Design and implementation of general purpose reinforcement learning agents

by

Tyler Edward Streeter

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Human Computer Interaction

Program of Study Committee:
Adrian Sannier, Major Professor
James Oliver
Dimitris Margaritis

Iowa State University

Ames, Iowa

2005

Graduate College
Iowa State University

This is to certify that the master's thesis of

Tyler Edward Streeter

has met the thesis requirements of Iowa State University

_____
Major Professor

_____
For the Major Program

Table of Contents

Acknowledgements

- I thank my wonderful wife, Shana, for her constant moral support, for listening to my ideas, for providing new perspectives on my research, and for saying "yes" in the first place. :)

- I want to thank my family, Ed, Cindy, Tony, Tianna, and Taraleh, and my parents-in-law, Kip and Deena, for their continued support throughout my education.

- I want to thank my thesis committee, Dr. Adrian Sannier, Dr. Jim Oliver, and Dr. Dimitris Margaritis, for their suggestions during the research process and for being willing to advise me on this work.

- I thank Joe Heiniger for years of interesting discussions, for his seemingly endless supply of inspiration from science fiction, and for his friendship.

- I thank Andres Reinot, Alan Fischer, and Oleksandr Lozitskiy for being part of the OPAL development team.

- I thank Russ Smith for writing ODE and the ODE community for continued development.

- I thank the researchers at Natural Motion whose initial work inspired my experiments with neuroevolution of humanoids. I especially thank Mat Best for his interest in my work.

- I thank Ken Stanley, author of the NEAT algorithm, for his helpful suggestions. In particular, he had the idea of using continuous-time recurrent neural networks. He helped me develop a CTRNN version of NEAT which I used to train walking bipeds.

- I thank Dexter Allen for several discussions on neuroevolution applied to legged creatures.

- I thank the faculty, staff, and students at the Virtual Reality Applications Center. The lab is an ideal environment for aspiring software developers. I have gained a tremendous amount of practical experience from working there.

- I wish to thank Dr. Adrian Sannier for his contagious spirit. His own curiosity about the world has provided a rich variety of interesting discussions. He fueled my initial drive to tackle the intelligence problem. I still think about our first ideas on intrinsic motivation and hierarchies of motor programs.

Abstract

Intelligent agents are becoming increasingly important in our society. We currently have house cleaning robots, computer-controlled opponents in video games, unmanned aerial combat vehicles, entertainment robots, and autonomous explorers in outer space. But there are many problems with the current generation of intelligent agents. Most of these problems stem from the fact that they are designed for very specific problems. Each intelligent agent has limited adaptability to new tasks; if conditions change slightly, the agent may quickly become confused. Additionally, a huge engineering effort is required to design an agent for each new task. Ideally, we would have a reusable general purpose agent design. Such a general purpose agent would be able to adapt to changing environments and would be easy to train to handle new tasks. To implement this agent design, we can use ideas from the field of reinforcement learning, an approach with strong mathematical foundations and intriguing biological implications. The available reinforcement learning algorithms are powerful because of their generality: agents simply receive a scalar reward value representing success or failure. Additionally, these algorithms can be combined with other powerful ideas (e.g. planning from a learned internal model).

This thesis provides a step towards the goal of general purpose agents. It discusses a detailed agent design and provides a concrete software implementation of these ideas. It covers the components necessary for such a general purpose agent, starting with a minimal design and proceeding to develop a more powerful learning architecture. The final design uses temporal difference learning, radial basis functions, planning, uncertainty estimations, and curiosity. The main contributions of this thesis are: a novel combination of temporal difference learning with planning, uncertainty, and curiosity; a discussion of correlations between theoretical reinforcement learning and reward processing in biological brains; a practical Open Source implementation of general purpose reinforcement learning agents; and experimental results showing learning performance on several tasks, including two physical control problems.

1 Introduction

Intelligent agents[1] today are primitive compared to human intelligence. Nevertheless, they are finding uses everywhere. They help diagnose diseases, they aid in making stock market predictions, they provide entertainment as physical robots and as opponents in video games, they perform dangerous military operations, they handle household chores, they detect credit card fraud, they explore other planets, and they construct automobiles.

Machine intelligence is probably one of the most important technologies to develop. In general, any human endeavor that could benefit from additional brain power will benefit from improved machine intelligence. On a grand scale, having agents with human-like intelligence would amplify our progress in any scientific field. On a more personal level, we could replace the user interfaces on our personal computers with intelligent assistants that manage menial tasks for us.

The major problem with current intelligent agents is that they lack flexibility. They are usually designed to operate under certain conditions for a specific purpose. This fact prevents them from adapting to new environments. It also makes the design process laborious because each agent is usually created from scratch.

A fairly new approach is that of creating agents to learn from direct interaction with their environments. No knowledge is bestowed from the agent's creators; everything must be learned through firsthand experience. This kind of agent must go through a developmental phase where it spends most of its time exploring, learning about its world's predictable properties. One of the major hypotheses of this approach is that these agents will be more adaptable and effective at solving complex problems. We argue here that reinforcement learning is a practical way to design and implement this type of agent.

---

[1] The term "agent" is used here to describe any intelligent system, including robots, animals, and humans.

Reinforcement learning is the problem of choosing the optimal action in a given situation in order to maximize future rewards (Sutton & Barto, 1998). Figure 1 shows the general situation with a set of abstract objects representing the agent and its environment. The agent performs an action which usually has some effect on the environment. Based on that action, the environment provides the agent with new sensory inputs (i.e. an observation) and a reward signal. Positive reward signals positively reinforce the actions that led to the rewarding situation, making them more likely to be performed in the future. Similarly, negative reward signals make the recent actions less likely to be performed.

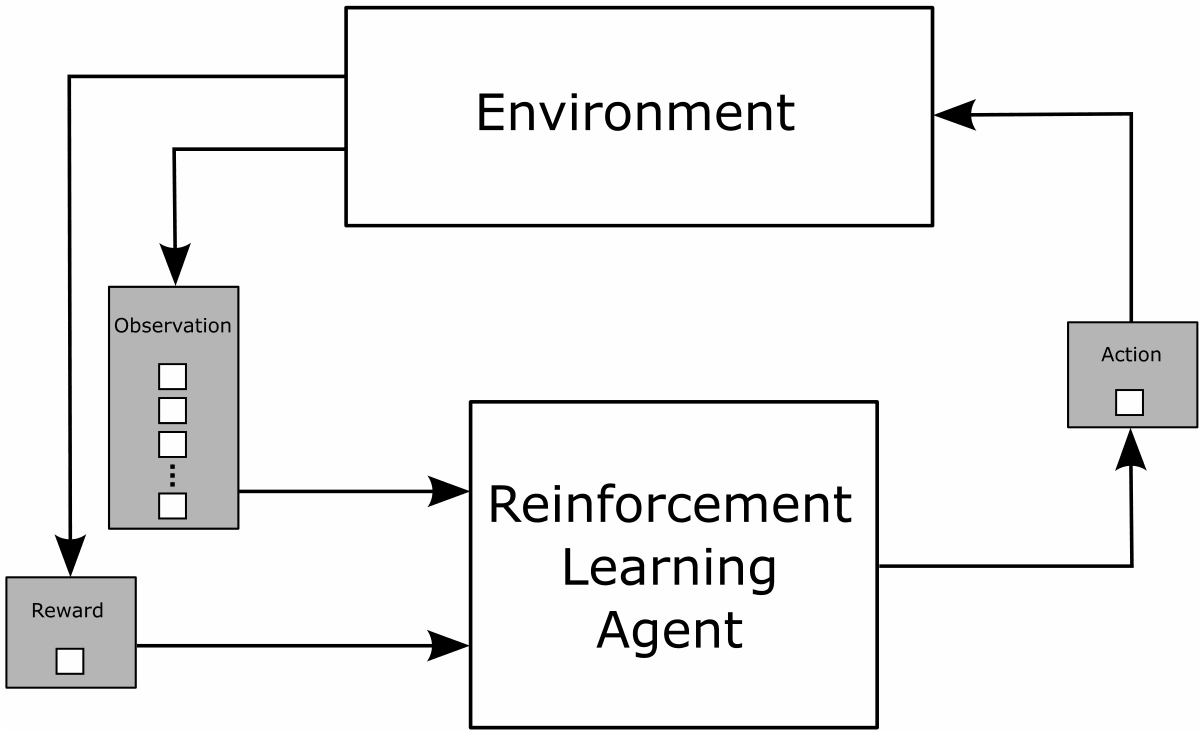

Figure 1: A typical reinforcement learning setup, including an environment that provides observations and rewards and an agent that responds with actions.

Reinforcement learning is a trial-and-error process. The only way for an agent to improve its performance is to take an action and experience the resulting reward (or lack thereof). This approach is very general. Almost any problem can be expressed as a reinforcement learning problem as long as the goal can be represented as a scalar reward value. Thus, algorithms designed to solve reinforcement learning problems are applicable to a wide variety of tasks.

When trying to solve reinforcement learning problems, a few major problems arise. If an agent receives a reward after a long sequence of actions, how does it know which actions to reinforce? For example, say we are training a dog to roll over on command, and we reward him only when successfully finishing the task. The dog must learn to reinforce the initial action of lying down even though he does not receive a reward until after rolling over and standing up again. Another problem is that of knowing when to try new actions and when to choose the one that has been most successful in the past.

Fortunately, a set of powerful reinforcement learning algorithms already exist. They can successfully deal with the problems mentioned above and more. Combined with other techniques (e.g. function approximation, planning), the core algorithms can scale to more complicated problem domains. However, even with the tools that are available, it is not yet clear which ones are best and how they should be combined.

This thesis presents a summary of some of the algorithms available for solving reinforcement learning problems and shows how they can be combined effectively to create a general purpose solution. The ideas discussed here are implemented as an Open Source software library (Verve http://verve-agents.sourceforge.net) designed to give application developers a useful tool for creating intelligent learning agents. This library can be applied to simple simulated agents in discrete grid worlds and to real robots acting in the physical world.

There are existing reinforcement learning software tools (RL Toolkit http://rlai.cs.ualberta.ca/RLAI/RLtoolkit/RLtoolkit1.0.html, Reinforcement Learning Toolbox http://www.igi.tugraz.at/ril-toolbox/general/overview.html, PIQLE http://www.lifl.fr/~decomite/piqle/index.html) available that provide general frameworks for experimenting with different algorithms. Other tools (SALSA http://www.cs.indiana.edu/~gasser/Salsa) are useful for teaching the concepts of reinforcement learning without having to write software. There is still a need for an "out-of-the-box" solution for non-researchers. Many developers could use a general purpose

learning tool without needing to understand the underlying complexity. This should help promote the use of reinforcement learning algorithms in more diverse applications.

The rest of this chapter describes some of the initial work that motivated this research and ends with a discussion of how this research applies to human computer interaction. The next chapter will discuss reinforcement learning in more detail, covering some of the specific challenges involved. Chapter 3 presents several interesting connections between theoretical reinforcement learning and biological brains. Chapter 4 introduces the Verve software library implementation and shows how it is designed to tackle the challenges in chapter 2. Chapter 5 goes through a series of experiments that test Verve's effectiveness. Chapter 6 ends the thesis with a summary, a list of contributions, and several areas of future work.

Initial Work

The author's early research, mainly inspired by work such as Reil & Husbands (2002), dealt with physically simulated humanoid robots (see Figure 2). These agents used artificial neural networks to transform sensory inputs (joint angles) to continuous motor outputs (desired joint angles). The muscles were simple PD (proportional derivative) controllers that applied muscle forces in order to achieve the desired joint angles. Open Dynamics Engine (ODE http://www.ode.org) was used to simulate physics.

The training phase for these humanoids used a genetic algorithm to optimize motor control performance. The process started with a population of random neural network controllers. Each member of the population was evaluated on a given task and assigned a fitness value for its performance. For example, for the task of standing up without falling, higher fitness was given for maintaining a high average head height. After all members were evaluated, the worst members were discarded, and the best ones generated new offspring using genetic crossover and mutations. This entire sequence progressed through a long series of generations until a certain level of fitness was achieved.

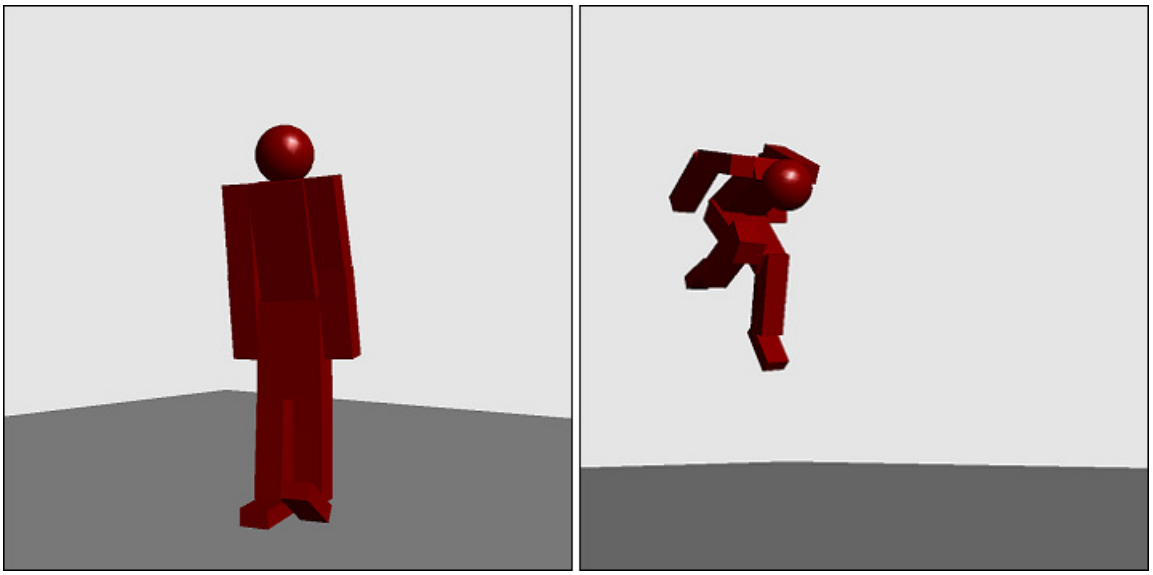

Figure 2: Physically simulated humanoid robots learning to stand and jump.

After successfully getting simulated humanoids to stand and jump, the author used the neuroevolution algorithm NEAT (Stanley & Miikkulainen, 2002; The NeuroEvolution of Augmenting Topologies (NEAT) Users Page http://www.cs.utexas.edu/users/kstanley) to evolve controllers for biped walking (see Figure 3). NEAT has three major advantages over the simpler method used for the standing and jumping tasks: 1) it uses a principled method of genetic crossover – it keeps track of similar genes to help splice genomes in appropriate places, 2) it uses speciation to protect new mutations until they are ready to compete with the rest of the population, and 3) it starts from a minimal neural network (inputs and outputs only) and slowly adds complexity.

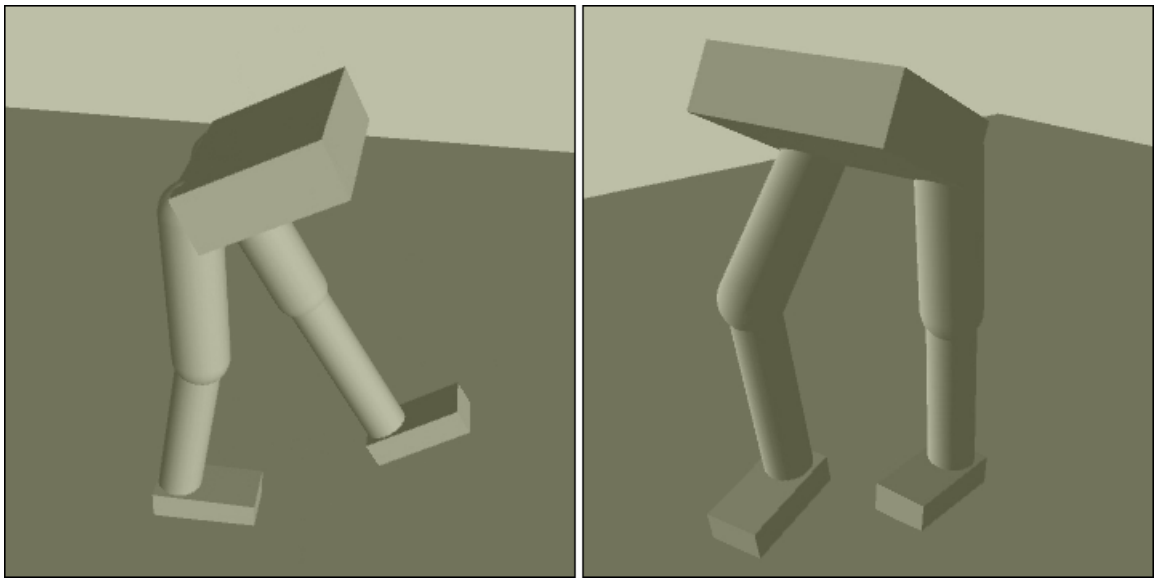

Figure 3: A physically simulated biped learning to walk.

After getting mediocre results with the walking bipeds[1], the author began to question whether genetic algorithms were the best way to solve motor control tasks. After all, it is unlikely that such a process is behind motor learning throughout the lifetime of an animal. One of the major problems with evolutionary methods in general is that learning does not proceed incrementally but only at the end of each trial. Ideally, agents should learn continuously throughout their lifetimes. At this point the author began studying reward-based learning in biological brains to gain insight into how animals and humans achieve goals. Instead of trying to evolve the whole neural network structure from scratch, it seemed more efficient to incorporate existing knowledge about the structure of biological brains. Soon after, the author came upon the recently-discovered connection between biological reward processing (in dopamine neuron activity) and temporal difference learning. This naturally led to further study of the foundations of reinforcement learning.

[1] The best agents could take five steps before falling over.

Intelligent Agents and HCI

Human Computer Interaction (HCI) is the study of the interaction between humans and computers. The overall goal is to increase the usefulness of computer technology by applying it to new problems and by making it less cumbersome to use. Ideally, computers would be able to read our thoughts and aid us without requiring much of our attention. Obviously, we are currently very far from this goal. What we need is more intelligent computers that can adapt to us and learn our intentions, however subtle they may be. We need agents that understand natural language (including any user-specific nuances) and, given simple commands, can achieve complex goals. This would enable user interfaces much more powerful than the ones we have now.

In order to develop such intelligent, adaptable user interfaces, we must continue to make advances in machine intelligence. Specifically, we must develop agents that acquire broad skill sets which they can use to solve wide varieties of tasks. With this goal in mind, the work described in this thesis is a part of the foundation for future intelligent, interactive devices and adaptable interfaces.

2 Reinforcement Learning Challenges

Reinforcement learning, as we have already discussed, presents us with several challenges. One challenge described earlier is known as the "temporal credit assignment" problem, i.e. deciding which of a sequence of previous actions led to a reward. A related problem is the "structural credit assignment problem," the problem of knowing which internal parts of the agent need to be reinforced. This chapter highlights these and other challenges we face when designing general purpose reinforcement learning agents. Some of these challenges are fundamental to the basic functioning of a reinforcement learner, while others help make the core algorithms more practical (i.e. scaleable to large state spaces). We will not cover all possible aspects of the various issues and algorithms; we will focus on the most pertinent information for the goals of this thesis. For more information, see Barto & Sutton (1998).

Note that the reward signal does not specify *how* to make adjustments to improve behavior; it is simply a coarse performance evaluation (i.e. success or failure). Essentially, reinforcements received after performing an action increase the probability that the action will be repeated. According to Thorndike (1911, p. 244): "Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond." This statement, known as Thorndike's "Law of Effect," was one of the early attempts to explain how reinforcing events affect behavior.

Two of the main internal components of most reinforcement learning agents are: 1) a *value function* which maps states to value estimations, and 2) a *policy* which maps states to actions[1]. Figure 4 shows a simple agent design with these components.

