

# A neural reinforcement learning model for tasks with unknown time delays

Daniel Rasmussen (drasmuss@uwaterloo.ca)

Chris Eliasmith (celiasmith@uwaterloo.ca)

Centre for Theoretical Neuroscience, University of Waterloo

Waterloo, ON, Canada, N2J 3G1

## Abstract

We present a biologically based neural model capable of performing reinforcement learning in complex tasks. The model is unique in its ability to solve tasks that require the agent to make a sequence of unrewarded actions in order to reach the goal, in an environment where there are unknown and variable time delays between actions, state transitions, and rewards. Specifically, this is the first neural model of reinforcement learning able to function within a Semi-Markov Decision Process (SMDP) framework. We believe that this extension of current modelling efforts lays the groundwork for increasingly sophisticated models of human decision making.

**Keywords:** reinforcement learning; neural model; SMDP

## Introduction

One of the most successful areas of cross-fertilization between computational modelling and the study of the brain has been the domain of reinforcement learning (RL). This began with the work of Schultz (1998), who demonstrated that the well-defined computational mechanisms of models (e.g., TD reinforcement learning) could provide insight into some of the more opaque mechanisms of the brain (e.g., dopamine signalling).

The models used in that early work were purely algorithmic, with little relation to the biological properties of the brain. However, since that first demonstration many new models have been developed, allowing novel or more detailed comparisons to neural mechanisms—models that more closely reflect the structures of the brain (Frank & Badre, 2012; Stewart et al., 2012), the behaviour of individual neurons (Seung, 2003; Potjans et al., 2009), or synaptic learning mechanisms (Florian, 2007; Baras & Meir, 2007).

In our work we seek to retain the neuroanatomical detail of these models, but expand their functionality; that is, to build models capable of more powerful learning and decision making, enabling them to solve more complex problems. Here we present some first steps in that direction. Specifically, we will discuss the implementation and show early results from a model that is able to solve tasks requiring extended sequences of actions, in environments where there may be unknown and variable time delays between actions and rewards.

## Background

Sutton & Barto's seminal introduction to reinforcement learning illustrates the important challenge for expanding the function of neural RL models: "Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal...In the most interesting and challenging cases, actions may affect not only the imme-

diate reward but also the next situation and, through that, all subsequent rewards (Sutton & Barto, 1998)."

Most existing neural models have performed only associative reinforcement learning, where there is no consideration of future reward (Niv et al., 2002; Seung, 2003; Baras & Meir, 2007; Florian, 2007; Izhikevich, 2007; Frank & Badre, 2012; Stewart et al., 2012). An example of this type of task is bandit learning, where the agent selects one of  $n$  available options, receives reward, then is reset back to the choice point. Each trial is independent, so the agent only needs to learn the immediate reward associated with each option, and then pick the best one. This can be expressed in the RL notation as

$$Q(s, a) = r(s, a) \quad (1)$$

where  $Q(s, a)$  is the agent's estimate of the value of taking action  $a$  in state  $s$ , and  $r(s, a)$  is the immediate reward received for performing that action in that state. These  $Q$  values can be learned by observing  $r(s, a)$  and then updating  $Q(s, a)$  to bring it closer to the observation. The challenge addressed by many of the models above is how to do that update in a neurally plausible manner.

An example of a more complex reinforcement learning task is a navigation problem, where an agent seeking to reach a goal must choose a direction to move. The agent may receive no immediate reward for making a choice, but there are still good and bad choices (bringing it closer to or farther from the goal). In order to make correct decisions, the agent needs to be able to learn not only the immediate rewards, but the rewards to be expected in the future after taking a given action. This can be expressed as

$$Q(s, a) = r(s, a) + \gamma Q(s', a') \quad (2)$$

In other words, the value of taking action  $a$  is equivalent to the immediate reward (as in the previous case), plus the expected value of the action taken in the resulting state (indicating the future reward expected from that state). The future value is discounted by  $\gamma < 1$  to indicate that future rewards are valued less than immediate rewards. The  $Q$  values can be learned by comparing the predicted value of action  $a$  to the observed values upon arriving in state  $s'$ . This is the temporal difference (TD) learning formula<sup>1</sup>:

$$\Delta Q(s, a) = \kappa [r(s, a) + \gamma Q(s', a') - Q(s, a)] \quad (3)$$

Most complex problems of the type faced by the brain require the consideration of the future impact of a given action; thus

<sup>1</sup>More specifically, this is the SARSA learning update (Rummery & Niranjan, 1994).

building models capable of this type of learning is an important step in understanding the decision making processes in the brain.

