Proceedings of IDETC/CIE 2006
ASME 2006 International Design Engineering Technical Conferences
& Computers and Information in Engineering Conference
September 10-13, 2006, Philadelphia, Pennsylvania, USA

DETC2006-99651

VERVE: A GENERAL PURPOSE OPEN SOURCE REINFORCEMENT LEARNING TOOLKIT

Tyler Streeter
VR Applications Center
Iowa State University
1620 Howe Hall
Ames, IA 50011
tylerstreeter@gmail.com

James Oliver
VR Applications Center
Iowa State University
1620 Howe Hall
Ames, IA 50011
oliver@iastate.edu

Adrian Sannier
School of Informatics
Arizona State University
PO Box 877705
Tempe, AZ 85287
adrian.sannier@asu.edu

ABSTRACT

Intelligent agents are becoming increasingly important in our society in applications as diverse as house cleaning robots, computer-controlled opponents in video games, unmanned aerial combat vehicles, entertainment robots, and autonomous explorers in outer space. However, the broader adoption of intelligent agents is often hindered by their limited adaptability to new tasks; when conditions change slightly, agents may quickly become confused. Additionally, a substantial engineering effort is required to design an agent for each new task. This paper presents an adaptable, general purpose intelligent agent toolkit based on reinforcement learning (RL), an approach with strong mathematical foundations and intriguing biological implications. RL algorithms are powerful because of their generality: agents simply receive a scalar reward value representing success or failure, which greatly simplifies the agent design process. Furthermore, these algorithms can be combined with other techniques (e.g., planning from a learned internal model) to improve learning efficiency. The design and implementation of an open source RL toolkit is presented here as a step towards the goal of general purpose agents. Experimental results show learning performance on several tasks, including two physical control problems.

1 INTRODUCTION

Intelligent agents 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.

Reinforcement learning (RL) is the problem of choosing the optimal action in a given situation in order to maximize future rewards [1]. Figure 1 shows the general situation with a set of abstract components representing an agent and its environment. The agent performs a context-dependent 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.

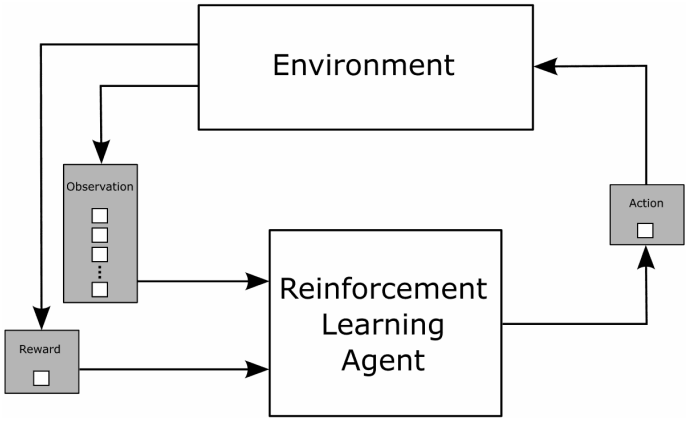

Figure 1: A typical RL setup, including an environment that provides observations and rewards and an agent that responds with actions.

RL 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 an RL problem as long as the goal can be represented as a scalar reward value. Thus, algorithms designed to solve RL problems are applicable to a wide variety of tasks.

When trying to solve RL 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 those that have been most successful in the past.

Fortunately, a set of powerful RL 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 paper presents a summary of some of the algorithms available for solving RL 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 [2] designed to give application developers a useful tool for creating intelligent learning agents. This tool can be applied to a variety of problems, ranging from simple simulated agents in discrete grid worlds to real robots acting in the physical world. The implementation and results are discussed more fully in [3].

There are existing RL software tools [4, 5, 6] available that provide general frameworks for experimenting with different algorithms. Other tools [7] are useful for teaching RL concepts without having to write software. However, 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 RL algorithms in more diverse applications.

The next section discusses reinforcement learning in more detail. It covers some of the specific challenges involved and presents a general purpose solution. Section 3 introduces the Verve software library, an implementation of the ideas discussed in section 2. Section 4 presents a series of experiments that test Verve's effectiveness. Section 5 concludes the paper with a summary and a list of contributions.

2 RL CHALLENGES AND SOLUTIONS

RL problems present 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 RL 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 paper. For more information, see [1].

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.

Two of the main internal components of most RL agents are: 1) a *value function* which maps states to value estimations, and 2) a *policy* which maps states to actions [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 part of a "critic" system that reinforces the actor using prediction errors.