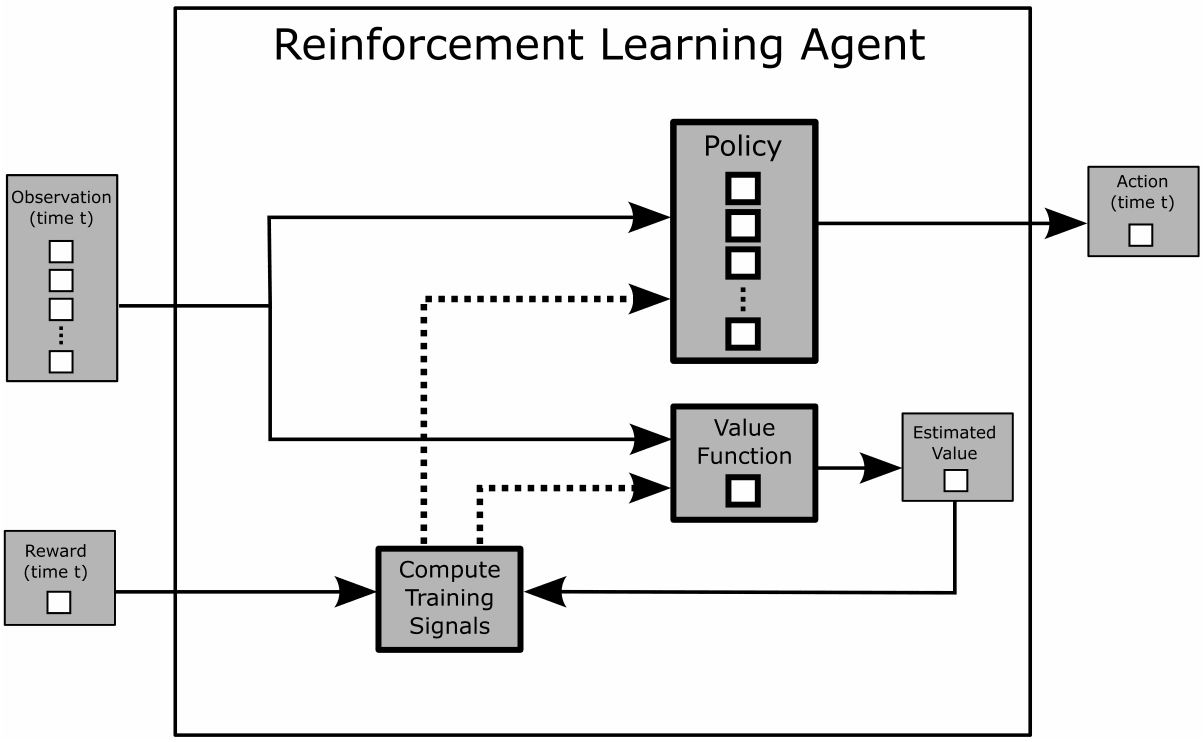

Figure 4: The internals of an initial agent design. The main features are the value function and the policy. Based on the given observation, the policy chooses an action, and the value function estimates the value of the current situation. A prediction error is computed and used to train the value function and policy.

This agent can sense the state of its environment, use its policy to choose actions, and reinforce actions using the primary reward signal. Throughout the chapter we will continually add new components to this design.

---

[1] Having the value function and policy in separate memory structures is sometimes called an "actor-critic" architecture: the policy is an "actor" that continually performs actions, and the value function is a "critic" that reinforces the actor using prediction errors.

Temporal Credit Assignment

While an agent is interacting with its environment, usually rewards are received discontinuously. The agent might move through hundreds of different states, receiving zero reward, before finally reaching its goal state where it receives a positive reward signal. Without a sophisticated way to process rewards, the agent can only reinforce actions taken immediately before receiving the reward.

Imagine a rat running through a maze to find a cheese reward. The rat's primary reward is very discontinuous (ignoring the effects of cheese smell gradients leading to the goal). The rat never benefits from this food reward until it actually reaches the goal and eats the cheese. The rat still solves the task effectively, so it must do more than just reinforce the actions taken immediately before finding the cheese. Again, this is the temporal credit assignment problem. After a long sequence of actions, how does the rat know which ones contributed to getting the reward (i.e. which ones should be positively reinforced)?

A more fundamental question is: How does an agent know which states are more valuable than others when the reward is actually only present in a single state? One answer is that the agent must *learn* the value of each state through direct experience. There is usually more "value" in being close to the reward than being far away, both spatially and temporally. This information is not present in the environment; it must be learned. Thus, an effective intermediate step in solving a reinforcement learning task is to learn a "value function." A value function (more specifically, a state value function[1]) is a mapping of states to values. Given some state, the value function returns the (usually estimated) value associated with being in that state. It transforms the discontinuous primary reward into a continuous internal

[1] Another type of value function is an "action value function" that represents the value of taking an action in a specific state. The Q learning approach, for example, uses an action value function. Here, we will only focus on state value functions. It is unclear that action value functions have any definite advantage over state value functions.

signal[1]. It is helpful to think of the agent playing the "hot-or-cold" game, where the rewards are "hot." The learned value function tells the agent whether it is in a hot or cold state. At first the value function might be wildly inaccurate, but it should improve through experience.

Two main questions arise at this point: 1) how does the agent learn an accurate value function?, and 2) once the value function is learned, how should the agent use it?

It is important to define what we mean by the "value" of a state. The usual meaning is the expected sum of future rewards, i.e. how much reward we can expect to receive from this state forward. Thus, the value of a state is the reward received at that state plus the sum of rewards that can be expected after that point. One major problem with this approach is that the future sum of rewards could be infinitely large. We can alleviate this by discounting rewards received farther into the future.

Assuming we have a complete model of the environment (including state transitions and rewards), we can search through all possible states and compute the value of each. Starting at an initial state, we iterate through every possible action and compute the next states. From each of those states, we iterate through every possible action and compute the next states… (This type of exhaustive branching is similar to how most computer chess programs operate.) Whenever we find a reward, we "backup" its value to previous states. This effectively spreads out the value from the reward to the states leading up to it.

---

[1] Rather than giving the agent rewards discontinuously (e.g. zero reward everywhere, and a +1 reward when the task is completed successfully), it is possible to use a continuous reward function. For example, the agent could receive a reward inversely proportional to its distance from some goal. This can be advantageous in some cases because it provides much more information early in the learning process. Right from the start, the agent experiences a reward gradient at every step. However, it can introduce an unwanted bias in nontrivial problems that leads the agent to locally optimal solutions. In general it is safer to provide sparse rewards and force the agent to learn the value of every state on its own.

But what if we do not have a complete model of the environment? This is a valid concern. Most of the time we assume the agent has no initial knowledge of its environment. Another method for learning a value function is by taking samples from actual experience. Without any prior knowledge of its environment, an agent can interact with it directly, keeping track of the average rewards received after being in each state.

The first method described above is called dynamic programming. It has a strong mathematical foundation, but it requires a full model of the environment, making it impractical for agents operating in new territory. The second method is the Monte Carlo approach. It does not require any kind of environment model, but it is difficult to use incrementally (usually all learning updates occur at the end of a long sequence of events).

A fairly new method which has some of the benefits of dynamic programming and Monte Carlo methods is called temporal difference learning. It does not require an environment model[1], and it can perform incremental updates at every time step, so it has advantages over both dynamic programming and Monte Carlo methods. These are necessary requirements in most on-line learning scenarios where the agent starts with no knowledge of the world. "Temporal difference" refers to the fact that the goal is to learn to predict the difference in value between successive time steps. Agents using temporal difference learn an estimate from an estimate; they "bootstrap" the learning process by starting with an initial (usually random) estimated value function and incrementally improve its accuracy based on the previous estimate.

We will now derive the basic equation for one-step temporal difference learning (i.e. TD(0)). Our initial assumption is the following:

$$V(s_t) = r_t + r_{t+1} + r_{t+2} + \ldots$$

[1] Although temporal difference learning does not require an environment model, there is nothing preventing this. It is possible to use a learned environment model with temporal difference to improve learning performance (i.e. "planning", discussed in detail later).

where $V(s_t)$ is the value of the current state. We assume that the value of the current state, s_t, is equal to the current reward, r_t, plus the rewards received at all times after time t. To avoid the possibility of an infinite sum of future rewards, we discount future rewards exponentially with a discounting constant, γ. This makes the value of immediate rewards greater than the current value of rewards received later (i.e. "one in the hand is worth two in the bush"). The following reflects this change:

$$V(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \ldots$$

We can simplify these ideas to get the following form:

$$V(s_t) = r_t + \gamma V(s_{t+1})$$

In other words…

$$0 = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Of course, this assumes that the value function V is completely accurate. This will not always be true. When the value estimation is not correct, we have the following scenario:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

where δ_t is the temporal difference (TD) error. This error is positive when the reward is higher than expected, and it is negative when the reward is lower than expected. Essentially, the TD error provides the agent with more informative feedback than the primary reward signal itself.

The TD error value can be used to update the value function to make it more accurate in the future. The following update equation shows the basic idea:

$$V(s_t) \leftarrow V(s_t) + \eta_{value}\, \delta_t$$

where η_{value} is the value function learning rate. This equation updates the value of the current state using the current prediction error. When the TD error is zero (implying a perfect value estimation), no changes occur.

Now that we know how to learn the value function, we can use it to reinforce actions. More importantly, we can reinforce actions performed at *every* step, not just when receiving rewards. This is achieved by using the same temporal difference error used to train the value function. If the agent takes an action, and the following TD error is positive, the value of the new state is higher than expected, so we positively reinforce that action. Similarly, we negatively reinforce actions that result in negative prediction errors. To "reinforce an action," we simply adjust the action selection probability of the previous action in the direction of the error. For example, if things were better than expected, the positive TD error increases the previous action's selection probability.

Over time the agent's estimated value function and policy grow closer to the ideal value function and policy. Interestingly, the two components depend on each other: the value of a state is dependent upon the actions being chosen, and the policy's actions are reinforced based on the value function's estimates.

Structural Credit Assignment

Now that we know *when* to reinforce actions, how do we know which ones to reinforce? Which structural parts of the agent's value function and policy should be affected when there is a non-zero prediction error? This is known as the "structural credit assignment" problem.

The naïve approach is to update the value of the previous state and reinforce the previously chosen action, i.e. the 1-step TD(0) method introduced in the previous section. However, we can do better. Each value estimation and action can use a separate *eligibility trace*, e, whose purpose is to track structural components (e.g. connection weights in a neural network) that

are eligible for modification. They increase when the corresponding value estimation or action is used, and they decrease exponentially over time (see Figure 5). The assumption is that state value estimations and actions performed just prior to a non-zero temporal difference error were most likely to contribute to that error.

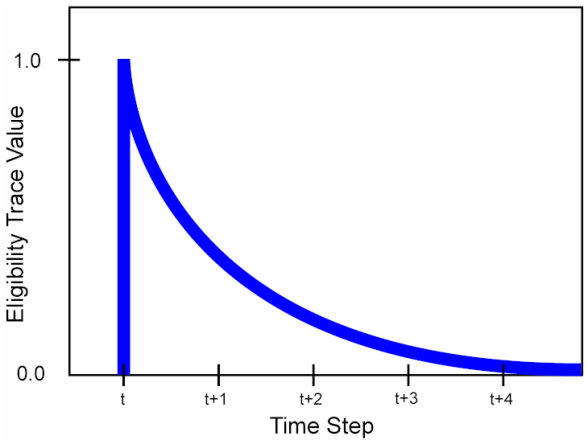

Figure 5: A single eligibility trace decaying over time.

Figure 6 presents a different view, showing the levels of eligibility traces from several previous value estimations and actions at the current time. This figure helps to show how much the previous value estimations and actions should be affected. Those that occurred just prior to a TD error will be affected more.

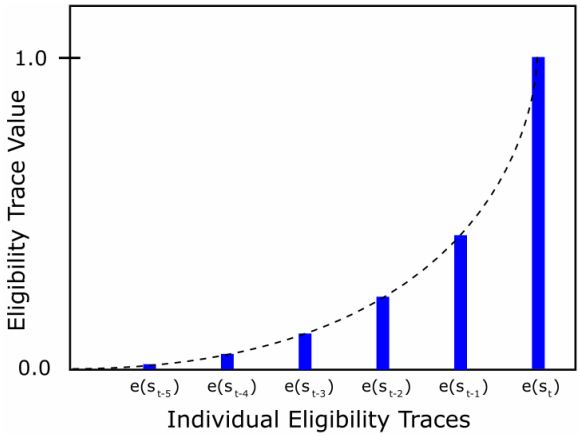

Figure 6: Eligibility traces at time t for several previous value estimations and actions, assuming states are not revisited.

Each eligibility trace is updated on every time step. The traces for the current value estimation and action are increased (e.g. set equal to 1). All other traces are exponentially decreased as follows:

$$e(s_t) \leftarrow \gamma\lambda e(s_t)$$

where λ is a decay constant that ranges from 0 to 1. When TD errors occur, they are applied to state values and actions in proportion to their eligibility traces. We use the same TD error (δ_t) equation as before, but the value function update equation now includes eligibility traces. The following shows the new value function update:

$$V(s_t) \leftarrow V(s_t) + \eta_{value}\, \delta_t e(s_t)$$

Temporal difference learning with eligibility traces is called TD(λ). Theoretically, eligibility traces provide a link between temporal difference learning and Monte Carlo methods. When $\lambda = 0$ we get the simple one-step TD(0) rule, but as λ approaches 1, TD(λ) becomes more similar to Monte Carlo learning since it keeps track of all previous states and actions. TD(1), however, is more general than Monte Carlo because it allows incremental learning. The main result we achieve by using eligibility traces is that we can perform structural credit assignment more effectively by targeting specific (structural) parts of the value function and policy when performing updates.

Exploration vs. Exploitation

Imagine this situation: you are invited annually to attend a banquet, and each year you must decide which meal you want. You are always given the choice of eating chicken, beef, or crème de rate de chèvre avec la boue. You usually choose chicken because it tastes great every time. Once in a while you choose beef, but it's a risky choice because sometimes it is not cooked well enough. You *never* choose crème de rate de chèvre avec la boue because you have no idea what it is, but it might actually be a better choice than chicken. Do you stay with your current favorite, or do you risk trying something new?

This is an example of the exploration vs. exploitation dilemma. An agent can take exploratory actions with the hope of finding something better than its current best solution, or it can simply exploit its current policy and never try new actions. Exploring is certainly a risk that could lead to good or bad results.

There is a definite tradeoff here because both exploration and exploitation are necessary at times. Early in the learning process the agent needs to explore to find out which actions are ideal in different situations. Even "mature" agents need to explore if they live in constantly-changing environments. Exploitation is equally essential; an agent that always takes exploratory (i.e. random) actions will never improve. Often it is important to use the current best policy.

Currently there are only a few standard solutions to this problem. One is the "ε-greedy" method. Most of the time, the agent chooses its best known action (according to its learned policy). Every once in a while (with probability ε), it instead chooses a random action. Another method is the "softmax" action selection method. Instead of choosing from among all actions equally during an exploratory move, softmax methods assign each action a different probability of being chosen at each state. The best actions are given higher probabilities than poor actions. These probabilities become the parameters that are adjusted during policy learning.

Large State Spaces

The methods we have covered up to this point solve reinforcement learning problems effectively, but in order for them to be practical in real situations, they need compact state representations. The simplest methods assume that each state is represented as a single entry in a table. This is obviously impractical for agents operating in large, continuous state spaces. A robot with only 10 continuous sensors, each one discretized into 10 different values, would require a table of 10^{10} unique entries (i.e. states). There are way too many states in this case for the agent to test and evaluate each one individually.

The solution to this problem is to represent states more compactly with function approximation. Instead of keeping track of each unique state separately, we seek to find a function that approximates the state space with a small number of adjustable parameters. The number of parameters is usually much smaller than the number of unique states. The downside is that each state is never represented exactly since we are only *approximating* the state space. Function approximation also allows generalization to unseen data. This feature is important because an agent in a continuous environment will probably never experience the same exact state twice. Using an approximate function of the state space, it can sample a few states and generalize about the rest.

It is important to note that temporal difference learning with linear function approximation will provably converge to the optimal solution[1]. (Convergence is still questionable with nonlinear function approximation, such as backpropagation with multilayer neural networks.) Also, there is only one optimal solution in the linear case, so we need not worry about converging to local maxima. For these reasons we will focus our discussion on linear representations. With this in mind, now the basic problem is to represent the state space in a linear form. To illustrate this problem we will discuss a few examples that use discrete inputs. Later we cover methods that handle continuous inputs.

To provide an accurate value estimation, at first it appears that all we need to do is learn a linear combination of the sensory inputs. Say we have a robot living in a cold climate that enjoys hot cocoa and being indoors where it is warm. This robot has two sensors: one that detects the presence of hot cocoa, and one that detects whether the robot is indoors. Figure 7 shows a simple neural network representing this robot's value function.

---

[1] The convergence proofs do have a few other requirements, such as using a learning rate that decreases over time (here we will use a constant learning rate and not worry about achieving the exact optimal solution) and other assumptions that do not affect the discussion in this thesis.

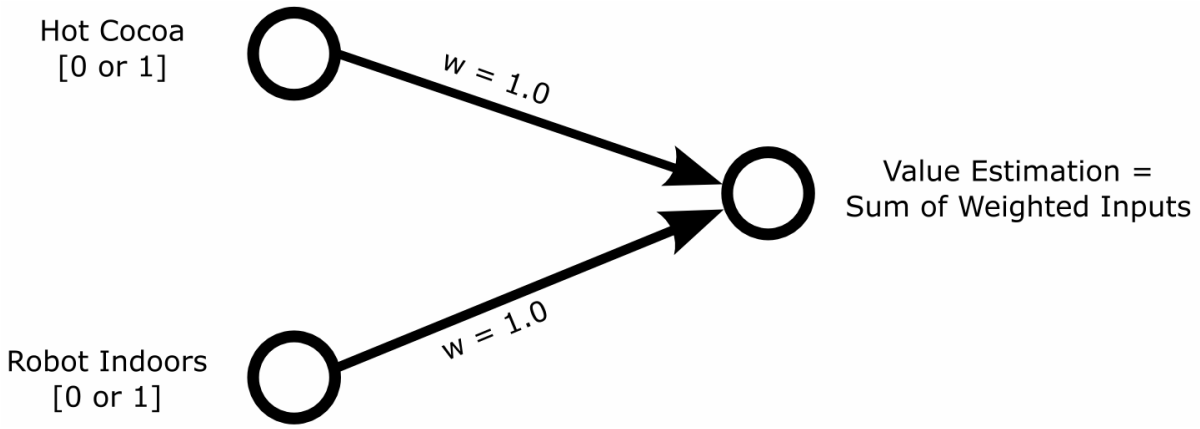

Figure 7: A linear value function approximation for a task with independent inputs.

In this case the presence of either input is *always* good. The values of all possible states are: no hot cocoa, robot not indoors: value = 0.0; no hot cocoa, robot indoors: value = 1.0; hot cocoa, robot not indoors: value = 1.0; hot cocoa, robot indoors: value = 2.0. The neural network can represent this value function easily. Now imagine a different robot that also lives in a cold climate. This robot has two sensors: one that detects whether a fire alarm is sounding, and one that detects whether the robot is indoors. It is good to be indoors unless the fire alarm is sounding. In this case the value of one input depends on the presence of the other. See Figure 8.

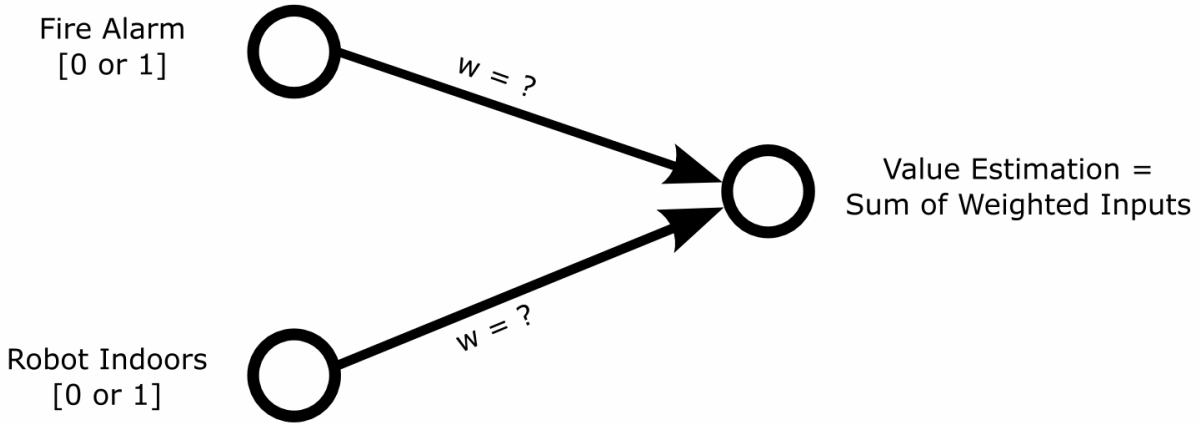

Figure 8: A linear value function approximation for a task with dependent inputs. The weights are not shown here because this value function is impossible to represent with a linear neural network.