There have been models built that solve these types of tasks, but often they take the TD error signal (Equation 3) as given, or it is computed outside the model (Foster et al., 2000; Strösslin & Gerstner, 2003). This reduces to a problem very similar to Equation 1, where the agent has a signal coming in and only needs to worry about the current value of that signal. The challenging aspect of TD learning is how to learn with only immediate rewards as input to the model.

Potjans et al. (2009) presented one of the most complete neural models of reinforcement learning. In order to compute the TD error they use two activity traces, one fast and one slow, on the output of the neurons representing the  $Q$  values. For a brief window in time after the system transitions from state  $s$  to state  $s'$ , the slow trace will still be representing  $Q(s, a)$  while the fast trace will be representing  $Q(s', a')$ ; combining that information with the incoming reward enables the neurons to calculate the equivalent of Equation 3.

The downside of this approach is that the necessary information is only present immediately after the state transition, within that window of time before the slow activity trace catches up to the fast; if action selection occurs earlier than the state transition, or if the rewards are not delivered within that window, the system will not be able to learn. This is true of all systems that rely on some type of activity/eligibility trace to preserve the action values (e.g., Izhikevich, 2007; Florian, 2007). These models rely on an environment that follows a reliable clock-like sequence of action selection, state transition, and reward.

In some cases that may be a reasonable assumption, but in our work we seek a more general mechanism that can learn when there is an unknown and potentially variable delay between action selection and state transition or reward. This can be expressed as a Semi-Markov Decision Process (SMDP; Howard, 1971). Whereas in basic MDPs (the standard model for RL tasks) states, actions, and rewards all occur instantaneously, SMDPs introduce the concept of a time delay between action selection and state transition, and rewards that can be delivered at different points in time.

One way to address the problem of time delays in the MDP environment (without resorting to SMDPs) is to imagine the delay period as a series of state transitions. That is, the states/actions/rewards continue to proceed in a regular clock-like manner, and time delays are represented by multiple cycles through that loop. However, this requires the learning to propagate back through all the “decisions” made during the delay period. This greatly complicates the learning process, and for lengthy delay periods with many different decisions it can render successful learning practically impossible. An important advantage of the SMDP framework is that it encapsulates all the activity of the delay period within a single learning update. This is particularly useful in situations such as hierarchical decision making, discussed more in the con-

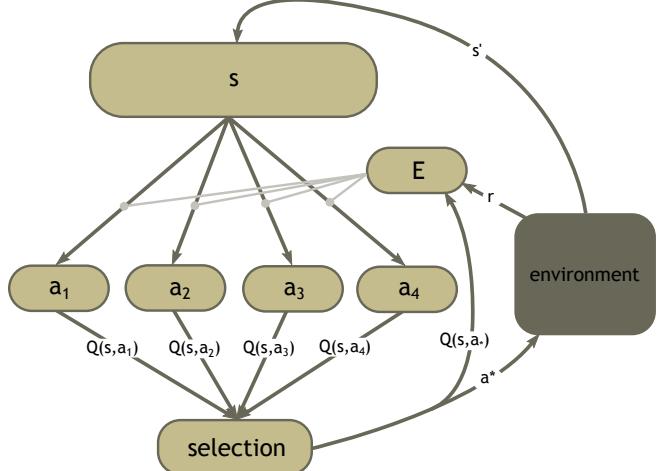


Figure 1: Overall architecture of the model, see text for details. The interior of the  $E$  component is shown in Figure 2.

clusion.

