and Cohen, J. D. (2001). An integrative theory of prefrontal cortex function. *Annual Review of Neuroscience*, 24:167–202.

Mirolli, M., Santucci, V. G., and Baldassarre, G. (2013). Phasic dopamine as a prediction error of intrinsic and extrinsic reinforcements driving both action acquisition and reward maximization: A simulated robotic study. *Neural Networks*, 39:40–51.

Morimoto, J. and Doya, K. (2001). Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. *Robotics and Autonomous Systems*, 36(1):37–51.

Morris, G., Nevet, A., Arkadir, D., Vaadia, E., and Bergman, H. (2006). Midbrain dopamine neurons encode decisions for future action. *Nature Neuroscience*, 9(8):1057–63.

Mugan, J. and Kuipers, B. J. (2009). Autonomously learning an action hierarchy using a learned qualitative state representation. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Neumann, G., Maass, W., and Peters, J. (2009). Learning complex motions by sequencing simpler motion templates. *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1–8.

Niv, Y. (2009). Reinforcement learning in the brain. *Journal of Mathematical Psychology*, pages 1–38.

O'Doherty, J., Dayan, P., Schultz, J., Deichmann, R., Friston, K., and Dolan, R. J. (2004). Dissociable roles of ventral and dorsal striatum in instrumental conditioning. *Science*, 304(5669):452–4.

O'Keefe, J. and Dostrovsky, J. (1971). The hippocampus as a spatial map. Preliminary evidence from unit activity in the freely-moving rat. *Brain Research*, 34:171–175.

O'Reilly, R. C. and Frank, M. J. (2006). Making working memory work: a computational model of learning in the prefrontal cortex and basal ganglia. *Neural Computation*, 18(2):283–328.

O'Reilly, R. C., Frank, M. J., Hazy, T. E., and Watz, B. (2007). PVLV: the primary value and learned value Pavlovian learning algorithm. *Behavioral Neuroscience*, 121(1):31–49.

Parr, R. and Russell, S. J. (1998). Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*.

Parsons, M., Li, S., and Kirouac, G. (2007). Functional and anatomical connection between the paraventricular nucleus of the thalamus and dopamine fibers of the nucleus accumbens. *Journal of Comparative Neurology*, 1063(April):1050–1063.

Pessiglione, M., Seymour, B., Flandin, G., Dolan, R. J., and Frith, C. D. (2006). Dopamine-dependent prediction errors underpin reward-seeking behaviour in humans. *Nature*, 442(7106):1042–5.

Pickett, M. and Barto, A. G. (2002). PolicyBlocks: An algorithm for creating useful macro-actions in reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 506–513. Morgan Kaufmann.

Polsky, A., Mel, B. W., and Schiller, J. (2004). Computational subunits in thin dendrites of pyramidal cells. *Nature Neuroscience*, 7(6):621–7.

Potjans, W., Morrison, A., and Diesmann, M. (2009). A spiking neural network model of an actor-critic learning agent. *Neural Computation*, 339:301–339.

Precup, D., Sutton, R. S., and Dasgupta, S. (2001). Off-policy temporal-difference learning with function approximation. In *Proceedings of the 18th International Conference on Machine Learning*.

Provost, J., Kuipers, B. J., and Miikkulainen, R. (2007). Self-Organizing Distinctive State Abstraction Using Options. In *Proceedings of the 7th International Conference on Epigenetic Robotics*.

Puterman, M. L. (1994). *Markov Decision Processes*. John Wiley & Sons, Inc., New York.

Randlov, J. and Alstrom, P. (1998). Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the 15th International Conference on Machine Learning*, pages 463–471, Madison.

Rasmussen, D. and Eliasmith, C. (2013). A neural reinforcement learning model for tasks with unknown time delays. In Knauff, M., Pauen, M., Sebanz, N., and Wachsmuth, I., editors, *Proceedings of the 35th Annual Conference of the Cognitive Science Society*, pages 3257–3262, Austin. Cognitive Science Society.

Rasmussen, D. and Eliasmith, C. (2014a). A neural model of hierarchical reinforcement learning. In Bello, P., Guarini, M., McShane, M., and Scassellati, B., editors, *Proceedings of the 36th Annual Conference of the Cognitive Science Society*, pages 1252–1257, Austin. Cognitive Science Society.

Rasmussen, D. and Eliasmith, C. (2014b). A spiking neural model applied to the study of human performance and cognitive decline on Raven's Advanced Progressive Matrices. *Intelligence*, 42:53–82.

Redgrave, P., Prescott, T. J., and Gurney, K. (1999). The basal ganglia: a vertebrate solution to the selection problem? *Neuroscience*, 89(4):1009–1024.