For this example a set of possible state values would be: no fire alarm and robot not indoors: value = 0.0; no fire alarm and robot indoors: value = 1.0; fire alarm and robot not indoors: value = 1.0; fire alarm and robot indoors: value = 0.0. There is no way to represent this value function linearly[1]. Nevertheless, it is essential to use a linear combination of sensory inputs to assure convergence. What we need is an intermediate state representation that combines sensory inputs into a set of more complex features. Figure 9 shows this visually. Note that only one feature is active at once. This enables learning updates to be localized in the neural network to a small set of connections.

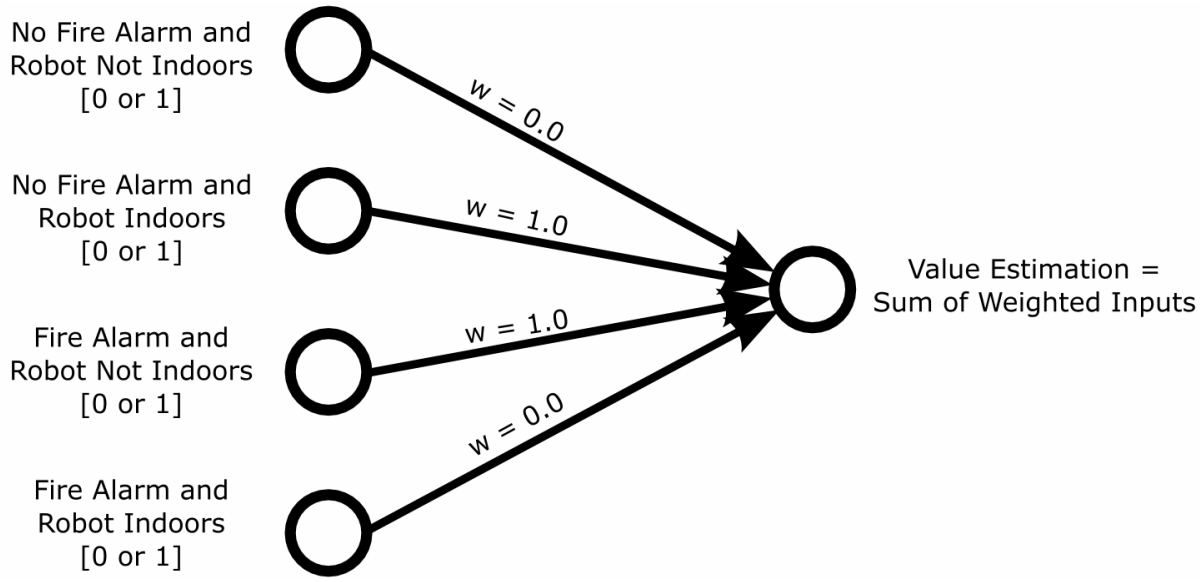

Figure 9: Linear function approximation with unique complex features. Only one feature can be active at once.

If all inputs are discrete values, as in these examples, we can simply enumerate all possible combinations of inputs into an exhaustive list of features. In some cases we will have continuous input values (e.g. readings from thermometers, accelerometers, cameras, etc.). These continuous inputs could simply be discretized into a finite set of inputs and handled as

[1] This problem is known as the XOR problem which is impossible to solve with linear neural networks.

described above. However, this method does not exploit the benefits of function approximation (e.g. it does not generalize to unseen states).

We can use radial basis functions (RBFs) to approximate continuous state spaces. This method uses a set of (usually Gaussian-shaped) curves to approximate a function in any number of dimensions. Each RBF is given a position in the input space. It responds to input data points based on its Euclidean distance from the points. The activation function for Gaussian RBFs is the following:

$$a = e^{\left(\frac{-\|i-c\|^2}{2\sigma^2}\right)}$$

where a is the activation level (between 0 and 1), i is the input data point, c is the RBF's center position, and σ is the RBF's "width" (i.e. the distance of one standard deviation from the center). The quantity in the numerator of the exponent represents the Euclidean distance from the RBF center to the input data point, which could be in a space of any number of dimensions. The collective effect of an array of RBFs is demonstrated in Figure 10. A single continuous value, even in 1-dimensional space, can be represented with an array of RBFs[1].

[1] In biological systems this is similar to population coding: a given quantity is encoded in the combination of activation levels from a population of neurons.

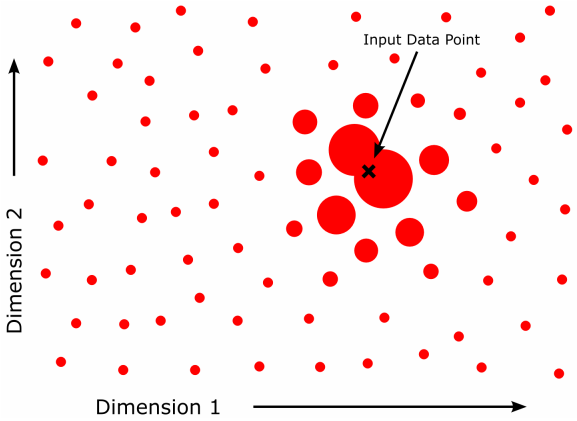

Figure 10: An array of radial basis functions in 2-dimensional space representing an input data point. Each RBF here is a separate circle with a diameter proportional to the RBF's activation level. (The more distant RBFs would actually have a near zero diameter.)

When representing a continuous value, only a few RBFs near the value are active (similar to the discrete representation in Figure 9 where a single feature is active at once). This allows localized learning which is important for learning with function approximation. With other methods, changing a single parameter could affect the entire approximation.

In the general case we can create an array of RBFs that combines all sensory inputs into a single, massive state representation. Any given point in this space would represent a unique combination of sensory inputs, approximated by a set of RBFs in close proximity. Every RBF in this array is essentially a complex feature (e.g. a feature for a car-driving agent could represent "steering angle = 0.3 deg, velocity = 105 km/hr, 12 liters of fuel left, 58 km to the next gas station, 103 m following distance from the car ahead"). If we had some knowledge of the task being performed, we could hand-design features to fit the task. We may not need to combine *all* sensory inputs; we could just combine those that are highly dependent (as described in the examples above), in which case there would be separate RBF arrays. Since we are designing a general purpose system, this is not an option: we must combine all sensory inputs. The main drawback of this approach is that the runtime performance suffers. Computing the RBF state representation grows slower exponentially with the number of inputs being combined because the total number of RBFs is equal to k^n, where k is a constant,

and *n* is the number of inputs. It might help to start with the exhaustive representation and later remove those RBFs representing input combinations that rarely get used[1].

Figure 11 shows our agent design with an additional module that processes sensory input data into a more effective RBF state representation. It also makes explicit the use of TD errors to train the value function and policy.

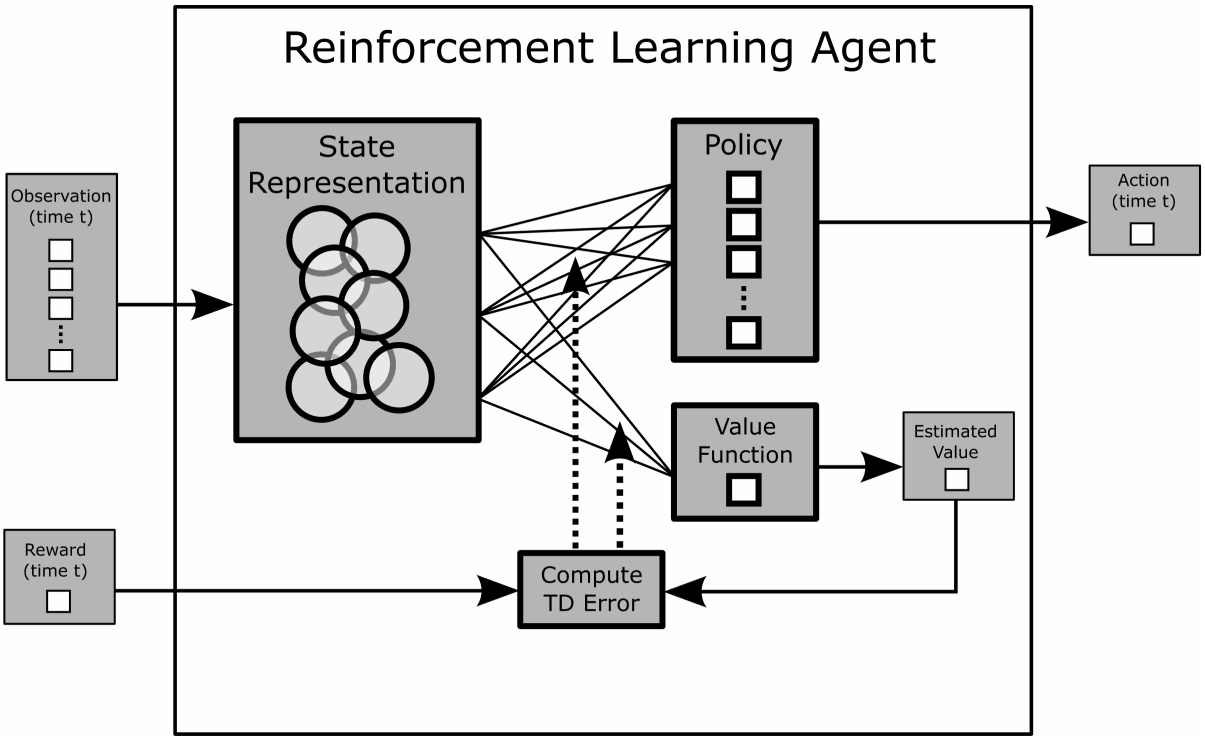

Figure 11: An agent that processes incoming observations into an internal state representation which provides more informative features. This agent uses linear neural networks to represent the value function and policy. The prediction error has been replaced with a more specific "TD error" signal which trains the neural networks.

[1] This may be what occurs in biological brain: we start out with many more connections than we need, and we lose connections that rarely get used.

Efficient Learning from a Few Trials

Ideally, a reinforcement learning agent should be able to learn its value function and policy from very few trials. For this to be possible, the agent must efficiently use any knowledge acquired through direct interaction with its environment.

The idea here has roots in dynamic programming (the reinforcement learning method that assumes a complete model of the environment, including state transitions and reward receipt). With an internal model of the external environment, the agent can test different actions against the model and learn from the expected outcomes without having to interact with the real world. This process goes by many names, including "planning," "imagining," and "using simulated experiences." All of these terms convey the same intriguing idea: that an agent can perform simulated scenarios within its mental model of the environment.

Of course a general purpose agent will have no initial knowledge of its environment. It must learn the model through experience. Grzeszczuk, Terzopoulos, & Hinton (1998) trained neural networks to predict the motion of physical objects. By replacing traditional dynamics simulation with a trained neural network, they were able to reduce the computational requirements necessary for animating digital characters and objects. The underlying concept is interesting because it shows that it is possible to learn predictive models of continuous physical environments.

Figure 12 shows a diagram of a new component we are adding to the agent design. This "predictive model" takes a state and action as input and outputs a state and reward. Based on the input state and proposed action, it tries to predict the next state and reward signal. The actual next state and reward are used to generate prediction errors which train the predictors. It also outputs an uncertainty value concerning its predictions. The uncertainty estimator tries to predict the combined mean squared error of the observation and reward predictors; it is then trained using the actual mean squared error.

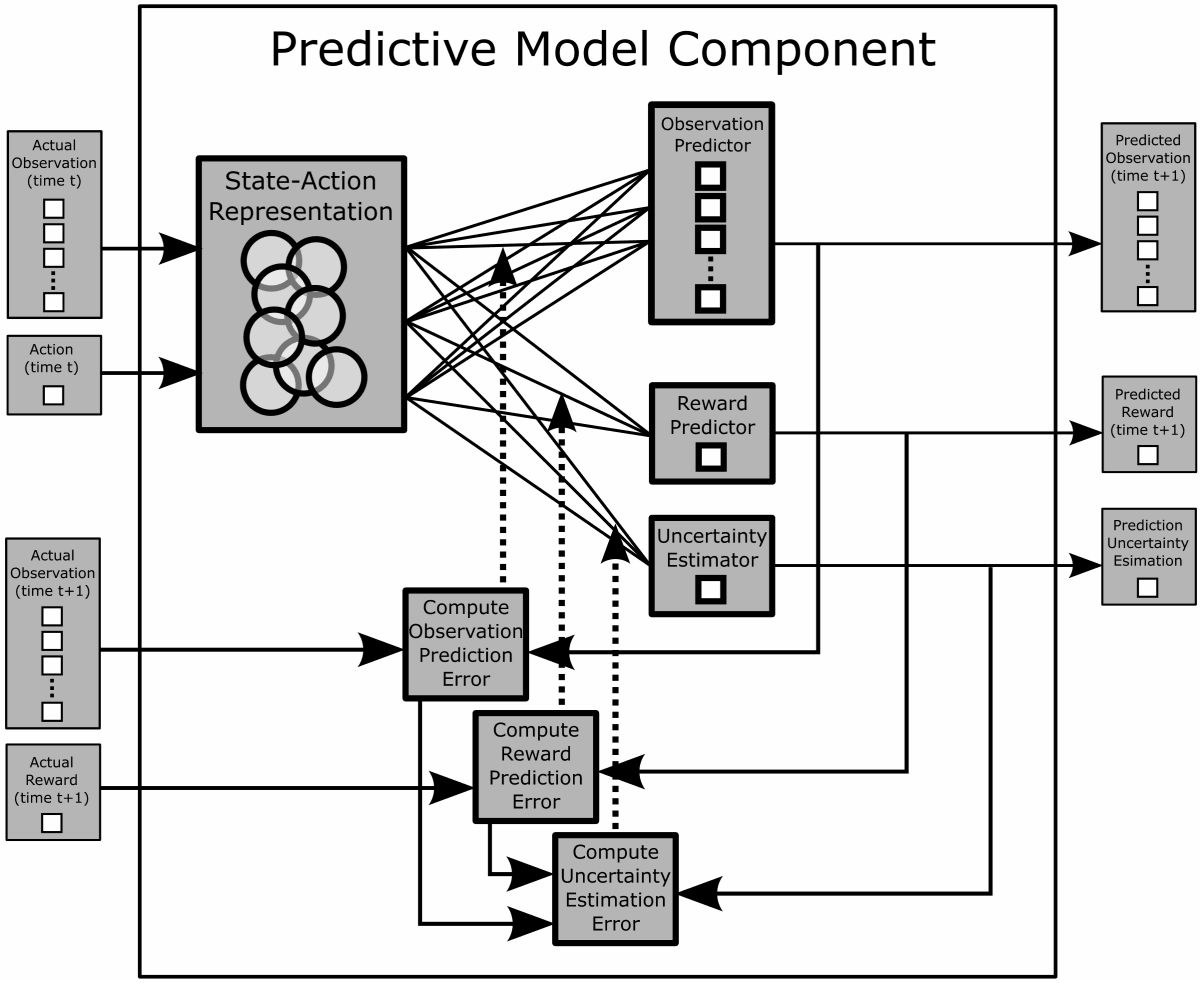

Figure 12: A predictive model that predicts the next observation and reward based on the current observation and action. This model trains itself by computing prediction errors between the actual and predicted values. This also maintains an uncertainty estimate for its own predictions.

We now have an agent design with two main components: the existing reinforcement learning component and the predictive model. Instead of taking observations and rewards directly from the environment as usual, the reinforcement learning component now gets this information from the predictive model. The value function and policy performance are thus entirely dependent upon the accuracy of the predictive model.

This agent has two modes of operation: model learning mode and planning mode. Model learning, shown in Figure 13, trains the predictive model only. The predictive model

constantly tries to predict the current observation and reward based on the previous observation and action. Its performance is improved by computing prediction errors from the actual current observation and reward.

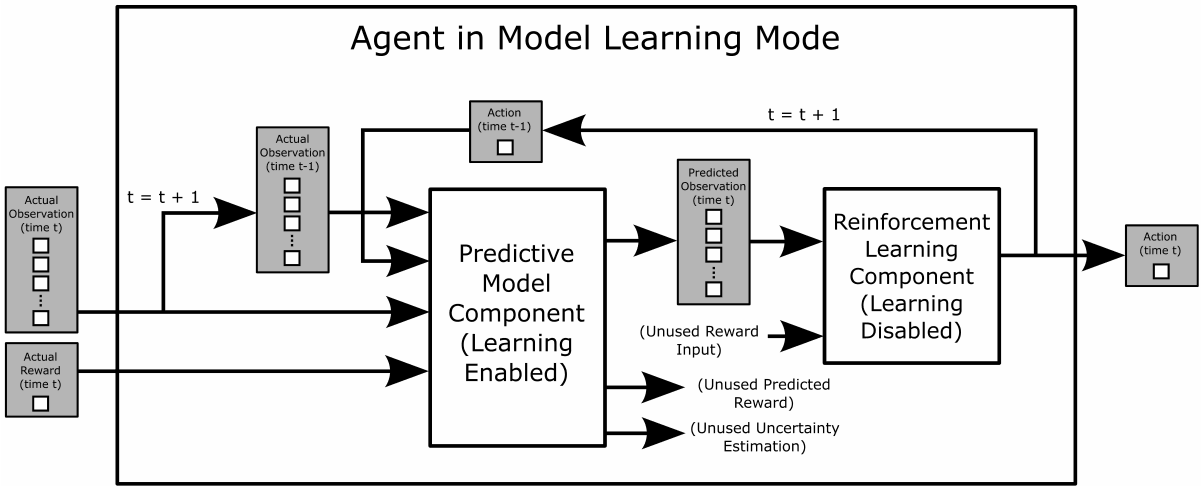

Figure 13: An agent learning an internal predictive model. In this mode the agent only trains the predictive model, not the reinforcement learning component.

Planning mode, shown in Figure 14, trains the reinforcement learning component using the predicted information. Essentially, the predictive model can replace the actual environment for the purpose of reinforcement learning. This allows the agent to think through long sequences of hypothetical interactions with the world, improving its value function and policy from simulated experience. Entire "planning trajectories" proceed by continually feeding actions back into the predictive model. The uncertainty estimation is used to determine the length of these sequences: when uncertainty about the predictions rises above a certain threshold, the agent stops planning: there is no point in planning when uncertainty is too high.

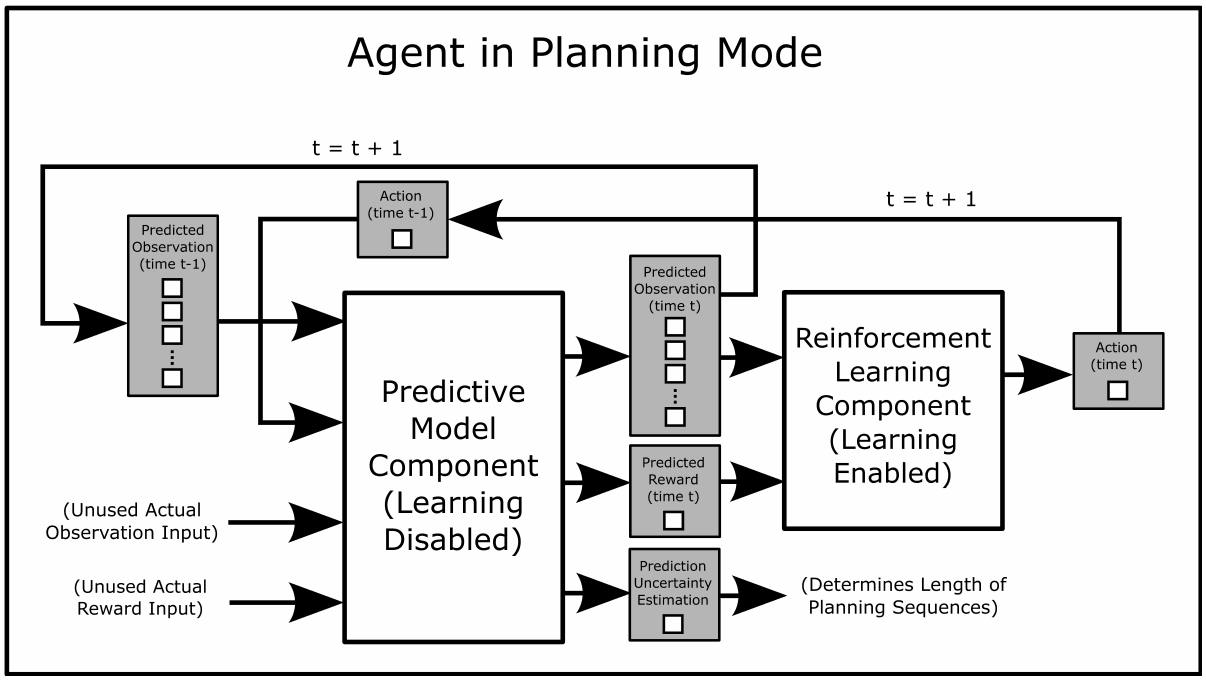

Figure 14: An agent in planning mode. Here, the agent is training its reinforcement learning component by using the predictive model's predictions. This can iteratively step through long planning trajectories in the agent's "imagined" environment. Note that this process occurs without any interaction with the actual environment. Planning sequences end when the prediction uncertainty is too high.