The learning update from Equation 3 can be reformulated for an SMDP environment (Bradtko & Duff, 1994; Sutton et al., 1999) as

$$\Delta Q(s, a) = \kappa \left[ \sum_{t=0}^{\tau-1} r(s, a, t) + \gamma^\tau Q(s', a') - Q(s, a) \right] \quad (4)$$

where  $t$  is the time elapsed since action  $a$  was selected,  $r(s, a, t)$  is the reward received at time  $t$ , and the transition to state  $s'$  occurs at time  $\tau$ . The obvious changes are that a) the rewards received are summed over time, and b) the discount is applied across the delay period. However, the more subtle change is that the agent does not know  $\tau$ . That is, it cannot rely on the rewards or discount being limited to some specific time window, or the update being applied at a particular time; it must simply wait, and be able to calculate Equation 4 whenever the state change occurs. For the sake of simplicity we have expressed time here as consisting of discrete time steps, but it can be expressed in the continuous case by taking the integral over the incoming reward signal (this is the approach used in our model).

With the SMDP framework, an agent can learn to select actions in a more general environment, incorporating arbitrary time delays into the reinforcement learning process. By taking this theory and implementing it in a neural model, we will develop a more powerful and flexible model of reinforcement learning in the brain.

## Methods

### Model architecture

The overall structure of the model is shown in Figure 1. At the top is a population of neurons representing the current state (we will discuss how the environmental state is translated into neural activities in the next section). Beneath are populations associated with each available action (four in this

case, but the model can work with any number). The state population is connected to each action population, and it is in the synaptic weights of these connections that the  $Q$  values are calculated. Assuming that correct weights have been learned, the output of the state neurons will cause each action population to represent the value of taking its associated action in that state (i.e.,  $Q(s, a_n)$ ).

In order to act, the model needs to make a decision based on those  $Q$  values; this is the purpose of the selection component. In our model the agent follows a simple greedy policy of always selecting the highest value action. We compute the max operation using the basal ganglia model described in Stewart et al. (2010). That output is used to activate inhibitory gates within the network of the selection component, so that neural populations corresponding to the non-selected actions will not be active. The output of the selection component is both the value of the selected action, which is sent to the error calculation network (to be discussed later), and the actual output of the agent (i.e., the action it sends to the environment).

The operation of the agent is independent of the details of the environment; this model is designed to function in any task that can be described in the SMDP framework. All that is required is that the environment somehow takes the output of the agent (e.g., an action such as “move left”), calculates an updated state (e.g., the new position of the agent), and sends the new state and any reward received back to the agent. As per the SMDP framework, the state transition can occur at any time, and the rewards can be delivered at any time. When the new state is sent to the agent, it will modify the activities in the state population, a learning update will be performed as in Equation 4, and the agent will decide on a new action.

### Representing and computing with neural activities

The model operates entirely in neural activities, but it needs to interact with environments and perform computations that are defined in terms of abstract mathematical variables. To translate back and forth between these two domains we use the Neural Engineering Framework (NEF; Eliasmith & Anderson, 2003).

The first component of the translation is encoding. For example, the abstract state output by the environment needs to be encoded into the activities of the state population. Suppose the state is represented by a vector  $x$  (perhaps describing the position of the agent). The model operates in continuous time, so the changing state over time can be represented by  $x(t)$ . That input signal is encoded into the activities of the state population as

$$s_i(x(t)) = G_i \left[ \alpha_i e_i x(t) + J_i^{bias} \right] \quad (5)$$

$s_i(x(t))$  represents the activity of neuron  $i$  in the state population.  $G_i$  is the neuron model; in our case we use leaky integrate and fire (LIF) neurons. The components within the braces represent the current that is input into the neuron model.  $\alpha$  and  $J_i^{bias}$  are parameters of the neuron, randomly

chosen from within biologically plausible ranges, representing the gain and background activity, respectively. The vector  $e_i$  identifies the neuron’s preferred stimulus, the area of the input space to which this neuron is most sensitive (these are also randomly chosen). Thus each neuron will respond to the input according to its internal parameters and how close the input is to the neuron’s preferred stimulus. The combined activity of the whole population then comprises a distributed representation of where the current input is in the input space. Note that while for demonstration purposes we have described this here in terms of encoding the state, this is a general purpose mechanism for encoding any input into the activities of a population of neurons.