Rescorla, R. and Wagner, A. (1972). A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement. In Black, A. H. and F, P. W., editors, *Classical Conditioning II: Current research and theory*, chapter 3, pages 64–99. Appleton-Century-Crofts, New York.

Reynolds, J. N., Hyland, B. I., and Wickens, J. R. (2001). A cellular mechanism of reward-related learning. *Nature*, 413(6851):67–70.

Ribas-Fernandes, J. J. F., Solway, A., Diuk, C., McGuire, J. T., Barto, A. G., Niv, Y., and Botvinick, M. M. (2011). A neural signature of hierarchical reinforcement learning. *Neuron*, 71(2):370–9.

Roesch, M. R., Calu, D. J., and Schoenbaum, G. (2007). Dopamine neurons encode the better option in rats deciding between differently delayed or sized rewards. *Nature Neuroscience*, 10(12):1615–24.

Rummery, G. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical Report September, Cambridge University.

Samejima, K., Ueda, Y., Doya, K., and Kimura, M. (2005). Representation of action-specific reward values in the striatum. *Science*, 310(5752):1337–40.

Schultz, W. (1998). Predictive reward signal of dopamine neurons. *Journal of Neurophysiology*, 80:1–27.

Schultz, W., Tremblay, L., and Hollerman, J. R. (2000). Reward processing in primate orbitofrontal cortex and basal ganglia. *Cerebral Cortex*, 10(3):272–84.

Seung, H. S. (2003). Learning in spiking neural networks by reinforcement of stochastic synaptic transmission. *Neuron*, 40(6):1063–73.

Simsek, O. and Barto, A. G. (2004). Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *21st International Conference on Machine Learning*, page 95, New York. ACM Press.

Singh, S. (1992). Transfer of learning by composing solutions of elemental sequential tasks. In *Machine Learning*, pages 323–339.

Singh, S. (1994). Reinforcement learning algorithms for average-payoff Markovian decision processes. In *Proceedings of the 12th AAAI*, pages 700–705. MIT Press.

Singh, S., Barto, A. G., and Chentanez, N. (2005). Intrinsically motivated reinforcement learning. In Saul, L. K., Weiss, Y., and Bottou, L., editors, *Advances in Neural Information Processing Systems*, pages 1281–1288. MIT Press.

Stewart, T. and Eliasmith, C. (2011). Neural cognitive modelling: A biologically constrained spiking neuron model of the Tower of Hanoi task. In *Proceedings of the 33rd Annual Conference of the Cognitive Science Society*, pages 656–661.

Stewart, T. C., Bekolay, T., and Eliasmith, C. (2012). Learning to select actions with spiking neurons in the Basal Ganglia. *Frontiers in Decision Neuroscience*, 6:2.

Stewart, T. C., Choo, X., and Eliasmith, C. (2010). Dynamic behaviour of a spiking model of action selection in the basal ganglia. In Ohlsson, S. and Catrambone, R., editors, *Proceedings of the 32nd Annual Conference of the Cognitive Science Society*, pages 235–240, Austin. Cognitive Science Society.

Stewart, T. C., Tripp, B., and Eliasmith, C. (2009). Python scripting in the nengo simulator. *Frontiers in Neuroinformatics*, 3(March):7.

Strösslin, T. and Gerstner, W. (2003). Reinforcement learning in continuous state and action space. In *International Conference on Artificial Neural Networks*.

Stuart, G., Spruston, N., Sakmann, B., and Häusser, M. (1997). Action potential initiation and backpropagation in neurons of the mammalian CNS. *Trends in Neurosciences*, 20(3):125–31.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning.* MIT Press, Cambridge.

Sutton, R. S., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211.

Taylor, M. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10:1633–1685.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8(3-4):257–277.

Tsitsiklis, J. and Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690.

Urbanczik, R. and Senn, W. (2009). Reinforcement learning in populations of spiking neurons. *Nature Neuroscience*, 12(3):250–2.

Vasilaki, E., Frémaux, N., Urbanczik, R., Senn, W., and Gerstner, W. (2009). Spike-based reinforcement learning in continuous state and action space: when policy gradient methods fail. *PLoS Computational Biology*, 5(12):e1000586.

Voelker, A. R., Crawford, E., and Eliasmith, C. (2014). Learning large-scale heteroassociative memories in spiking neurons.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3-4):279–292.

Whitehead, S. D. (1991). A study of cooperative mechanisms for faster reinforcement learning. Technical Report March, University of Rochester.

Wickens, J., Begg, A., and Arbuthnott, G. (1996). Dopamine reverses the depression of rat corticostriatal synapses which normally follows high-frequency stimulation of cortex in vitro. *Neuroscience*, 70(1):1–5.