Interestingly, we now have three components learning and interacting concurrently: the value function, the policy, and the predictive model. Each part's performance indirectly depends on the others.

There is one more issue to consider before finishing our discussion on predictive models. We cannot assume that the agent will learn an accurate model just by going about its usual business. We must provide an intrinsic driving force that makes the agent *want* to improve its model. This is the issue of curiosity. A curious agent is motivated to seek new situations and improve its predictions about the world.

One fairly simple model of curiosity is described in Singh, Barto, & Chentanez (2005). In this model the agents are given rewards for encountering novel states, which can be implemented assuming "novel states" means "poor model predictions." When the model

prediction errors are high, we simply give the agent a small reward proportional to the prediction errors. Over time the rewards will decrease as the predictions improve (i.e. the agent gets bored). This model is interesting from an information theory standpoint: since information is proportional to uncertainty, the agent is drawn to situations containing high amounts of information.

We can add a curiosity reward component to our agent by giving it an extra reward proportional to the predictive model's uncertainty estimation. When the agent is planning, and it comes across a hypothetical situation with high uncertainty, it will receive a curiosity reward, driving it to explore that situation in the real environment. Figure 15 shows an updated diagram with curiosity rewards.

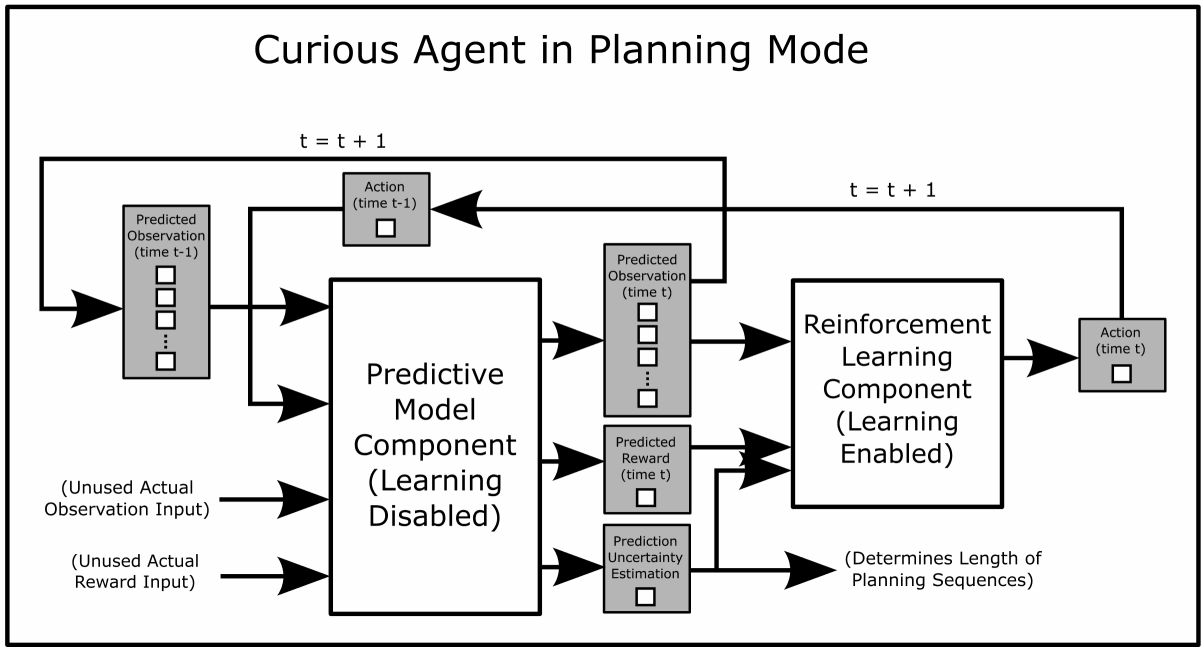

Figure 15: A "curious" planning agent. This agent is given small "curiosity rewards" for encountering novel situations (as determined by prediction uncertainty) which drive the agent to explore unfamiliar states.

Schmidhuber recently proposed a more advanced model of curiosity (Schmidhuber, 2005) that fixes a problem with simple models like the one described above. The problem can be

illustrated with the following example: if a robot encounters a television showing a static signal, it will stare at the screen forever. The random sensory inputs are constantly new, so it will keep receiving curiosity rewards. In Schmidhuber's model, curiosity rewards are proportional to the decrease in prediction errors. Not only must an agent experience a new situation, it must improve its predictions about that situation over time to receive the reward. Thus, the agent will only be rewarded in situations that actually contain predictable information (i.e. it is rewarding to learn). In this thesis we use the simpler curiosity model only. Future work will test the benefits of Schmidhuber's method.

Since the curiosity reward system operates using the same mechanism as other rewards, the agent will learn to seek out these curiosity rewards just as it seeks out normal external rewards. This will help keep the agent's model updated even in changing environments. It will be intrinsically driven to improve its predictions about the world, so it should learn to explore periodically those situations that change the most. In summary, curiosity can be viewed an internal drive whose purpose is to improve the predictive model's accuracy.

Summary

In this chapter we have described some of the most important challenges and tradeoffs that need to be addressed when designing a general reinforcement learner. The following list summarizes these issues:

- Agents must be able to solve the temporal credit assignment problem to be able to reinforce the appropriate actions. We can solve this by learning: 1) a value function that estimates the value of each state, and 2) a policy that chooses the best action to perform at each state. Both of these are trained using temporal difference learning.
- Agents must be able to solve the structural credit assignment problem. They should know which specific structural parts (namely, recent value estimations and actions) should be eligible for modification. Exponentially-decaying eligibility traces are an effective solution.

- Agents should find an appropriate balance between exploiting previous knowledge and taking exploratory actions. The softmax action selection method handles this by maintaining a probability distribution of actions to select in each state.
- Agents should be able to operate within environments with large amounts of sensory input. Rather than maintaining a table of every possible input combination, we can use function approximation with radial basis functions to generate a more compact state representation that generalizes to unseen states. This also lets us represent the value function and policy as a linear combination of the sensory inputs, which is necessary for temporal difference learning convergence.
- Agents should use knowledge gained from direct experience efficiently. By learning an internal predictive model about the environment, agents can "simulate reality" by imagining various situations. This allows them to improve their value functions and policies without having to interact with the environment repeatedly. Furthermore, agents with intrinsic curiosity rewards will be driven to improve their predictive models.

Before we introduce the software implementation of the ideas discussed in this chapter, we will talk about some correlations with biological brains. The next chapter reveals some of the interesting connections already hypothesized between biology and machine learning, especially concerning temporal difference learning.

3 Biological Inspiration

Neuroscientific research is an important source of insight for researchers developing intelligent agents. One of the most compelling reasons to study biological brains is that they are the greatest proof-of-concept that intelligence actually works. An additional benefit of taking this path is that we can design and test computational models which then help contribute to our general understanding of the brain.

The rest of this chapter reviews neuroscientific research using live animals and computational models to study the brain. The focus here is on those aspects that may help in our quest to design general purpose agents.

Reward Prediction

Since our main concern is reinforcement learning, we start by looking at how the brain processes rewards. The main result here is that the midbrain dopamine neurons appear to encode a signal similar to the temporal difference error.

First of all, from a biological standpoint, what is a reward? Cannon & Bseikri (2004) classify rewards into: 1) those that are hedonic experiences, and 2) those sensory cues that have an associated incentive value. The association in category 2 must be *learned* since most sensory cues are initially neutral. Relating these ideas to computational reinforcement learning algorithms, these two categories seem to correspond to direct external rewards and states with high estimated value.

Much of the work done by Wolfram Schultz (Suri & Schultz, 1998; Schultz, 2000; Schultz & Dickinson, 2000; Fiorillo, Tobler, & Schultz, 2003) deals with reward processing in animals. Schultz measured the response of midbrain dopamine neurons in monkeys as they learned to predict rewards. At first dopamine is released only when the reward is received. However, given some reward-predicting stimulus (say, a bell that sounds 1 second before the reward onset), over time the dopamine neurons respond to the reward-predicting stimulus rather than

the reward itself. If the reward is then omitted, the dopamine activity drops below baseline at the time of the predicted reward. These results are striking because they closely match the process of temporal difference learning. In Suri & Schultz (1998) the authors also note that the actor-critic model resembles the structure of the basal ganglia: the nigrostriatal dopamine neurons correspond to the critic, and the striatum corresponds to the actor.

For rewards to affect future behavior, they must first be detected and predicted (Schultz, 2000). Dopamine neurons in certain brain areas, including the substantia nigra, are sensitive to rewarding sensory events. An interesting fact is that these rewarding events are sensed through the same sensory modalities as all other sensory events (i.e. there are no special "reward sensors"). An association is learned that maps sensory inputs to rewards, enabling the prediction of future rewards. The learning rate of this association depends on the "reward prediction error." A reward occurrence must be surprising for learning to happen; if reward occurrences always match predicted rewards, nothing is learned. It is noted again here that this dopamine prediction error signal strongly resembles the temporal difference learning algorithm.

Various reward systems in the brain encode signals related to rewards. These signals are an essential part of goal-directed behavior through trial-and-error learning. Schultz & Dickinson (2000) give a good overview of how brains learn through prediction errors. This kind of learning allows individuals to "adapt to the predictive and causal structure of the environment" (p. 474). Prediction-based learning and behavior can take place over large time scales; events that occur now may affect behavior in the distant future. The generation and use of prediction errors "may contribute to the self-organization of goal-directed behavior" (p. 495). Notably, attention is proportional to prediction errors. If an event is surprising, it elicits more attention from an individual. During exploratory learning, these surprising events guide individuals to spend more time focused on those situations where their predictions are inaccurate.

Schultz & Dickinson (2000) also describe several brain structures that encode prediction errors related to rewards, punishment, external stimuli, and behavioral reactions. Dopamine neurons respond to two types of events: attention-inducing and reward-related stimuli. Norepinephrine neurons respond to reward-predicting stimuli (i.e. the events signaling that a reward is imminent), but not to primary rewards. This suggests that brains are able to form chains of reward-predicting stimuli that eventually lead to primary rewards. Climbing fibers, stretching from the inferior olive to Purkinje neurons in the cerebellum, output error-driven teaching signals. This activity constitutes a prediction error in motor performance. Neurons located in the intermediate layer of superior colliculus respond to the difference between current and future eye positions. Active neurons in the striatum respond to reward unpredictability, encoding a measure of uncertainty about the valuation of a particular state. A distinction is made between the function of two types of reward systems: those that output global signals to large neuron populations and those that influence very select groups. Each type of system might use a different learning mechanism to respond to these signals.

The general conclusion of Schultz & Dickinson (2000) is that predictions are a fundamental brain function. The two main purposes of predictions are 1) to bridge the temporal gap between event occurrence and future consequences (i.e. giving advance information that informs individuals about the future), and 2) helping to evaluate outcomes by comparing the actual state with the predicted state (e.g. to improve predictions by learning from errors).

Although there is much evidence that the neurotransmitter dopamine is used as a reward signal in the brain (Schultz, 2000), there is not yet a strong consensus. The focus of Cannon & Bseikri (2004) was to question whether dopamine is actually required for hedonic rewards to be processed in the brain. Dopamine-deficient mice were tested to perform reward-related behaviors. Even in the absence of dopamine, the mice were still able to distinguish between rewarded and unrewarded behaviors, though not as well as normal mice. This appears to oppose the idea of dopamine as a reward signal. However, the authors suggest that the dopamine-deficient mice may still have had reduced motivation or physical or mental ability

to obtain the reward. The conclusion was that there is still no strong evidence against dopamine being used as a reward signal.

Dopamine neurons that encode reward prediction errors may simultaneously encode a degree of uncertainty about predicted rewards (Fiorillo, Tobler, and Schultz, 2003). The reward prediction error would then measure the discrepancy between the actual reward occurrence and the predicted *probability* of reward occurrence. The measure of uncertainty is maximal when the predicted probability of reward occurrence is 0.5 and minimal when this probability is 0.0 or 1.0. The dopamine neurons encode the degree of uncertainty in their sustained activity; the peak sustained activity occurs at the time of potential reward which is the moment of greatest uncertainty. The reward probability itself is encoded in the neurons' phasic activity. It is suggested that attention is proportional to uncertainty – we pay more attention to uncertain events. Uncertainty might even reinforce risk-taking behavior to elicit exploration.

Suri & Schultz (1998) present results from a computational model (specifically, temporal difference learning) of dopamine neurons that encodes reward prediction errors. They simulate a neural network in an actor-critic architecture that learns to map a sequence of stimuli to appropriate actions. A sequence of 7 characters is presented to a neural network. The task was to map correctly each input character to a specific output character (i.e. map each state to a specific action). A reward is received only after giving correct outputs for all 7 inputs. Each state is represented as a temporal sequence of input signals. The neural net is trained by starting from the last character in the sequence and working backwards so that the task becomes increasingly more difficult[1]. Eligibility traces are used to extend synaptic modifications beyond the current activation. The model quickly learns the entire sequence using the temporal difference reinforcement signal. When the actions are reinforced by the external reward alone (i.e. without learning a value function), performance quickly degrades.

---

[1] Experimental results from the same task are given later in this thesis.

They suggest that after conditioning, reward-predicting stimuli essentially become rewards themselves.

It has been mentioned by several authors that brain reward signals are very similar to modern computational learning models (Schultz, 2000; Montague, Hyman, & Cohen, 2004). In Montague, Hyman, & Cohen (2004), the authors compare recent findings on dopamine function with computational theories of reinforcement learning. One underlying idea is that behavioral control is inseparable from the valuation of objects and circumstances. (In other words, value functions are necessary for behavior.) Specifically, the midbrain dopamine neurons are thought to be a key component in that they encode reward signals. The way in which we humans assign values to different situations is determined by these dopamine systems. Most notable is the statement that the computational theory of reinforcement learning "has informed both the design and interpretation of experiments that probe how the dopamine system influences sequences of choices made about rewards" (p. 760). In terms of neuroscientific research, reinforcement learning creates a framework for connecting the effects of specific neuromodulatory systems to behavior.

Sensory Prediction

It seems necessary that biological brains be able to predict sensory information. Given a set of sensory inputs, the brain can predict what the world will be like after some time. This ability enables the imagining/planning features that we discussed in the previous chapter. We now look at models of how the brain predicts sensory information.

Sensory prediction entails calculating the discrepancy between actual and predicted sensory inputs. Rao & Ballard (1999) discussed the idea of predictive coding where unexpected sensory information is passed to higher levels of sensory processing. This paper presents a model of the visual cortex based on sensory prediction. The model is arranged hierarchically with higher levels passing predicted inputs to lower levels and the lower levels sending prediction errors to higher levels. The prediction errors are computed from the difference between predicted and actual inputs. These errors then train the prediction-generating

components in the higher levels. The authors use the model to simulate several extra-classical receptive-field effects like end-stopping (cells that have maximal responses to oriented bars of certain lengths).

Predictive coding models like that of Rao and Ballard are very efficient at transmitting information. Only the unexpected events travel up the hierarchy. This is ideal from an information theory standpoint since only uncertain events contain information.

Another example of sensory prediction is described in Flanagan, Vetter, Johansson, & Wolpert (2003). This study gives evidence that we learn to predict the outcome of our actions before we learn to control the outcome itself. The two parts of motor control being studied were prediction (i.e. mapping motor commands to expected sensory inputs, a forward model) and control (i.e. mapping desired outcomes to the motor commands required to produce those outcomes, an inverse model)[1]. Test subjects were asked to move an object along a straight line while the object was being affected by an unpredictable force. The test subjects' grip force on the object represented their prediction of its motion, while their hand trajectories represented their control of the object. The results showed that subjects learned to predict (forward model) 7.5 times faster than they learned to control (inverse model).

Conclusions

There are already intriguing connections between neuroscience and machine intelligence research. In particular, midbrain dopamine neuron activity closely resembles the temporal difference prediction error signal. The established theories of reinforcement learning provide computational models for further brain research. The brain itself provides inspiration for new machine learning algorithms. Hopefully these connections will become stronger in the future and lead to a fuller understanding of intelligence in general.

---

[1] More details about these different models can be found in Jordan (1996).

The idea that reward processing in humans can be understood as a temporal difference learning process has several interesting implications. First, it could help to explain drug abuse behavior from a computational standpoint. There is evidence that dopamine neurons encode rewards that differ from a baseline level. If powerful addicting drugs set the baseline reward level incredibly high, all situations that normally provide some level of satisfaction fall well below baseline. Thus, the only interesting situations are those that lead to more drug intake. Second, it may be possible to tap into the brain's reward processing centers to create extremely adaptive user interfaces for computing devices. For example, imagine a personal computer augmented with biofeedback device measuring the user's dopamine neuron firing rates (noninvasively, ideally). The computer would have its own reinforcement learning agent whose rewards are proportional to the user's dopamine level. This would allow the computer to optimize the interface automatically into the most satisfying permutation for the user. Finally, it may seem almost too mechanistic to think of the complex human mind as a machine greedily planning its actions to acquire more rewards. However, this view seems less extreme if we take a broader view of rewards in general. We might include any of the following: food, sex, drugs, money, feeling loved, feeling respected, completing a difficult task, learning interesting information, feeling relief, etc[1]. These various sources of rewards might converge onto a single processing center, yielding something similar to the scalar reward value described in the previous chapter. In conclusion, it seems reasonable that the brain is trying to maximize reward intake by modifying its behavior with a temporal difference learning mechanism.

[1] For most of these it is hard to say if the situation is a hard-wired, direct reward (like food) or if it is a learned association (like money) that merely implies direct rewards.

4 Implementation

Now we will compile many of the ideas from chapter 2 into a concrete software implementation. The goal here is to provide a practical tool for use in real applications. Its main intended users are roboticists and game developers that need an out-of-the-box solution for their learning tasks. This tool is distributed as a free, Open Source software library, which provides several benefits: 1) the "free" aspect will help the software circulate faster and gain more exposure, and 2) the Open Source aspect enables users to study the implementation details, gaining practical insight on how reinforcement learning and curiosity-driven planning work. The current implementation is available online (Verve http://verve-agents.sourceforge.net). Also included in the library distribution is an array of test applications (including the ones used in chapter 5 of this thesis) that validate the library's usefulness and provide example source code.

Verve is a cross-platform, object-oriented library written in C++. It is built as a shared library (i.e. a ".dll" file in Windows, or a ".so" file in UNIX). It is organized as a set of classes, the main one being the Agent class. Typically, users create an AgentDescriptor object, which describes the general structure of an Agent, and set its various parameters (e.g. number of sensors, number of actions, sensor resolution, whether planning is enabled, etc.). Then they create an Agent object from the AgentDescriptor. Another way to create an Agent is by loading a saved Agent from an XML file.

Saving and loading Agents to and from XML files provides several benefits. Potentially long training sessions (lasting several days) can be saved at regular intervals to protect against power failures. Also, once an Agent has reached a desirable level of proficiency (i.e. has finished its training phase), it can be stored for practical use. In this case it can be helpful to disable learning once training is complete. This saves computational resources because the entire learning system is ignored (only the policy is used), and it enables more repeatable behavior.

To increase immediate usability, all free parameters use default values that were found experimentally to be useful in a variety of learning tasks. Adjusting some parameters manually may improve learning performance (e.g. some of the experiments in this thesis use very high policy learning rates), but this effect is more noticeable on simpler tasks that do not require much exploration.