The second aspect of the translation is decoding, translating the activities of a population of neurons back into an abstract value. For example, this allows the activities of neurons in the selection network to be interpreted as an action for the environment, or the activities of the action populations to be interpreted as  $Q$  values. This is accomplished by a weighted summation of the neural activities:

$$\hat{x}(t) = \sum_i s_i(x(t)) d_i \quad (6)$$

The weights, or decoders,  $d_i$ , can be calculated by

$$d = \Gamma^{-1} \Upsilon$$

where

$$\begin{aligned} \Gamma_{ij} &= \int s_i(x) s_j(x) dx \\ \Upsilon_j &= \int s_j(x) f(x) dx \end{aligned} \quad (7)$$

$f(x)$  gives the option of decoding a (possibly nonlinear) function of the encoded value. However, in most cases all that is desired is the identity of the represented value, in which case  $f(x) = x$ . With these two tools, encoding and decoding, we can translate back and forth between the neural activities of the model and the variables and computations of the RL framework.

### Learning

The basic process of TD reinforcement learning involves updating the agent’s estimation of the value of each action, the  $Q$  values. In the architecture of the model, this means modifying the synaptic weights on the connections between the state and action populations. To perform these updates we use an error modulated neural learning rule developed by MacNeil & Eliasmith (2011):

$$\Delta \omega_{ij} = \kappa \alpha_j e_j E s_i(x) \quad (8)$$

$\Delta \omega_{ij}$  is the change in the connection weight between neuron  $i$  (in the state population) and neuron  $j$  (in an action population).  $\kappa$  is the learning rate,  $\alpha_j$  and  $e_j$  are properties of neuron  $j$  (as shown in Equation 5),  $s_i(x)$  is the activity of neuron  $i$ , and  $E$  is the error. For this model, the error is the desired change in the  $Q$  value, i.e.,  $\Delta Q(s, a)$  from Equation 4. This is

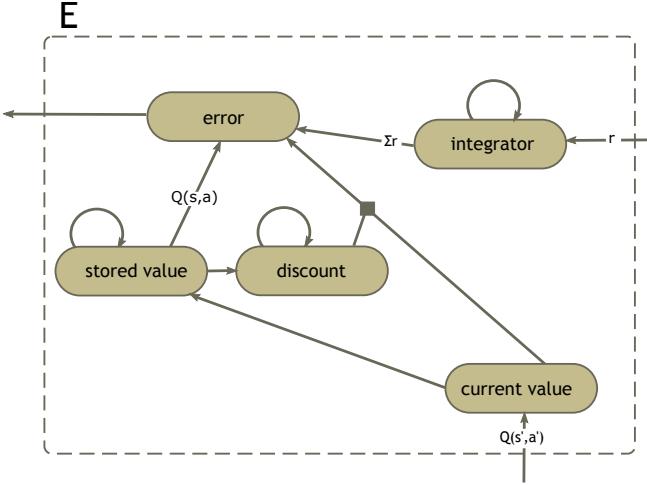


Figure 2: Network to calculate the SMDP learning error (the interior processing of the  $E$  component shown in Figure 1). See text for details.

a neurally plausible weight update in that it only makes use of information available locally at neuron  $j$  (assuming all neurons also receive the error signal  $E$ ). MacNeil & Eliasmith (2011) show that this learning rule will cause the weights to be adjusted so as to minimize  $E$ , meaning that over time the weights will come to calculate the desired  $Q$  values.

### Error calculation

The previous section raises the question of where the error,  $E$ , comes from. That is, how is Equation 4 computed? The network that performs this calculation is shown in Figure 2. Note that this is the  $E$  component shown in Figure 1, and receives the inputs shown there (the  $Q$  value of the selected action, and the reward from the environment).