The following is a current list of the library's major features:

- Agents learn to solve reinforcement learning tasks using temporal difference learning with eligibility traces in an actor-critic architecture.
- Agents use a dynamically-growing radial basis function state representation. This combines sensory inputs into higher-level features and allows generalization to unseen inputs. The state representation grows dynamically, allocating resources for new states as they are experienced.
- Agents use a softmax action selection scheme which maintains separate selection probabilities for each action.
- The library can be used for discrete environments or for real-time control in continuous environments. It is ideal for games and robotics.
- Agents can use any number of sensory inputs and actions. Sensors can be discrete (e.g. battery power is low, medium, or high) or continuous (e.g. a distance value returned by a laser rangefinder). Continuous sensors have a "resolution" setting which determines their acuity.
- Agents learn a predictive model of their environments through experience. This increases their learning performance by allowing them to learn from simulated experiences (i.e. planning).
- An internal uncertainty estimation automatically determines the length of planning sequences. (If uncertainty is too high, there's no point in continuing planning.)
- Agents use a model of curiosity which drives them to explore new situations. This helps them to improve their predictive models.

- Once an agent learns a task proficiently (i.e. finishes its training phase), learning can be disabled to save computational resources.
- Agents can be saved to and loaded from XML files.
- The distribution includes Python bindings (generated with SWIG).
- The library is unit tested.
- The source code is heavily commented.
- The distribution includes the ability to output value function data to a text file for visualization. It also includes a separate application to generate PNG image files from value function data for agents with either one or two sensors.

There are several limitations in the current implementation (which will be addressed in future work, as described at the end of this thesis). The following is a list of the major limitations:

- Computational space and time requirements grow exponentially with the number of inputs. This is mainly due to the state representation that combines all inputs into a higher level representation.
- Agents learn to select from a finite number of actions, but they do not learn continuous control signals. The action set must be predefined by the user. Future implementations will autonomously learn continuous action signals instead of simply acting as a switching system.
- Agents have no temporal state representation, so they cannot predict future events at specific times.

Most of the features of this library are designed to solve the problems introduced in chapter 2. The general architecture is same as the one developed in that chapter. More specifically: the value function and policy are stored as separate data structures (i.e. an actor-critic architecture) approximated with linear neural networks and are trained through temporal difference learning; the state representation uses radial basis functions to combine sensory inputs; actions are chosen using a roulette selection scheme; the predictive model uses linear neural networks trained with a delta rule; and the agent gets curiosity rewards to help it

improve its predictive model. Agents use discrete and continuous sensors. Discrete sensors take an index representing one of several distinct values, and continuous sensors take any value between -1 and 1.

One interesting detail is that a few free parameters are specified as time constants. This stems from the fact that Agents are updated in real time (i.e. each update step takes a time delta that specifies how much time has elapsed since the previous update). The usual way of setting a neural network learning rate parameter, for example, is by using a constant value that affects how far each weight is adjusted *per update*. A learning rate of 0.1 attempts to reduce the overall error by 10% per update. However, since each update in our case represents a certain amount of real time, we would rather let users set how much error is reduced *per second*. Time constants let us specify how long it takes (in seconds) for errors to be reduced by 63%. For example, a learning rate time constant of 0.1 s attempts to reduce errors to 37% of their initial values after 0.1 s, regardless of size of the Agent update time delta.

The rest of this chapter covers more specific implementation details. For more information, the (heavily commented) source code is available from the Verve website.

Linear Neural Networks

The neural networks used in Verve are quite simple. Each one is linear, so there are no "hidden" layers of neurons. Every neuron's activation (i.e. firing rate, i.e. output) is computed from its weighted input sum. The following equation shows how this input sum is computed:

$$i_t = \sum \vec{w}_t \vec{x}_t$$

where i is the neuron's current weighted input sum, $\vec{w}$ is a vector of the current input connection weights, and $\vec{x}$ is a vector of the current input activation. The neuron's activation value is then computed from this input sum using an activation function. Almost

all neural networks in Verve use the linear activation function $f(x) = x$ where x is the neuron's input sum. Only the policy neural network differs. Its neurons use a sigmoid "squashing" function which constrains the activation to be within 0 and 1. The following is the typical sigmoid activation function which is being used here:

$$f(x) = \frac{1}{1 + e^{-x}}$$

The final activation of each neuron is the following:

$$y_t = f(\sum \vec{w}_t \vec{x}_t)$$

where y is the neuron's current activation, and f is either the linear or sigmoid activation function.

Temporal Difference Learning

The value function and policy are both represented with linear neural networks. The activity of the input neurons (in the state representation) ranges from 0 to 1. The value function uses a single neuron to represent the estimated value. This neuron has unbounded activity because the value estimation should not be bounded. The policy uses a separate neuron for each action. The activation of each policy neuron, determined by its input connection weights and the current state, represents its action selection probability. On each update, a roulette selection scheme chooses the next action based on these probabilities. The chosen neuron has its activation set to 1, and the rest are set to 0[1]. We initialize the value function's and the policy's weights to small values near zero. This makes the initial value estimation of each state close to zero, and it ensures that each action has an equal selection probability initially.

---

[1] This is known as a "winner-take-all" method.

We now describe the various update rules in detail. As we described in chapter 2, the TD error, δ, is computed as follows:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

We then use the TD error to update the connection weights in the value function and policy neural networks:

$$w_t \leftarrow w_t + \eta \delta_t e_t$$

where η is the learning rate for either the value function or policy (depending on which is being updated), and e is the current eligibility trace of weight w. This rule basically performs gradient descent on the weights using the TD error. The learning rates for the value function and policy, η_{value} and η_{policy}, are specified with time constants as follows:

$$\eta_{value} = \frac{1 - e^{-\frac{dt}{\tau_{value}}}}{sum}$$

$$\eta_{policy} = k \eta_{value}$$

where dt is the elapsed time between updates, *sum* is a factor dependent on the RBF resolution (see the section on the RBF state representation), τ_{value} is the value function learning time constant, and k is the policy learning multiplier. This multiplier affects how long exploratory behavior lasts during training. Remember that the policy neurons use the monotonic sigmoid activation function. This function has a smoothed step-like curve near $x = 0$, but it outputs values close to 0 or 1 for most of its range. A high policy learning multiplier can quickly polarize the action selection probabilities, usually driving one action's probability to 1 and the rest to 0. This can be beneficial once the ideal action is known, but if it happens too early, it can destroy exploration.

The eligibility traces of the value estimation connections are increased as follows:

$$\text{if } x > e_t,\ e_t \leftarrow x$$

where x is the connection's input neuron activity. This increases the traces of the connections coming out of the active RBF neurons, making their weights eligible for modification. The eligibility traces of the policy's connections are increased as follows:

$$\text{if } xy > e_t,\ e_t \leftarrow xy$$

where x is the input neuron activity, and y is the output neuron activity. This rule makes the selection probability of the current action neuron eligible for modification. All traces are decreased exponentially according to the following rule:

$$e_t \leftarrow \gamma\lambda e_t$$

We use time constants to specify the discount factor, γ, and the eligibility trace decay constant, λ:

$$\gamma = e^{-\frac{dt}{\tau_{disc}}}$$

$$\lambda = e^{-\frac{dt}{\tau_{elig}}}$$

We limit the external reinforcement signal to be between -1 and 1. This is to ensure that users do not try to use the reward magnitude to affect learning performance. (Instead, they should adjust the learning time constants.) Higher reward magnitudes can give higher initial TD errors which polarize the policy's action probabilities more quickly. With that said, it appears to be most effective to use the full reward range (e.g. from -1 to 1 instead of just 0 to 1). This seems to produce faster learning.

RBF State Representation

Each radial basis function in the state representation is a separate neuron with activation proportional to its Euclidean distance from the input data point (as described in chapter 2). The only extra complication here is that we wish to combine discrete input sensors with continuous ones. We do this by maintaining a separate RBF array for each possible discrete input combination. For example, say we have a robot with a discrete sensor that detects whether it is daytime or nighttime and several continuous sensors that detect its orientation in 3D space. The robot's agent would create two RBF arrays: one representing orientation during the day, and one representing orientation at night. In the special case when there are only discrete sensors, each possible input combination uses a separate RBF neuron. These "discrete RBF" neurons are handled specially: they have no center position, and they are maximally active (i.e. activation = 1) when they match the discrete input data (otherwise, activation = 0).

When an agent is created, for each discrete sensor, users specify the number of distinct values that sensor can represent. For continuous sensors there is a global "resolution" parameter that determines how many RBFs span each continuous input dimension. This value also determines the "width" (distance of one standard deviation from the center) of each RBF, which is set to 2 / resolution. The 2 here is the range of each continuous input which varies from -1 to 1. Every continuous sensor has an additional Boolean "circular" parameter that specifies whether the sensor is detecting a circular input range that can jump instantly from -1 to 1 (e.g. the angle of a wheel).

The *sum* parameter, which is used to scale the temporal difference and predictive model learning rates, represents the maximum possible activation sum of all RBFs. The *sum* is set to 1 when only discrete sensors are used because only one neuron is active at once. It was determined experimentally (results not shown) that *sum* should be set to $2.5^{\text{resolution}}$ when continuous sensors are used. The purpose of this is to scale the learning rate automatically with the RBF resolution. For example, it reduces the learning rate when resolution is high (implying more connections from the RBFs to the value function and policy neurons). The

training errors should have the same overall magnitude, but they must apply smaller changes to each connection when there are more connections.

Using RBFs for our state representation has many benefits, but it is also computationally expensive, requiring a number of RBFs that grows exponentially with the number of sensory inputs. To help ease some of this burden, we can allocate RBFs on demand. Whenever a new observation (containing discrete and continuous data) arrives, the system uses the discrete data to locate the correct RBF array. Then it activates the RBFs using the continuous data. Each RBF's activation level is classified as "no activation" if the data point is beyond four standard deviations from the center, "low activation" if the data point is between one and four standard deviations, or "high activation" if the data point is within one standard deviation. The low and high activation RBF neurons are stored in a list (to enable a more efficient training process that trains only the necessary connection weights). If no RBFs have high activation, we assume that the input data point is far away from all existing RBFs. If this happens, we allocate a new RBF and center it on the data point.

Predictive Model

The predictive model uses several distinct "predictor" modules, as diagrammed in chapter 2. These include the observation predictor, the reward predictor, and the uncertainty estimator. Each of these modules uses a separate linear neural network trained with a simple delta rule. The observation predictor is trained from the error between its predicted next observation and the actual next observation. Similarly, the reward predictor is trained from the error between its predicted next reward and the actual next reward. Finally, the uncertainty estimator, which tries to estimate the mean squared error of the observation and reward predictions, is trained from the error between its estimated MSE and the actual MSE. Remember that the predictive model is only trained in model learning mode, not in planning mode. On each update, the agent enters model learning mode to train the predictive model. Then, it starts a planning sequence which proceeds until uncertainty is too high.

We will start by describing what happens in model learning mode. The delta learning rule, which is a supervised learning rule, makes sense for predictive model learning because the actual environment provides real experience that can act as training examples. It must function differently from the gradient descent rule used for TD learning. The value function and policy use a single TD error value that modifies all eligible parameters, but the predictive model neural network has multiple error signals, one for each output neuron. The delta learning rule for linear neural networks is the following:

$$w_t \leftarrow w_t + \eta(d_t - y_t) f'(i_t) x_t$$

where η is the learning rate, d is the desired activation of the output neuron, y is the actual activation of the output neuron, f' is the first derivative of the output neuron's activation function, i is the output neuron's weighted input sum, and x is the input neuron's activation. Since we are using linear activation functions for all output neurons in the predictive model, the learning rule becomes the following:

$$w_t \leftarrow w_t + \eta_{pred}(d_t - y_t) x_t$$

η_{pred} is the predictive model's learning rate. This learning rate is specified with a time constant:

$$\eta_{pred} = \frac{1 - e^{-\frac{dt}{\tau_{pred}}}}{sum}$$

where dt is the elapsed time between updates, sum is a factor dependent on the RBF resolution (see the section on the RBF state representation), and τ_{pred} is the predictive model's learning time constant.

Now we describe details pertaining to planning mode. A "planning step" refers to the complete cycle of selecting an action, using that action and the current observation to predict the next observation and reward, and training the reinforcement learning component with the predicted next observation and reward. Each planning phase consists of a variable-length sequence of planning steps. These sequences can be viewed as trajectories that start from the current actual state and following the agent's policy through imaginary space. This "trajectory sampling" method is conceptually simple, and it focuses learning on the states actually experienced[1].

The predictive model maintains an internal estimate of its uncertainty about any given prediction. The uncertainty here is the mean squared error in the observation and reward predictions. The purpose of this system is to help the agent automatically determine the length of its planning sequences. A particular planning sequence will proceed until either 1) the estimated uncertainty at any given step in the sequence exceeds some threshold, or 2) the maximum planning sequence length is exceeded. The rationale behind this mechanism is that there is no sense in continuing a plan if the agent is too uncertain about the outcome. In the future it might be interesting to use an *accumulated* uncertainty value during planning sequences. Instead of ending a planning sequence when the current uncertainty exceeds a threshold, the sequence would end when an accumulated uncertainty value exceeds the threshold.

The predictive model has two free parameters that affect planning: the maximum number of steps to take during a planning sequence, and the maximum amount of estimated uncertainty to tolerate before ending a planning sequence. The user can set each of these parameters when creating a new Agent.

---

[1] Other methods include random sampling from the entire state space (which is impractical for large state space) and prioritized sweeping (which is currently only practical for discrete state spaces). See Sutton & Barto (1998) for more details on these methods.

The last implementation detail here concerns curiosity. As described in chapter 2, curiosity drives agents to explore novel parts of the state space. It focuses exploratory behavior on the states likely to yield the most information. Curiosity is implemented using the same uncertainty estimations as before. We provide the agent with curiosity rewards at each planning step proportional to the estimated uncertainty for the most recent prediction. The total reward received by the agent is the sum of the usual external reward (which is actually a predicted quantity) and these intrinsic curiosity rewards. Since we use the same reward processing mechanism as usual, this drives the agent towards uncertain states. Over time the agent reduces its uncertainty about these states; its curiosity rewards diminish, causing it to become "bored" and search for other rewards. We can think of the learning process for a curious agent as a boundary of uncertainty continually spreading out with curiosity rewards around the edge.

Code Sample

This section shows source code for a generic Agent training application. The purpose of this is to give a tangible example of how Verve Agents are used in practice.

```
// Define an AgentDescriptor.
verve::AgentDescriptor agentDesc;
agentDesc.addDiscreteSensor(4);  // 4 possible values.
agentDesc.addContinuousSensor();
agentDesc.addContinuousSensor();
agentDesc.setContinuousSensorResolution(10);
agentDesc.setNumOutputs(3); // 3 actions.

// Create the Agent and an Observation initialized
// to fit this Agent.
verve::Agent agent(agentDesc);
verve::Observation obs;
obs.init(agent);

// Set the initial state of the world.
initEnvironment();

// Loop forever (or until some desired learning
// performance is achieved).
```

```
while (1)
{
     verve::real dt = 0.1;

     // Update the Observation based on the current
     // state of the world.  Each sensor is
     // accessed via an index.
     obs.setDiscreteValue(0, computeDiscreteInput());
     obs.setContinuousValue(0, computeContinuousInput0());
     obs.setContinuousValue(1, computeContinuousInput1());

     verve::real reward = computeReward();

     // Update the Agent.
     unsigned int action = agent.update(reward, obs, dt);

     // Apply the chosen action to the environment.
     switch(action)
     {
          case 0:
               performAction0();
               break;
          case 1:
               performAction1();
               break;
          case 2:
               performAction2();
               break;
          default:
               break;
     }

     // Simulate the environment ahead by 'dt' seconds.
     updateEnvironment(dt);
}
```

Sometimes the intent of the application is to train an Agent to achieve a certain level of proficiency for some task, and then use the trained Agent in some other application. To accomplish this, the user can simply disable learning when training is complete.

Summary

The implementation described in this chapter represents the initial version of an ongoing project. Future versions will add more advanced features that aim to remove the current limitations. Also, testing the Verve agents in a variety of real applications will help reveal its strengths and weaknesses and drive further development.

Now that we have a software implementation of our agent design, we can test it in a set of experiments. The next chapter will describe some initial results that test the main features in a variety of learning tasks.

5 Experiments

This chapter provides a set of initial experimental results. The purpose of these experiments is to validate Verve's effectiveness in a variety of tasks and to demonstrate some of the tradeoffs involved in practice.

We start by testing agents in simple discrete environments with the reinforcement learning component only (no planning). Then we look at two physically simulated control tasks, again with the reinforcement learning component only. We then show results for planning agents and for planning agents with curiosity.

Discrete Environment Tasks

The tasks in this section test agents in fairly simple discrete environments using only the reinforcement learning component. Many of the tasks involve 1-dimensional or 2-dimensional discrete grid worlds. We should expect learning performance to be much slower in the 2-dimensional environments simply because there are more states to explore. Although the environments here have discrete states, this does not prevent us from using continuous sensors. We will perform some of the tasks in this section twice: once using discrete sensors, and once using continuous sensors. This will enable a comparison between the value functions and learning performance with both types of sensors. Unless noted otherwise, all experiments in this section use an agent update step size of 0.1 s and a τ_{value} of 0.1 s.

10-Armed Bandit

The 10-armed bandit task is a fairly standard reinforcement learning problem. Alluding to a "one-armed bandit" (i.e. a slot machine), a 10-armed bandit (a specific case of the more general n-armed bandit) is a hypothetical machine with 10 levers. Pulling each lever produces a different amount of reward. There is basically a single state. From that state, the agent must learn to choose the single best action (i.e. pulling the lever with the biggest average payoff). In this experiment each lever's mean reward value is initially set randomly

using a Gaussian distribution with mean 0 and variance 1. When each lever is pulled, it returns a random reward from a Gaussian distribution using its mean reward value and variance 1.

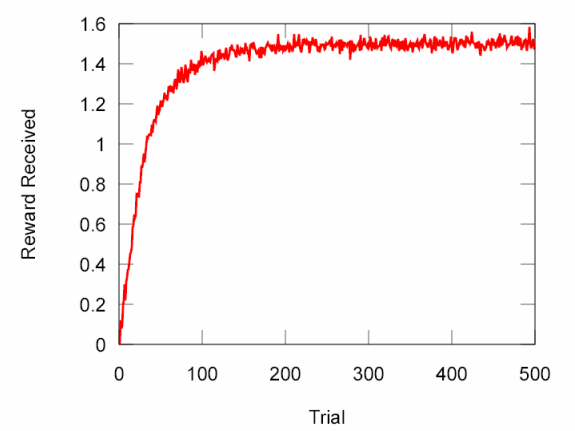

Figure 16: Learning performance on the 10-armed bandit task using the following parameters: policy learning multiplier = 100, τ_{disc} = 0.3 s, number of runs averaged = 2000.

The results of this experiment are shown above in Figure 16. Through exploration over time the agent learns to increase the probability of choosing the action that yields the highest long-term reward sum.

Sequential Action Learning

This test is adapted from Suri & Schultz (1998). The major difference here is that we do not use a temporal stimulus representation. The agent is presented with an input signal and is expected to output a corresponding action signal. If the correct action is selected, the next input signal is presented. Reinforcement is zero for every step except when the agent correctly chooses the final action in the sequence, in which case the reinforcement is 1. Training proceeds in reverse order, starting from the last input/output pair in the sequence and working backwards. The agent is trained on each step of the sequence for 100 trials. Over time this training scheme allows the agent to assign value to the earliest reward-predicting stimulus in the sequence. For more details about the task, see Suri & Schultz (1998).

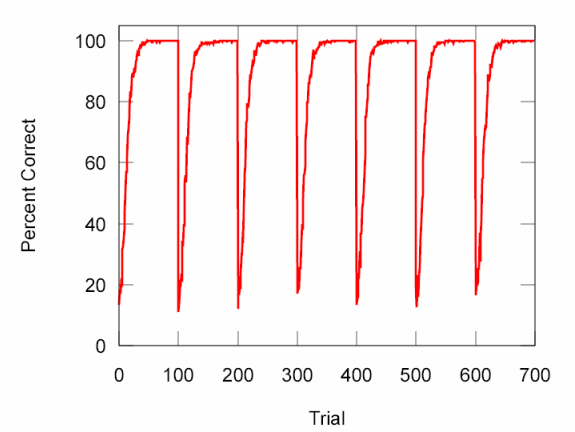

Figure 17: Learning performance on the sequential actions task using the following parameters: policy learning multiplier = 1000, τ_{elig} = 0.2, τ_{disc} = 5.0 s, number of runs averaged = 200.

The results here, shown in Figure 17, are very similar to those in Suri & Schultz (1998). The agent learns to perform the correct actions 100% correctly after about 30 trials. The learning curve for each new block of 100 trials is rougly the same, even though the final block (trials 600-700) requires the agent to perform 7 actions correctly before receiving a reward. The only way this can be accomplished is by learning the correct value and action for each state prior to the reward.

1D Hot Plate

The 1D "hot plate" task involves a robot in a 1-dimensional world with 20 distinct positions. The center of the world (the hot plate) is painful for the robot and gives a -1 reward on each step, but the extreme edges are safe zones that give a reward of 1. It has a single sensor that detects its position in the world and has three actions (move left, move right, or do nothing). The robot must learn to move to one the edges as quickly as possible. As soon as the agent reaches one of the edges, the trial is ended. Performance is measured as the number of steps taken before reaching the goal (one of the edges). Figure 18 plots the learning performance when using discrete and continuous sensors.

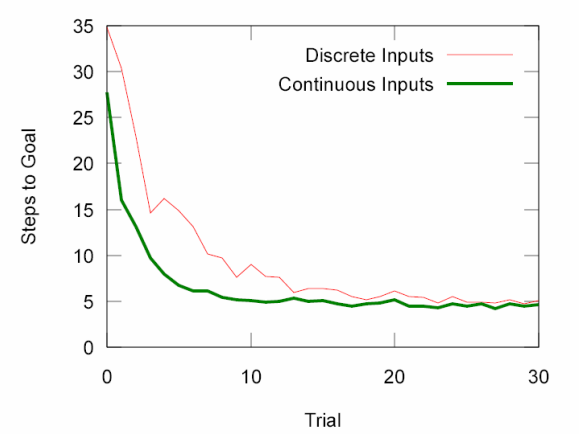

Figure 18: Learning performance on the 1D hot plate task using the following parameters: policy learning multiplier = 10, position input discretization (discrete inputs plot only) = 20, continuous sensor resolution (continuous inputs plot only) = 10, number of runs averaged = 300.

Figure 19 shows a sequence of learned value functions observed at various points during training. The value functions at the end of trial 30 are nearly optimal.

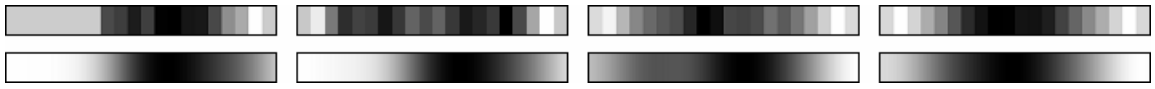

Figure 19: Learned value functions observed at the end of the 1st, 5th, 10th, and 30th trials of the 1D hot plate task, tested with discrete and continuous sensors. This agent used the same parameters as in the 1D hot plate learning performance plot.

1D Signaled Hot Plate

This task is identical to the normal 1D hot plate, except that there is only one safe zone. Furthermore, this safe zone is chosen randomly for each trial; it could be either on the left or right side. An input signal informs the robot on which side the safe zone is located. The agent has two sensors, including the discrete reward location signal sensor and its position sensor, and it has the same three actions as the normal 1D hot plate. The robot must learn to use this signal to get to the goal as quickly as possible. Figure 20 and Figure 21 show the resulting learning performance and value functions.

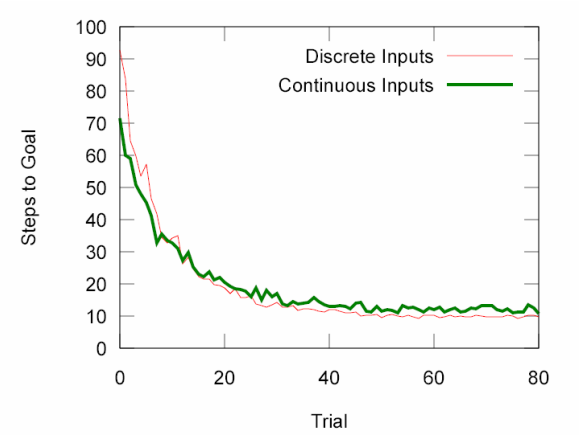

Figure 20: Learning performance on the 1D signaled hot plate task using the following parameters: policy learning multiplier = 10, signal input discretization = 2, position input discretization (discrete inputs plot only) = 20, continuous sensor resolution (continuous inputs plot only) = 25, number of runs averaged = 500.

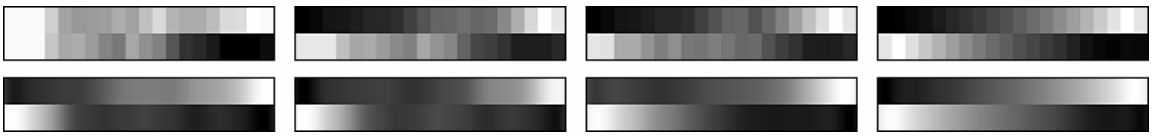

Figure 21: Learned value functions observed at the end of the 2nd, 5th, 10th, and 80th trials of the 1D signaled hot plate task, tested with discrete and continuous sensors. The two rows in each value function image correspond to the two signal states (one signaling a reward on the left, one signaling a reward on the right). This agent used the same parameters as in the 1D signaled hot plate learning performance plot.

2D Hot Plate

The 2D hot plate task is similar to the 1D hot plate task. This time the robot lives in a square 2D world. All four edges of this world are safe zones. The agent has two sensors that detect the robot's x and y position. It has five actions: move left, right, up, down, or do nothing. Figure 22 and Figure 23 show the learning performance and value functions.

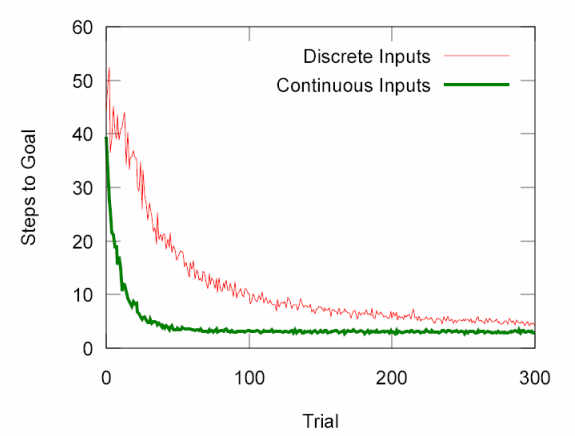

Figure 22: Learning performance on the 2D hot plate task using the following parameters: policy learning multiplier = 10, position input discretization (for discrete inputs plot only) = 20, continuous sensor resolution (for continuous inputs plot only) = 10, number of runs averaged = 300.

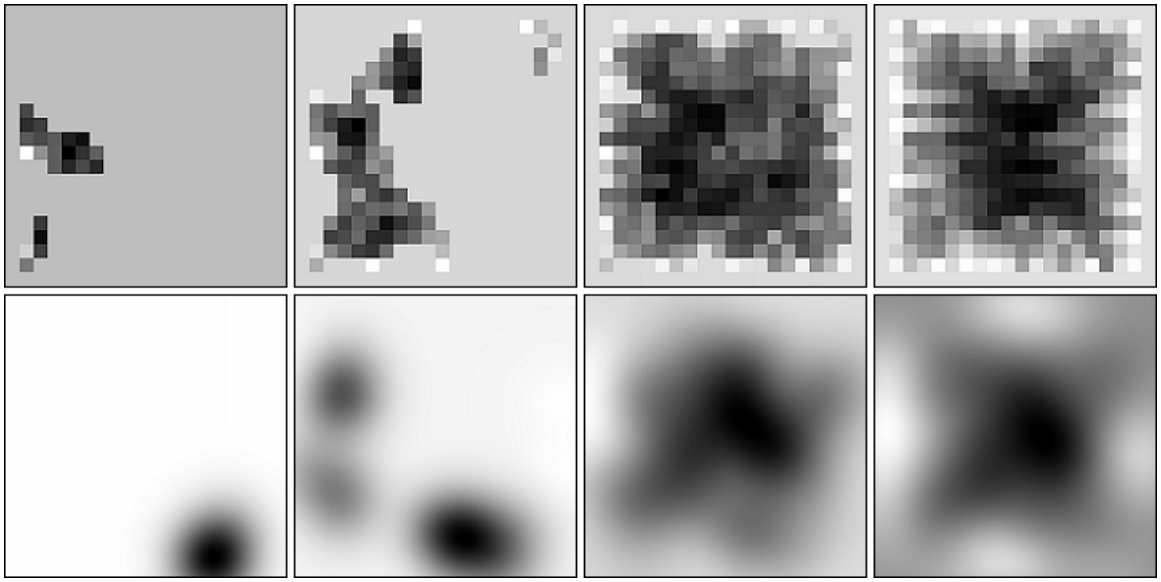

Figure 23: Learned value functions observed at the end of the 2nd, 10th, 50th, and 300th trials of the 2D hot plate task, tested with discrete and continuous sensors. This agent used the same parameters as in the 2D hot plate learning performance plot.

To gain an understanding of how the dynamic RBF allocation system works, we can visualize the RBF positions during learning on the 2D hot plate task. Figure 24 shows RBF

positions from an agent using a full (non-dynamic) RBF representation. The RBFs are equally spaced along each dimension.

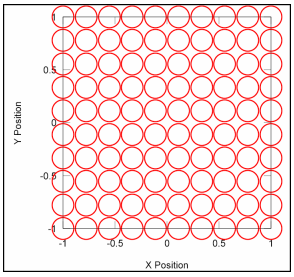

Figure 24: RBF positions from an agent using a full RBF state representation on the 2D hot plate task. The circle diameters here are meaningless; the actual RBFs overlap their neighbors significantly.

Figure 25 shows results from adding RBFs dynamically. As the agent explores different parts of the state space, it adds new RBFs. Eventually, all of the states that are actually experienced (in this case, the whole 2D grid) get covered with RBFs.

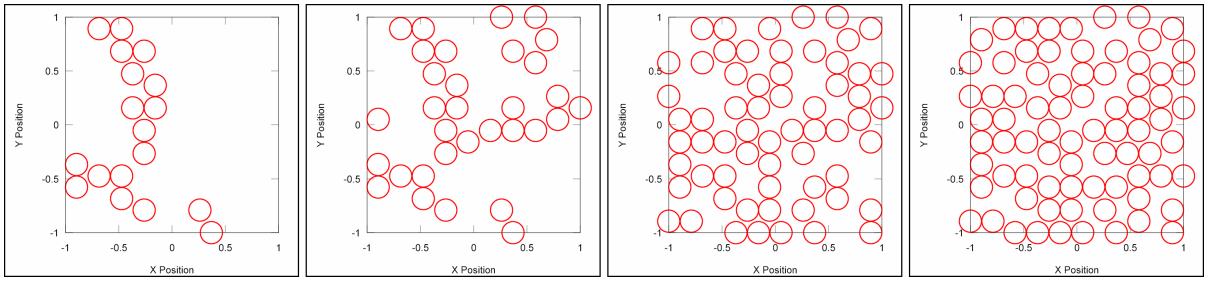

Figure 25: RBF positions observed at the end of the 5th, 10th, 50th, and 300th trials of the 2D hot plate task from an agent using a dynamically allocated RBF state representation. The circle diameters here are meaningless; the actual RBFs overlap their neighbors significantly.

2D Signaled Hot Plate

The 2D signaled hot plate task is a combination of the 1D signaled hot plate and the 2D hot plate. The robot lives in a square 2D world with a single safe zone at one of the edges. The safe zone is moved randomly at the start of each trial. A signal indicates where the safe zone is located. The agent's inputs are the robot's x and y position and discrete the reward location signal (which could be one of four possibilities). There are five outputs: move left,

right, up, down, or do nothing. The learning performance on this task is shown in Figure 26. Value function images are not shown here because they become more difficult to visualize when the state space is more than 2-dimensional.

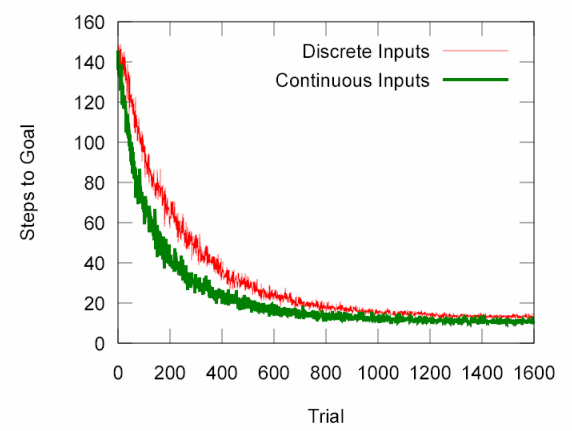

Figure 26: Learning performance on the 2D signaled hot plate task using the following parameters: policy learning multiplier = 10, signal input discretization = 4, position input discretization (discrete inputs plot only) = 20, continuous sensor resolution (continuous inputs plot only) = 25, number of runs averaged = 300.

2D Maze #1

This task tests an agent in a simple 2D maze environment (see Figure 27). There is a single start state and goal state which are always in the same locations. The agent receives -1 reward everywhere except the goal state where it receives a reward of 1. It can sense the robot's x and y position, and it can move left, right, up, down, or do nothing. This is essentially the same as the hot plate tasks except that the goal is in a single location, and the grid contains wall barriers. Figure 28 shows the agent's learning performance.

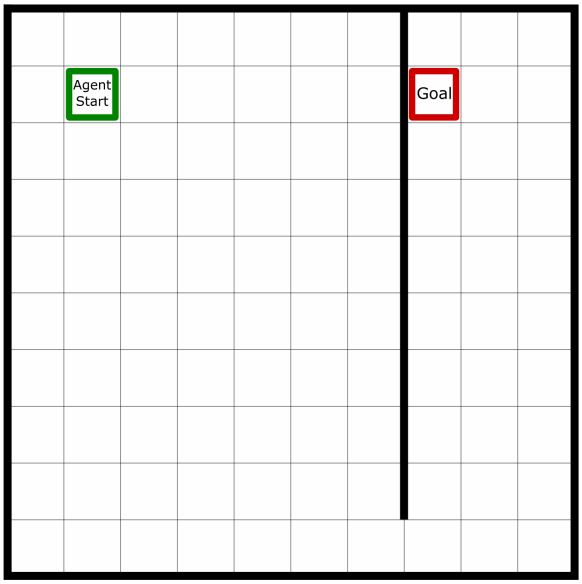

Figure 27: The layout of the environment in the 2D maze #1 task.

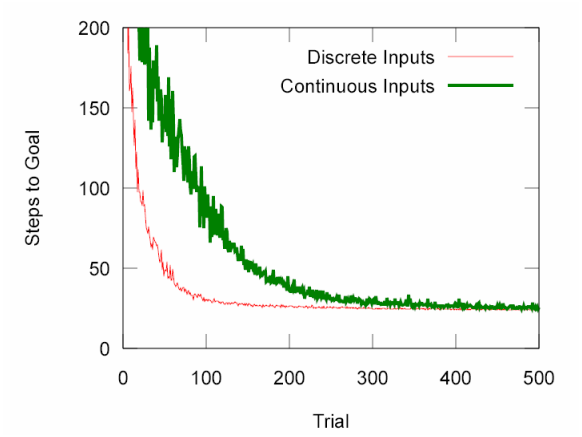

Figure 28: Learning performance on the 2D maze #1 task using the following parameters: policy learning multiplier = 1, position input discretization (for discrete inputs plot only) = 10, continuous sensor resolution (for continuous inputs plot only) = 15, number of runs averaged = 50.

Figure 29 shows the agent's learned value functions. In both the discrete and continuous cases it is easy to see the structure of the maze (especially the wall barrier) in the final value function images.

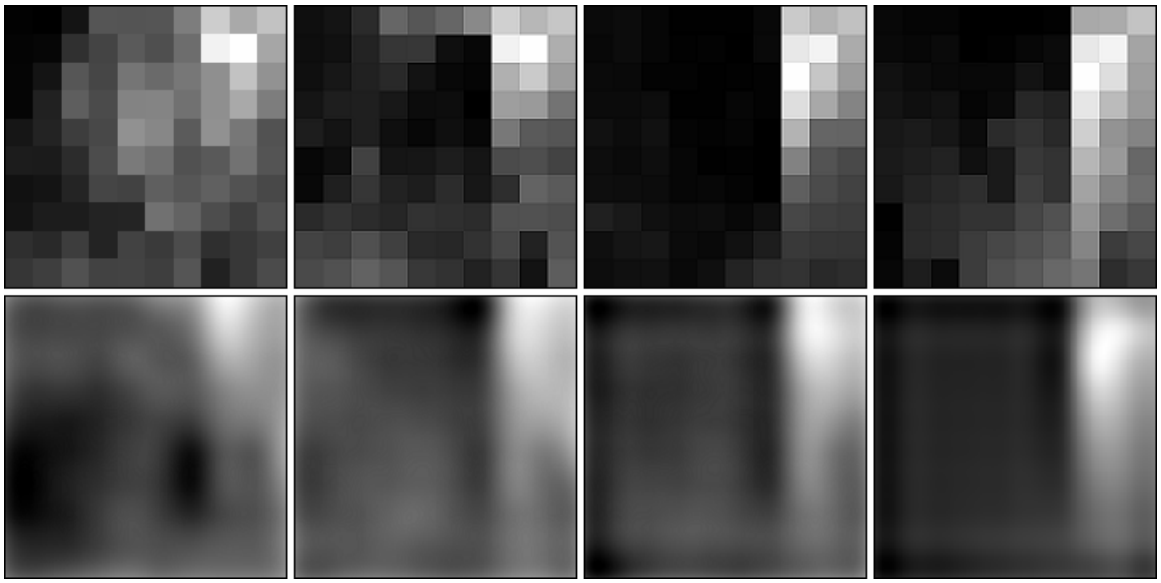

Figure 29: Learned value functions observed at the end of the 2nd, 5th, 10th, and 100th trials of the 2D maze #1 task, tested with discrete and continuous sensors. The following parameters were used: policy learning multiplier = 1, position input discretization (for the discrete inputs images on the top) = 10, continuous sensor resolution (for the continuous inputs images on the bottom) = 15.

2D Maze #2

This is another maze task with a more difficult layout (see Figure 30). The main idea we wish to demonstrate here is the tradeoff between high and low policy learning rates. Using a high policy learning rate changes the action selection probabilities faster which removes exploratory behavior early. This could be good or bad depending on the difficulty of the task.

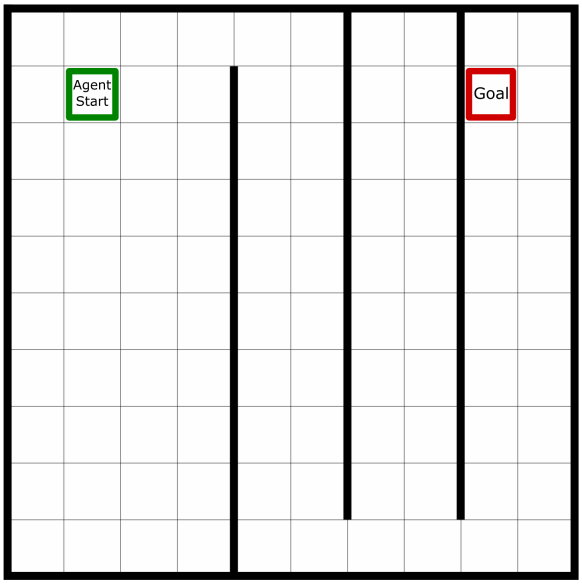

Figure 30: The layout of the environment in the 2D maze #2 task.

We test an agent in the maze with a high policy learning rate and again with a low policy learning rate. Figure 31 shows the resulting learning performance plot using a high policy learning rate. A good solution is found earlier and exploratory behavior is removed faster, leading to better learning performance for this task.