One challenge is the integration of the incoming reward (the summation in Equation 4). This is accomplished by the top-right component in the network. The central feature of the integration population is the recurrent connection, which allows it to maintain its activity in the absence of input. This means that as new rewards enter the population they will be added to the previous rewards already being represented, so that the population represents the sum of the given rewards. The details of how to set up a recurrent network to perform these kinds of computations are described in Eliasmith (2005).

The “current value” population represents the value of the currently selected action. When the action is first selected, this value is transferred into the “stored value” population in the bottom left. Again, this is a population that will maintain its represented value via its recurrent connections. When a state transition occurs, the bottom population will then be representing the value of the selected action in the new state,  $Q(s', a')$ , while the “stored value” population maintains  $Q(s, a)$ .

The discount is calculated by integrating the value repre-

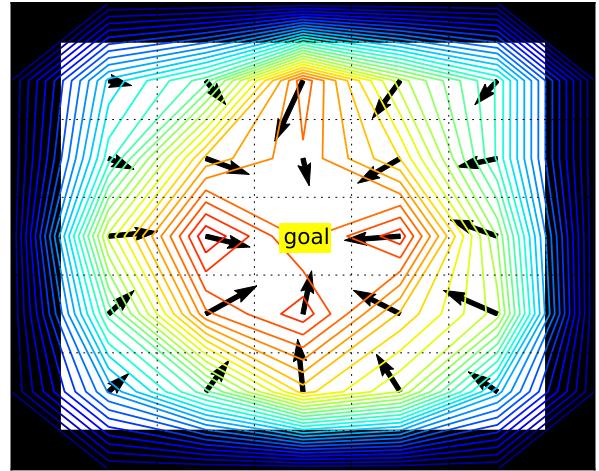


Figure 3: Example of policy learned by the agent. The arrows represent the weighted sum of the four possible movement directions, where each direction is weighted by the learned  $Q$  value of that action. Contours indicate the state value (the value of the highest-valued action).

sented in the “stored value” population, using the same recurrent setup as is used to integrate the incoming reward. This value is then subtracted from the current  $Q$  input to calculate a discounted action value. This is not identical to the discount expressed in Equation 4, but it has a similar computational effect: it reduces the value of future states proportional to the time elapsed and the value of the state.

The final “error” population thus has all the pieces it needs to compute the SMDP learning update. It adds the accumulated reward and the discounted  $Q(s', a')$  value, and subtracts the stored  $Q(s, a)$  value, resulting in the error signal required by the neural learning rule (Equation 8).

### Results

We tested the model on a spatial navigation task (the same task used in Potjans et al. 2009). The agent is randomly placed in a  $5 \times 5$  grid, surrounded by walls. The agent’s state is its  $x, y$  location in the grid, and the available actions are movement in the four cardinal directions. Selecting any of those actions will cause the agent to move one square in that direction, unless it is attempting to move into a wall in which case it remains in the same position. The agent’s time in each state is randomly determined, ranging between 600 and 900ms. The task is to move to some fixed target location. This is equivalent to a water-maze type task, where the agent has no idea where the goal might be, and must find it by exploring the environment. When the agent finds the goal state it receives a constant reward of 1 as long as it remains in the state. After a brief period of time the agent is then moved to a random location, and must find the target again.

Figure 3 shows an example of a policy learned by the model after spending approximately 2hrs of simulated time

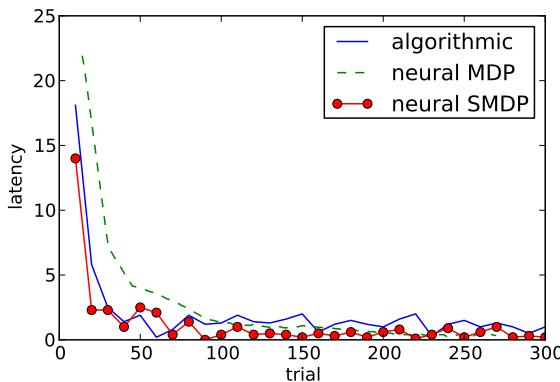


Figure 4: Comparison of learning times between a) an algorithmic implementation of RL (basic table-based Q-learning), b) the neural MDP reinforcement learning model of Potjans et al. (2009), and c) the model presented here. Latency is measured as the difference between the Manhattan distance from start to goal and the number of steps taken by the model. Data for b) from Potjans et al. (2009).

in the task. The arrows display the weighted sum of the four movement directions, where the weights are the learned  $Q$  values associated with each action. Since the agent picks the highest valued action, it will move in whichever cardinal direction is closest to the direction of the arrow. The contours indicate the value of the highest valued action (i.e., the state value function). It can be seen that the agent has successfully learned a policy that will take it to the goal state from any position, despite the random time delays.

Figure 4 shows a comparison between the learning times of our model and that of Potjans et al. (2009), with a purely computational RL implementation as a baseline. Each trial begins when the agent is placed at a random location in the grid, and ends when it reaches the goal (at which point it is placed in a new location for the next trial). We have followed Potjans et al. in using latency as a measure of how well the agent has learned the task. Latency is defined as the difference between the Manhattan distance between the start and goal ( $\text{start}_x - \text{goal}_x + \text{start}_y - \text{goal}_y$ ), which is the shortest possible path length, and the number of steps taken by the model. It can be seen that our model performs better than that of Potjans et al., and roughly equivalently to the purely computational solution. It is also worth noting that our model is operating in the more challenging SMDP framework, with random time delays; it is unlikely that the Potjans et al. model would be able to perform this task at all.

SMDPs also provide a more powerful language with which to describe problem domains, by allowing for the incorporation of time directly into the task description. For example, Figure 5 shows a task similar to that in Figure 3, but certain states (shown in grey) take a longer period of time for the agent to move through (simulated by adding three seconds

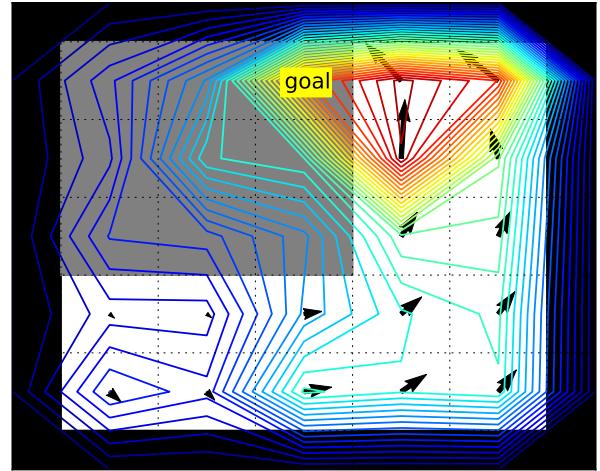


Figure 5: Policy learned by the system in a task where certain states take longer periods of time to move through (shown in grey). The agent has learned to avoid the slow areas even though it requires taking a less direct route.

to the usual randomly determined state transition time). This means that the most efficient route to the goal is no longer a direct path; the agent has learned to trade off the cost of a detour with the cost of moving through the slow areas. Time is often an important part of real world tasks, thus the ability to incorporate time directly into the agent's learning is another advantage of the SMDP framework.

## Discussion

We have presented a novel neural model capable of autonomous reinforcement learning. The model is able to solve complex tasks that require an extended sequence of actions in order to achieve the reward, rare for biologically based neural models. In addition, it is able to solve these tasks in a realistic SMDP environment, where there are potentially random and unknown delays between action selection, state transition, and reward. We believe this is currently the only neural model capable of this type of performance.

This model is still only an early step on the path of expanding the functional capabilities of neural RL models, and there are a number of ways in which it can be improved. First, more neural detail could be incorporated into the model. For instance, incorporating more realistic spiking neurons would allow for more detailed comparisons to neural recordings. Another improvement to the model would be a more principled approach to exploration. At the moment exploration is accomplished by injecting random noise into the action values as they enter the selection component (a neural approximation of the standard  $\epsilon$ -greedy approach). However, in the future it would be desirable to have more control over the exploration process, so that, for example, the agent could make decisions about how much exploration to pursue based on its current knowledge.