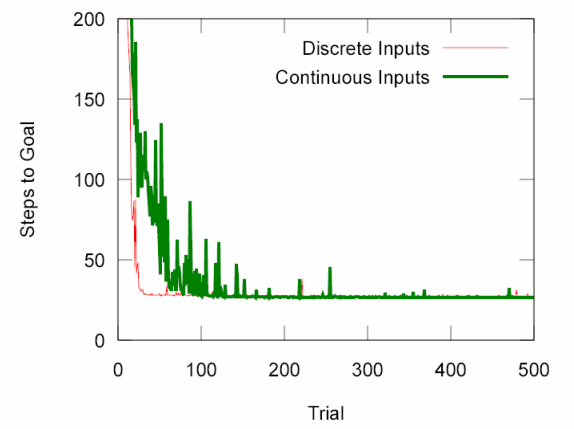

Figure 31: Learning performance on the 2D maze #2 task with a high policy learning rate. The the following parameters were used: policy learning multiplier = 5, position input discretization (for discrete inputs plot only) = 10, continuous sensor resolution (for continuous inputs plot only) = 15, number of runs averaged = 50.

Figure 32 show the learning performance when using a lower policy learning rate. Here, the agent uses exploratory actions longer, resulting in slower learning performance on this task.

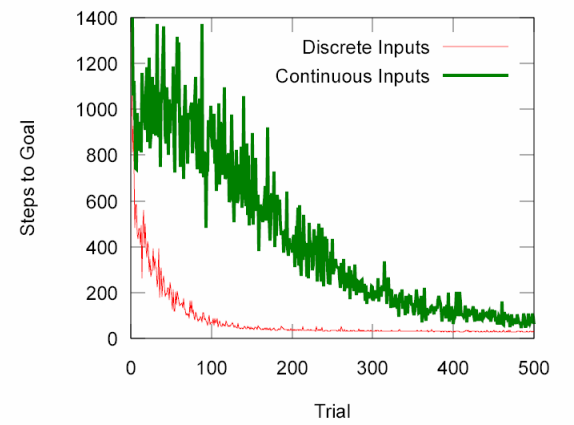

Figure 32: Learning performance on the 2D maze #2 task with a low policy learning rate. The the following parameters were used: policy learning multiplier = 0.5, position input discretization (for discrete inputs plot only) = 10, continuous sensor resolution (for continuous inputs plot only) = 15, number of runs average = 50.

Figure 33 compares the value functions of both agents at the same trial, displaying a more accurate value function when the agent is allowed to explore longer.

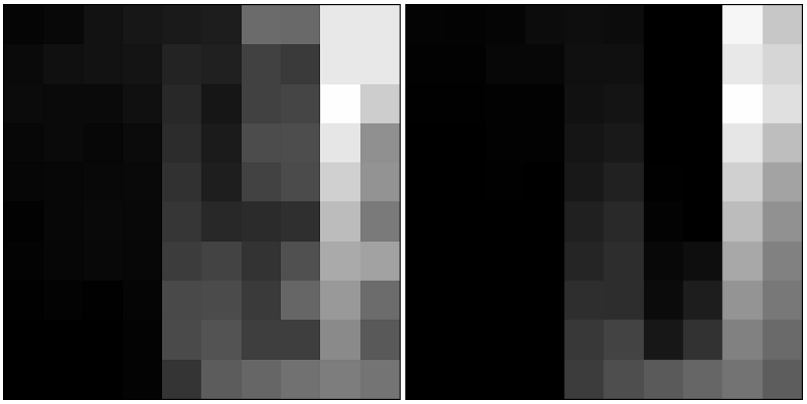

Figure 33: Value functions from the 100th trial of the 2D maze #2 task using discrete sensors. The image on the left is from an agent with a high policy learning multiplier (5). The image on the right is from an agent with a low policy learning multiplier (0.5). Learned value functions are more accurate when the agent is allowed to explore longer.

Physical Control Tasks

We now cover results from two physical control tasks, still using only the reinforcement learning component without planning. The agents here must learn to apply appropriate forces in order to control physically simulated systems. The core physics simulation software used here is Open Dynamics Engine (ODE http://www.ode.org). To simplify the process of constructing physically simulated environments, the author helped develop the Open Physics Abstraction Layer (OPAL http://opal.sourceforge.net). OPAL wraps ODE with a high-level interface and provides developers with intuitive objects (e.g. solids, joints, motors, and sensors) and XML serialization. Although the experiments in this section use very minimal physical environments, OPAL and ODE are powerful enough to manage complex worlds with expansive terrains, ground and air vehicles, legged robots, etc.

All experiments here were simulated with gravity set to 9.81 m/s^2. One of the more important parameters that must be set when running a physics simulation is the duration of each simulation step (i.e. the simulation step size). This should always be smaller than the Verve agent update step size. This is because the agent will expect the environment to have changed before each update. If the physics step size were larger than the agent's step size, the agent would choose an action, and its next observation would be identical to the previous one.

Pendulum Swing-Up

The pendulum swing-up task is one of the classic control problems used to test learning systems. The problem is that of getting a freely-swinging pendulum to hold itself upright and stay balanced (see Figure 34). The agent receives a reward of 1 when the pendulum is within 45 deg of vertical; otherwise, it receives a reward of -1. It has two continuous input sensors: the pendulum angle, and the pendulum angular velocity. It has three actions: apply a constant torque in one direction, apply a constant torque in the other direction, or do nothing. The pendulum is underactuated, so the agent must learn to swing it back and forth to build momentum in order to reach the top. It must stop applying force at just the right time (or apply an opposing force before reaching the top) to avoid overshooting the goal. Each trial

lasts 20 s. At the start of each new trial, the pendulum is given a random angle and angular velocity. This helps the agent experience more of the state space faster.

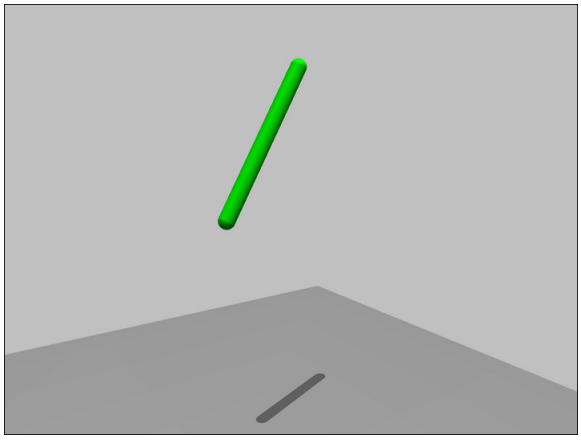

Figure 34: A physically simulated pendulum suspended in midair.

Figure 35 shows the agent's learning performance on the pendulum swing-up task. It reaches nearly optimal performance in about 60 trials.

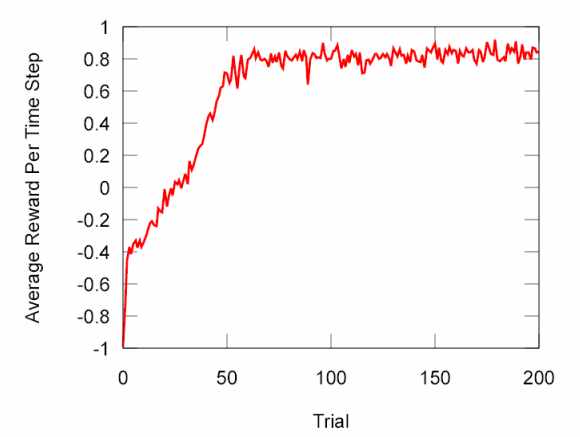

Figure 35: Learning performance on the pendulum swing-up task using the following parameters: physics step size = 0.01 s, agent step size = 0.1 s, pendulum mass = 1.0 kg, pendulum length = 1.0 m, ODE solver = "quickstep" (with 20 iterations per step), pendulum angle range = +/- 180 deg, pendulum angular velocity range = +/- 500 deg/s, pendulum torque range = +/- 2 N·m, continuous sensor resolution = 16, τ_{value}= 0.01 s, policy learning multiplier = 2, number of runs averaged = 10.

Figure 36 shows the (rather interesting) value functions learned over the course of a single run. Note that the agent's continuous sensor for the pendulum angle is circular because the angle can jump directly from -180 to 180 and vice versa. This means that the value function images would be more realistic if we wrapped them around a cylinder to join the ends of the input range.

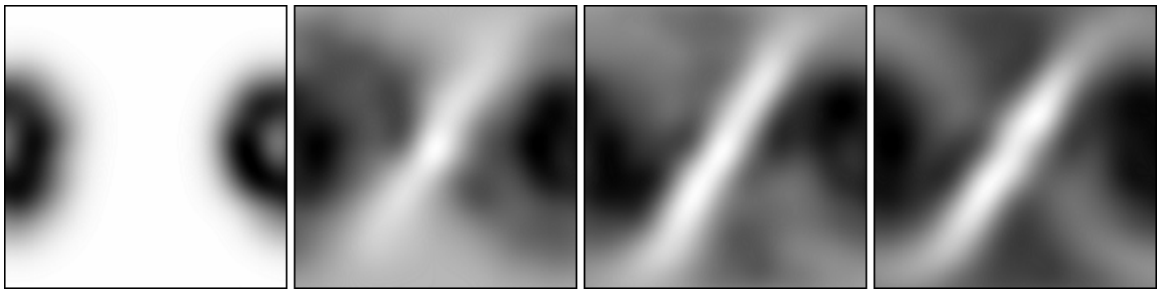

Figure 36: Learned value functions observed at the end of the 1st, 5th, 20th, and 100th trials of the pendulum swing-up task. The horizontal axis in each image is the pendulum's angle (i.e. the angle between the pendulum and vertical), whose range is +/- 180 deg and wraps directly from -180 to 180. The vertical axis is the pendulum's angular velocity in deg/s which ranges from -500 to 500 deg/s. This agent used the same parameters as in the pendulum learning performance plot.

Finally, Figure 37 shows the value function and policy neural networks before and after learning. The connection weights display distinguishable patterns that correspond to the actual state space. The center region of the connection weights contains mainly positive connections leading to the value function neuron, indicating a high value estimation for those states.

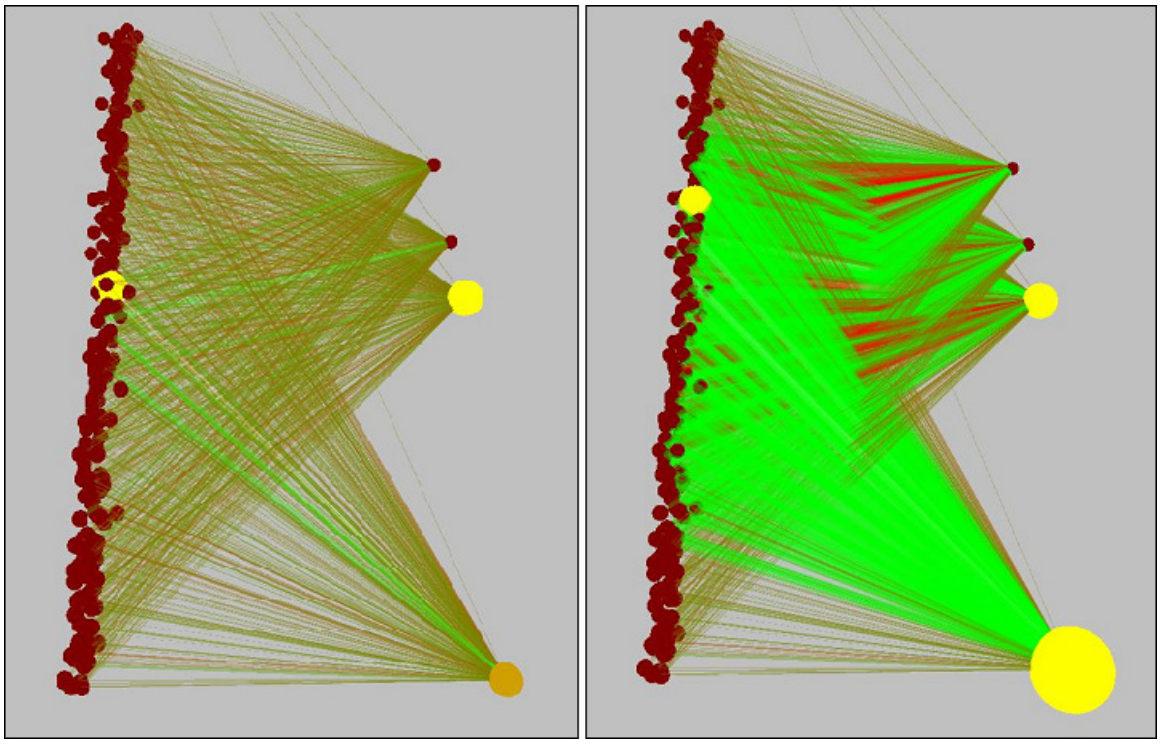

Figure 37: A visual representation of the pendulum agent's neural networks before (left) and after (right) learning. The neurons on the left side of each image are the state representation. The neurons on the top right represent the policy's three actions. The neuron on the bottom right represents the value function. Green connections are excitatory (positive); red connections are inhibitory (negative). Thicker connections have a larger weight magnitude.

Cart-Pole/Inverted Pendulum

This task, known as the cart-pole task or the inverted pendulum task, is another classic learning problem. The problem is that of learning to balance a pole attached to a cart by applying forces to the cart alone (see Figure 38). If the cart position is beyond one end of the track, or if the pole falls beyond some threshold angle, the agent is given a -1 reward; otherwise, it is given a reward of 1 on every step. It has four continuous input sensors: the cart position, the cart velocity, the pole angle, and the pole angular velocity. It has three actions: apply a constant force to the left left, apply a constant force to the right, or do nothing. A common goal for this task is to achieve a balancing time of 30 min (1800 s).

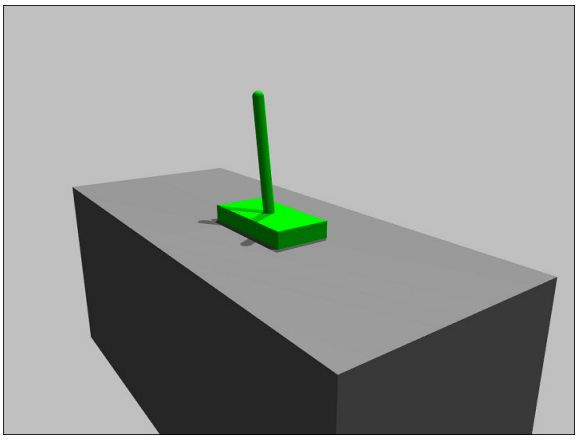

Figure 38: A physically simulated cart with an attached pole situated on a platform.

Figure 39 shows the learning performance of a single typical run. Once the important parts of the state space have been fully explored, failures become very sparse, leading to a roughly exponential increase in learning performance.

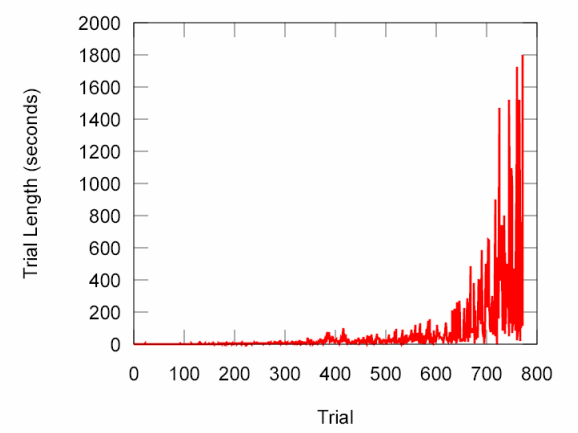

Figure 39: Typical learning performance for a single run of the cart-pole task using the following parameters: physics step size = 0.01 s, agent step size = 0.05 s, cart mass = 1.0 kg, pole mass = 0.1 kg, pole length = 1.0 m, coefficient of (static and kinetic) friction between cart and ground = 0, ODE solver = "quickstep" (with 20 iterations per step), cart x position range = +/- 2.4 m, cart x velocity range = +/- 2.4 m/s, pole angle range = +/- 12 deg, pole angular velocity range = +/- 100 deg/s, cart force range = +/- 10 N, continuous sensor resolution = 8, τ_{value} = 0.001 s, policy learning multiplier = 50.

Planning

We can see the effect of planning by testing agents on a 2D maze task with and without planning. Here we reuse the 2D maze #2 from before. The expected result is that the agents with planning will learn significantly faster than those without planning, even though the planning agents will do no reinforcement learning until their predictive models are accurate enough to attempt planning sequences. Figure 40 shows the resulting learning performance from this experiment. The learning performance is definitely faster with planning enabled.

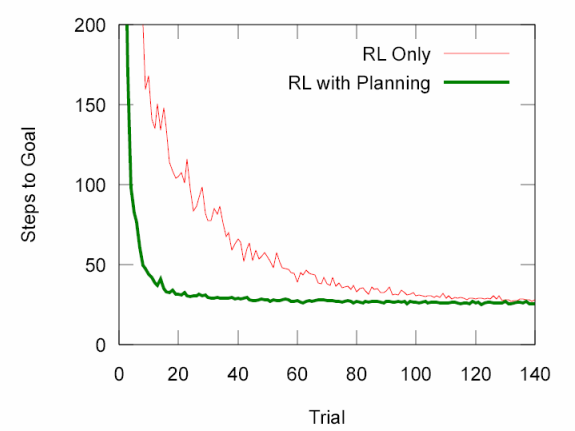

Figure 40: Comparison of learning performance on the 2D maze #2 task with and without planning enabled. This was performed with discrete inputs only. The following parameters were used: τ_{pred} = 0.0 s, policy learning multiplier = 1, position input discretization = 10, maximum planning sequence length = 100, number of runs averaged = 50.

Figure 41 shows the actual mean squared error for observation and reward predictions, averaged over every step for each trial. For this task it does not take long for the average MSE to approach zero.

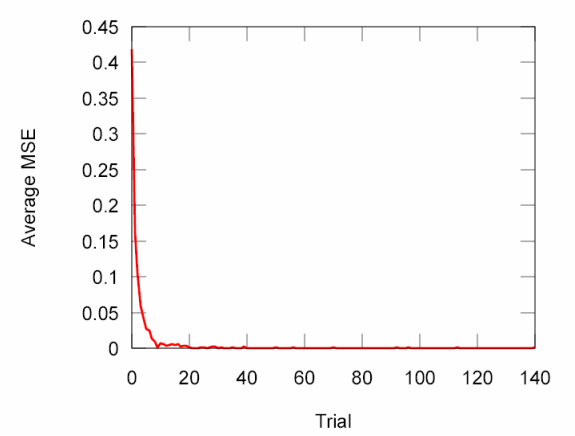

Figure 41: The predictive model's mean squared error for each trial of the 2D maze #2 task, averaged over every time step. This was performed with discrete inputs only. The following parameters were used: τ_{pred} = 0.0 s, policy learning multiplier = 1, position input discretization = 10, maximum planning sequence length = 100, number of runs averaged = 50.

Figure 42 shows the number of planning steps taken for each planning sequence. It appears that the average planning sequence length is near 25 steps for this task.

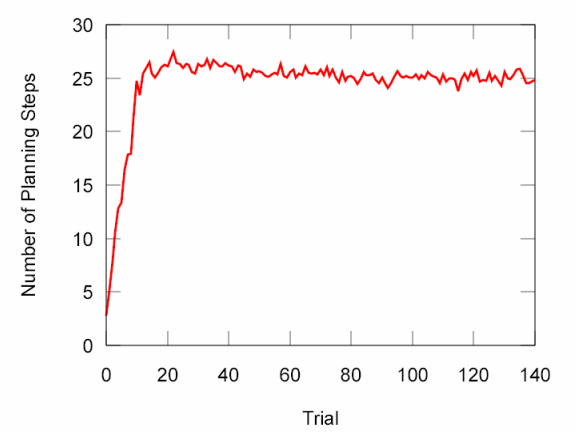

Figure 42: The average number of planning steps taken per time step for each trial of the 2D maze #2 task. This was performed with discrete inputs only. The following parameters were used: τ_{pred} = 0.0 s, policy learning multiplier = 1, position input discretization = 10, maximum planning sequence length = 100, number of runs averaged = 50.

Planning with Curiosity

Now we demonstrate the usefulness of curiosity by performing a simple task in a new 2D grid environment (shown in Figure 43). The environment contains two rewards, one larger than the other. The smaller reward is closer to the agent's initial position. The two reward positions and the agent's initial position are always the same. Performance is measured by summing the rewards received at each time step. Each trial lasted exactly 100 time steps. The intended result is that normal planning agents will quickly find the small reward but not spend enough time exploring to find the larger reward. The curious agents, on the other hand, should be more likely to find the larger reward and attain a larger reward sum.

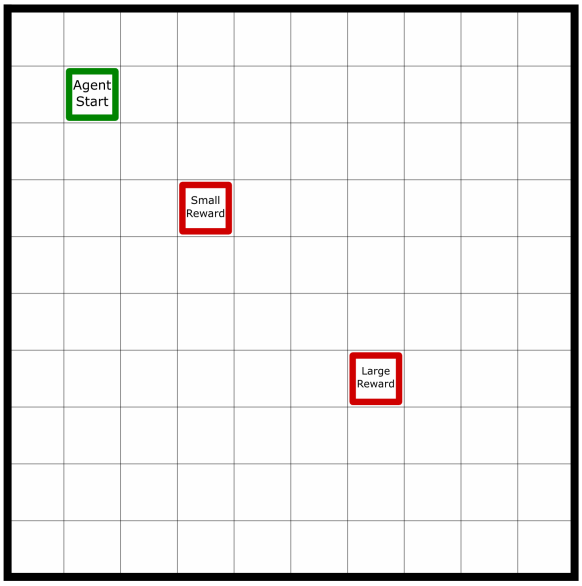

Figure 43: A 2D grid environment used to test curiosity. The main features are the agent's initial position, a small reward (0.1), and a large reward (1.0). All empty locations yield a reward of -1.0.

Figure 44 compares the learning performance of planning agents with and without curiosity. The curious agents definitely achieve a higher reward sum.

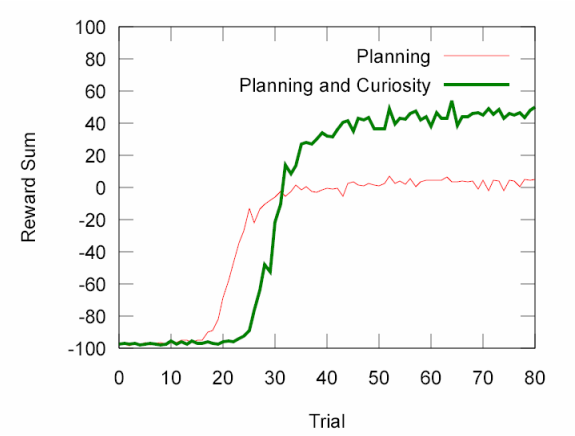

Figure 44: Learning performance on the 2D grid curiosity task with and without curiosity enabled. This was performed with discrete inputs only. The following parameters were used: τ_{pred} = 0.0 s, policy learning multiplier = 0.1, position input discretization = 10, maximum planning sequence length = 50, number of runs averaged = 30.

Figure 45 compares the agents' value functions. The non-curious agent considers the area near the small reward more valueable because it has not explored far enough to find the large reward. The curious agent's value function is much more beneficial for acquiring a larger reward sum.

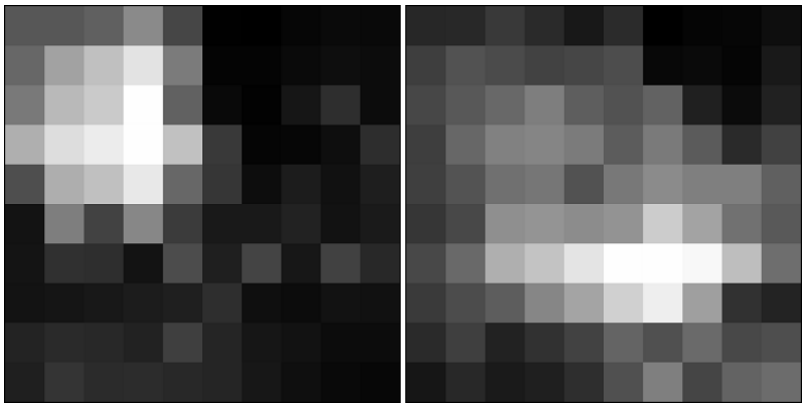

Figure 45: Value functions from the 80th trial of the 2D grid curiosity task using discrete sensors. The left and right images are from an agent with planning alone and an agent with planning and curiosity, respectively. The agent starts each trial near the upper left corner. Curious agents were able to find a much larger reward farther from the starting point.

Discussion

In this section we review some of the major results from this chapter. We discuss some of the tradeoffs involved and a few hypotheses.

The discrete environment tasks without planning gave expected results. In each case more inputs always yielded longer learning times because of the extra time required to explore the state space. The number of trials required to achieve near optimal performance seemed to scale linearly with the number of states in the state space. If this is indeed true, we can conclude that the reinforcement learning component alone is not sufficient to solve tasks with much larger state spaces in a practical amount of time.

On all four hot plate tasks the agents with continuous sensors performed as well or better than those with discrete sensors. This may be because the RBFs provide the advantage of generalization. Agents with continuous sensors already have an approximate idea of what nearby values will be, whereas agents with discrete sensors must learn each state value from scratch. However, the agents with continuous sensors did significantly worse on the 2D maze tasks. This could be because it is more difficult to approximate the maze value functions with RBFs. It might help to increase the continuous sensor resolution, thus providing more RBFs for approximation. Practically speaking, for the discrete environment tasks the situation is somewhat artificial when using continuous sensors because we are using RBFs to approximate discontinuous states.

The physical control tasks gave promising results. The agents were able to solve each problem in a reasonable number of trials even without planning enabled. An obvious next step would be to attempt these tasks with planning.

The results on the 2D maze #2 task showed the effect of changing the policy learning rate. High value policy learning rates quickly polarize the action selection probabilities, leading to a greedy selection scheme. This can sometimes be good because it can yield near-optimal learning performance. However, it also greatly diminishes exploration. This is probably

only beneficial on simple tasks. More difficult tasks would likely produce more unreliable the results (the agent might find a good greedy solution soon early; otherwise, it might never find a good solution since it loses exploratory actions so quickly). Figure 31, Figure 32, and Figure 33 show the effects of this tradeoff. The learning performance is much better with a high policy learning rate, but the state space is barely explored (i.e. the learned value function is not very accurate).

The results with planning showed that it is indeed beneficial. The effect might be even more pronounced for more complex tasks, especially those that cannot be solved easily without it. It is interesting that the average number of planning steps per planning sequence never approached the maximum length of 100. Since the predictive model's mean squared error quickly approaches zero, why would the agent ever have to end a planning sequence early? It is probable that the agent never explores (and never has a drive to explore) much of the state space. The predictive model's MSE stays close to zero because the agent ignores the unexplored states during actual interactions. All planning trajectories end as soon as they reach the unexplored territory, with an average trajectory length around 25 in this case. If we performed this same experiment with curiosity enabled, the agent would be driven to explore the whole state space, so we would expect the average trajectory length to be higher.

One tradeoff is related to the planning uncertainty threshold. A higher threshold means that the agent is willing to perform longer planning trajectories through uncertain territory. Using a lower threshold forces the agent to wait until its predictive model is more accurate for a given state before attempting to plan at that state, confining planning trajectories to familiar territory. Another tradeoff concerns the maximum number of planning steps allowed. Short planning sequences are naturally less computationally expensive, but they might yield slower learning performance. Longer planning sequences can yield better learning performance at first, but they can waste time by planning through states that are already well-known.

It seems that once the predictive model is fairly accurate (yielding long planning trajectories), a lot of time is wasted planning in states that are already predictable. This leads us to one of

the more interesting avenues of future research. We could design an agent that encapsulates well-learned action sequences of into higher-level modules (i.e. “motor programs”). This idea is discussed in more detail in the next chapter.

Adding curiosity rewards slowed learning progress slightly, but it yielded a superior final result. This was due to the agent exploring more of its environment and finding a better reward. The intrinsic drive to experience new states forced the agent to improve its predictive model which was then used to train the value function and policy. A tradeoff with curiosity concerns how much the curiosity rewards are scaled compared to actual external rewards. The issue is similar to that of adjusting exploration by scaling the policy learning rate. In the case of the policy we are simple adjusting the amount of randomness that is allowed to affect action selection. In the case of curiosity, we are adjusting the agent’s drive to seek new situations. With curiosity rewards it is even possible for agents to make long-term plans to find interesting situations. We could say that curiosity is a more advanced form of exploration than simple randomness in the action selection.

One last tradeoff concerns efficient use of computational resources. (This is purely hypothetical at this point, but it should be easy to test.) Is it more efficient to allow planning (resulting in fewer, more computationally intensive trials) or to disable planning and let the agent learn directly from the environment (resulting in many trials that can be processed faster)? It would probably be better to disable planning only in simple environments where it is cheaper to process the environment than it is to process the agent’s mental model of the environment. Another factor is whether the agent is controlling a simulated body or an actual physical robot. For an actual robot it would be beneficial to use planning because it can be dangerous and expensive to perform trials in the real world.

6 Conclusions

This thesis began by discussing the key issues involved in designing general purpose reinforcement learning agents. The result of this discussion was an agent design that combines temporal difference learning, a radial basis function state representation, planning from a learned predictive model, uncertainty estimations to determine the length of planning sequences, and curiosity rewards proportional to uncertainty. Several neuroscientific research papers were then reviewed that provide connections between theoretical reinforcement learning and biological brains. After that we introduced the Open Source library Verve, a C++ implementation of the agent designed here. Finally, we demonstrated results from several experiments that tested basic reinforcement learning, planning, and curiosity. From these initial results, we can conclude that: 1) the agent implementation effectively solved all of the problems presented, 2) planning does indeed improve learning performance, and 3) curiosity can help the agent improve its predictive model.

This chapter briefly reviews the main contributions of this thesis. It then discusses several intriguing avenues of future work that could improve the agent design.

Contributions

In summary, the main contributions of this thesis are:

- a general purpose agent design that combines temporal difference learning, radial basis functions, planning, uncertainty, and curiosity
- a discussion of correlations between theoretical reinforcement learning and reward processing in biological brains
- a concrete Open Source implementation of the agent design
- experimental results showing learning performance on several tasks

It is hoped that these contributions will help spread reinforcement learning research to new audiences and add value to the field in general.

Future Work

So far we have described and implemented an agent with many powerful features. However, there are still several ways in which our agent design is limited. These include the lack of precise temporal predictions, limited scalability to large state and action spaces, and the inability to learn continuous motor signals from scratch. This section outlines future work to be done, starting with a list of experiments and finishing with ideas for more advanced agent designs.

Experiments

An immediate next step is to attempt more difficult learning tasks to test the current agent's limits. In the discrete environment category there are still many options available. We could try more difficult 2D mazes, predator/prey simulations, card games, board games, etc. The current implementation is limited when it comes to physical control tasks, the main reason being that it simply selects actions but cannot generate its own continuous control signals. Once this problem is alleviated (as described below), we could test the agents on more complex simulated robotics problems, like making legged robots walk or training a robotic arm to throw a ball.

It would be especially interesting to setup open-ended learning situations where agents explore using curiosity rewards alone. Inspired by Singh, Barto, & Chentanez (2005), we could put a physically simulated robot in a "playground" environment with several interesting objects and observe it playing. Another idea is a sort of interactive dog training application where users may reward an agent for anything they wish. They would focus the learning process by giving small rewards along the path to the intended behavior[1].

---

[1] The idea of applying such "shaping rewards" in general (either interactively or algorithmically) could be detrimental if the rewards inadvertently lead the agent into a locally optimal solution.

One final experiment idea is to train an agent to perform robot behaviors in simulation, and then transfer the experienced agent to a real robot. This process is described in detail in Nolfi & Floreano (2000). It appears to work very well as long as realistic sensor and actuator noise is included in the simulation.

Temporal State Representation

The current agents have no temporal state representation, so they cannot predict future events at specific times. For example, imagine one of the agents in the following classical conditioning situation. A bell rings consistently 2 s before a reward. If the bell rings, and the agent remains in the same physical state for the next 2 s, it will not be able to predict when the reward will occur. Its predictions are dependent upon changing observations from the external environment. What we need is an augmented concept of a "state" that includes not only the current observation, but an observation at a specific time relative to the present. Thus, a "state" is no longer solely dependent upon external events.

This new state representation could be implemented with more RBFs. In this case, each time step in the recent past needs a separate copy of the state representation being used here, i.e. we would have a "time t" state representation, a "time t - 10 ms" state representation, a "time t – 20 ms" state representation, etc. Input signals simply move from one RBF array to the next at each step like a scrolling marquee[1]. This new system would work naturally with the existing implementation.

Hierarchical Structures

In general, organizing states and policies hierarchically would help scale the learning process to larger state and action spaces. It would allow the agent to focus resources on higher levels of abstraction without wasting time on well-learned details. This section discusses ideas for implementing hierarchical policies that are learned from atomic behaviors. Many of the

[1] This idea is very similar to the coincidence detectors in the auditory system that localize sounds based on interaural time differences.

ideas here (especially the definition of a "motor program") stem from the theory of "options" (Sutton, Precup, & Singh, 1999).

Let us expand our definition of a policy into a new construct called a "motor program." A motor program is a mapping from a state to an action, and it has an initial state that triggers it and a goal state that stops it. We could construct hierarchies of these motor programs. A high level policy proceeds by sequentially executing its child policies. Thus, high level policies are temporally longer than low level ones. At the very bottom of the policy hierarchy is a set of inborn, hard-coded, atomic behaviors.

This system would enhance exploration by allowing it to act on many levels. Exploratory actions at high levels (e.g. performing a grasping behavior), composed of many low-level actions (e.g. individual finger muscle movements), would be necessary to solve high level tasks. For an agent to solve the task of grasping an empty can with a robotic arm and dropping it into a recycle bin, it would have to explore using actions at the level of grasping and releasing, not at the level of individual finger movements.

Planning would also be much more effective in such a hierarchical system. Instead of performing a planning trajectory through every low-level action in a sequence, the agent could treat a high level motor program as a single planning step. For example, instead of planning through every step of a grasping sequence, the agent would treat "grasp" as a distinct action. This would vastly improve the efficiency of the planning system. Say we have an agent that is allowed to take 20 planning steps per update. At first it would spend all 20 steps focused on low-level behavior. Once it learns several motor programs, however, it could spend each planning step on a high level motor program.

How would agents learn new motor programs from scratch? First we must imagine an agent that has learned an accurate model of its environment. Once the agent can perform long sequences with low uncertainty, it becomes pointless to repeatedly cycle through the same sequences. Instead, such a well-learned sequence should be encapsulated into a motor

program. The extent of this "well-learned" sequence would be defined as the start and end of a series of steps observed to give repeatable results over time (e.g. utilizing the usual uncertainty estimations). The predictive model could then skip over the low-level actions when planning and simply jump from the start state to the end state. The process of motor program learning might continually work on the highest level of programs yet constructed, occasionally refining lower-level programs. It would mainly focus on the levels of the hierarchy where unexpected events occur[1]. (When something unexpected happens, the agent might need to destroy certain parts of the hierarchy and relearn them.)

A hierarchy of motor programs might require a hierarchical state representation. Instead of having all sensory inputs converge onto a single RBF array, we could use a hierarchy of RBF arrays. The higher levels combine low level elements to represent more abstract concepts. At the very lowest level are the inborn, hard-coded sensors that gather information directly from the environment. Each of these arrays might be connected to every array in the level below and the level above. However, this is probably not necessary. In many natural sensory systems (e.g. vision), the lower parts of the hierarchy (i.e. V1) do not need to connect to the lowest levels of other sensory systems (e.g. the somatosensory cortex). Only at the higher levels should the various major sensory input hierarchies be combined into a single hierarchy[2]. Still, how do we know which sensory input hierarchies to combine at which levels? We could start with all the higher levels fully connected and gradually remove connections that never get used.

Learning Continuous Motor Outputs

In this thesis we have simplified our discussion of motor outputs by focusing only on action selection. We have assumed that there already exists a set of actions from which to choose. This is the standard approach in most reinforcement learning work. To make reinforcement

---

[1] The idea of focusing on certain levels of the hierarchy seems to correspond to a model of attention.

[2] The issue of combining various sensory inputs into cohesive, high-level concepts is called the "binding problem."

learning agents more general (especially for physical control tasks), we need to let the agent learn these actions from scratch and choose from among them later.

To accomplish this we simply use the idea of hierarchical policies from the previous section. At the lowest level is a set of atomic policies[1] that must be hand-designed. These could include things like PD/PID controllers, time-dependent oscillatory signals, etc. The agent would start by exploring the effects of these primitive policies, eventually building more complex control signals from various combinations.

[1] It is likely that the lowest-level atomic actions in vertebrates might be encoded in the spinal cord. Furthermore, these "actions" might actually be feedback loops that self-adjust without any higher commands from the brain.

References

Cannon, C.M. & Bseikri, M.R. (2004). Is Dopamine Required for Natural Reward? *Physiology and Behavior*, 81(5):741-748.

Fiorillo, C.D., Tobler, P.N. & Schultz, W. (2003). Discrete Coding of Reward Probability and Uncertainty by Dopamine Neurons. *Science*, 299:1898-1902.

Flanagan, J.R., Vetter, P., Johansson, R.S. & Wolpert, D.M. (2003). Prediction Precedes Control in Motor Learning. *Current Biology*, 13:146-150.

Grzeszczuk, R., Terzopoulos, D. & Hinton, G. (1998). NeuroAnimator: Fast Neural Network Emulation and Control of Physics-Based Models. In *SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques,* pp. 9-20.

Jordan, M.I. (1996). Computational Aspects of Motor Control and Motor Learning. In Heuer, H. & Keele, S. (Eds.), *Handbook of Perception and Action: Motor Skills,* pp. 71-118. New York, NY: Academic Press.

Montague, P.R., Hyman, S.E. & Cohen, J.D. (2004). Computational Roles for Dopamine in Behavioral Control. *Nature*, 431:760-767.

The NeuroEvolution of Augmenting Topologies (NEAT) Users Page. http://www.cs.utexas.edu/users/kstanley (accessed 11-15-05).

Nolfi, S. & Floreano, D. (2000). *Evolutionary Robotics*. MIT Press, Cambridge, MA.

Open Dynamics Engine project website. http://www.ode.org (accessed 11-15-05).

Open Physics Abstraction Layer project website. http://opal.sourceforge.net (accessed 11-15-05).

PIQLE: a Platform Implementing Q-LEarning algorithms in JAVA project website. http://www.lifl.fr/~decomite/piqle/index.html (accessed 10-12-05).

Rao, R.P.N. & Ballard, D.H. (1999). Predictive Coding in the Visual Cortex: A Functional Interpretation of Some Extra-Classical Receptive-Field Effects. *Nature Neuroscience*, 2(1):79-87.

Reil, T. & Husbands, P. (2002). Evolution of Central Pattern Generators for Bipedal Walking in a Real-Time Physics Environment. *IEEE Transactions on Evolutionary Computation*, 6(2):159-168.

Reinforcement Learning Toolbox project website. http://www.igi.tugraz.at/ril-toolbox/general/overview.html (accessed 10-12-05).

RL Toolkit project website. http://rlai.cs.ualberta.ca/RLAI/RLtoolkit/RLtoolkit1.0.html (accessed 10-12-05).

SALSA (System using Artificial Life to Study Adaptation) project website. http://www.cs.indiana.edu/~gasser/Salsa (accessed 10-12-05).

Schmidhuber, J. (2005). Self-Motivated Development Through Rewards for Predictor Errors/Improvements. In *Developmental Robotics 2005 AAAI Spring Symposium,* pp. 1-3.

Schultz, W. (2000). Multiple Reward Signals in the Brain. *Nature Reviews Neuroscience*, 1:199-207.

Schultz, W. & Dickinson, A. (2000). Neuronal Coding of Prediction Errors. *Annual Review of Neuroscience*, 23:473-500.

Singh, S., Barto, A.G. & Chentanez, N. (2005). Intrinsically Motivated Reinforcement Learning. In Saul, L.K., Weiss, Y. & Bottou, L. (Eds.), *Advances in Neural Information Processing Systems 17,* pp. 1281-1288. Cambridge, MA: MIT Press.

Stanley, K.O. & Miikkulainen, R. (2002). Efficient Reinforcement Learning through Evolving Neural Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference,* pp. 569-577.

Suri, R.E. & Schultz, W. (1998). Learning of Sequential Movements by Neural Network Model with Dopamine-Like Reinforcement Signal. *Experimental Brain Research*, 121(3):350-354.

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

Sutton, R.S., Precup, D. & Singh, S. (1999). Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, 112:181-211.

Thorndike, E.L. (1911). *Animal Intelligence*. Hafner, Darien, CT.

Verve project website. http://verve-agents.sourceforge.net (accessed 11-15-05).