Another avenue for future work is to incorporate the learning components of this system into a more complete agent model. The inputs and outputs of this model are abstract, thus it ignores the complexity of sensory processing and motor output. However, recent work in our lab has developed an integrated brain model that is able to perceive visual input, process it internally, and control motor outputs (Eliasmith et al., 2012). That model was able to perform associative reinforcement learning, but not the more complex learning displayed here. Adding the abilities of this model into that detailed neural agent would allow for the study of the full reinforcement learning process, from input through to output.

One of the most interesting possibilities opened up by this model is the construction of a neural model capable of hierarchical reinforcement learning (Barto & Mahadevan, 2003). In hierarchical RL the “actions” that an agent chooses between can be augmented with subroutines that define whole new behaviours. For example, instead of the agent just choosing between “go left”, “go right”, and so on, one of its options could be “go to the doorway”, which would then lead to a sequence of decisions designed to take the agent to that location. What all of the hierarchical approaches have in common is that they use the SMDP framework as their underlying structure. The unknown time delay between action and state transition can be used to encapsulate the time when the high-level action is executing. The SMDP framework allows the agent to incorporate those time delays and rewards, and learn how to correctly select between its complex set of actions. A model such as the one we present here is a step toward a functional neural model capable of hierarchical learning and decision making.

### Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada, Canada Research Chairs, the Canadian Foundation for Innovation, and Ontario Innovation Trust.

### References

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

Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 1–28.

Bradtko, S. J., & Duff, M. O. (1994). Reinforcement learning methods for continuous-time Markov decision problems. In *Advances in Neural Information Processing Systems*.

Eliasmith, C. (2005). A unified approach to building and controlling spiking attractor networks. *Neural Computation*, 17(6), 1276–1314.

Eliasmith, C., & Anderson, C. (2003). *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. Cambridge: MIT Press.

Eliasmith, C., Stewart, T. C., Choo, X., Bekolay, T., DeWolf, T., Tang, C., & Rasmussen, D. (2012). A large-scale model of the functioning brain. *Science*, 338(6111), 1202–1205.

Florian, R. V. (2007). Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19(6), 1468–502.

Foster, D. J., Morris, R. G., & Dayan, P. (2000). A model of hippocampally dependent navigation, using the temporal difference learning rule. *Hippocampus*, 10(1), 1–16.

Frank, M. J., & Badre, D. (2012). Mechanisms of hierarchical reinforcement learning in corticostriatal circuits 1: computational analysis. *Cerebral Cortex*, 22(3), 509–26.

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.

MacNeil, D., & Eliasmith, C. (2011). Fine-tuning and the stability of recurrent neural networks. *PloS ONE*, 6(9), e22885.

Niv, Y., Joel, D., Meilijson, I., & Ruppin, E. (2002). Evolution of Reinforcement Learning in Uncertain Environments: A Simple Explanation for Complex Foraging Behaviors. *Adaptive Behavior*, 10(1), 5–24.

Potjans, W., Morrison, A., & Diesmann, M. (2009). A spiking neural network model of an actor-critic learning agent. *Neural Computation*, 339, 301–339.

Rummery, G., & Niranjan, M. (1994). *On-line Q-learning using connectionist systems* (Tech. Rep. No. September).

Schultz, W. (1998). Predictive reward signal of dopamine neurons. *Journal of Neurophysiology*, 80, 1–27.

Seung, H. S. (2003). Learning in spiking neural networks by reinforcement of stochastic synaptic transmission. *Neuron*, 40(6), 1063–73.

Stewart, T. C., Bekolay, T., & 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., & Eliasmith, C. (2010). Dynamic behaviour of a spiking model of action selection in the basal ganglia. In S. Ohlsson & R. Catrambone (Eds.), *Proceedings of the 32nd Annual Conference of the Cognitive Science Society* (pp. 235–240). Austin: Cognitive Science Society.

Strösslin, T., & Gerstner, W. (2003). Reinforcement learning in continuous state and action space. In *International Conference on Artificial Neural Networks*.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning*. Cambridge: MIT Press.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2), 181–211.