2.1 Temporal Credit Assignment

While an agent is interacting with its environment, rewards are usually 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.

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 an RL task is to learn a "value function." A value function (more specifically, a state value function) is a mapping of states to values. Given some state, the value function returns the (usually estimated) value associated with being in that state. (Another type of value function is an "action value function" that represents the value of taking an action in a specific state. Here, we will only focus on state value functions.) The value function transforms the discontinuous primary reward into a continuous internal signal. 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.

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). See [1] for a more complete coverage of both methods.

A fairly new method which has some of the benefits of dynamic programming and Monte Carlo methods is called temporal difference learning [1]. It does not require an environment model (though it can still benefit from such a model), 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:

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

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. The following reflects this change:

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

We can simplify these ideas to get the following form:

$$V(s_t) = r_t + \gamma V(s_{t+1}) \tag{2-3}$$

In other words…

$$0 = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2-4}$$

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:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \quad (2\text{-}5)$$

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:

$$V(s_t) \leftarrow V(s_t) + \eta_{value}\, \delta_t \quad (2\text{-}6)$$

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.

2.2 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 [1]. They increase when the corresponding value estimation or action is used, and they decrease exponentially over time. 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.

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:

$$e(s_t) \leftarrow \gamma\lambda e(s_t) \quad (2\text{-}7)$$

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:

$$V(s_t) \leftarrow V(s_t) + \eta_{value}\, \delta_t e(s_t) \quad (2\text{-}8)$$

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.

2.3 Exploration vs. Exploitation

The exploration vs. exploitation dilemma is the problem of deciding when to use previous knowledge to guide actions and when to take exploratory actions, with the hope of finding something better. 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 [1]. 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 [1]. 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.

2.4 Large, Continuous State Spaces

The methods we have covered up to this point solve RL 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. The convergence proofs do have a few other requirements, such as using a learning rate that decreases over time (here we use a constant learning rate and do not worry about achieving the exact optimal solution) and other assumptions that do not affect the discussion in this paper. Convergence is still questionable with nonlinear function approximation, such as backpropagation with multilayer neural networks. 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.

Now the basic problem is to represent the value function and policy as linear combinations of the states. The problem with doing this in general is that linear representations, such as single layer neural networks, are fundamentally limited in what they can represent. Specifically, they can only represent problems that are linearly separable. This excludes certain problems (like the classic XOR problem). See [8] for a more detailed description of neural networks and linear separability.

One way to get around these limitations and still use methods like linear neural networks is to augment the state representation. Instead of learning a linear combination of the sensory inputs directly, it is better to generate a set of more complex features that represent various combinations of the sensory inputs. For example, if all inputs are discrete values, we could simply enumerate all possible combinations of the inputs, resulting in an exhaustive list of states.

In most cases we will have continuous input values (e.g. readings from thermometers, accelerometers, cameras, etc.). To form a set of features in continuous state spaces, we can use radial basis functions (RBFs) [8], which allow localized learning and some generalization. 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:

$$a = e^{\left(\frac{-\|i-c\|^2}{2\sigma^2}\right)} \tag{2-9}$$

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 2. A single continuous value, even in 1-dimensional space, can be represented with an array of RBFs. 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.

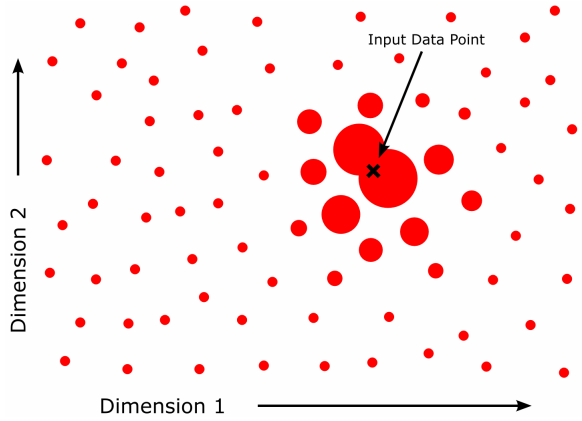

Figure 2: An array of radial basis functions in 2-dimensional space representing an input data point. Each RBF here is a separate circle with a diameter proportional to the RBF's activation level. (The more distant RBFs would actually have a near-zero diameter.)

When representing a continuous value, only a few RBFs near the value are active. This allows localized learning, which is important for learning function approximations without catastrophic interference (i.e. changing a single parameter does not affect the entire approximation).

In the general case we can create an exhaustive 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, 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. This may be what occurs in biological brains: we start out with many more connections than we need, and we lose connections that rarely get used.

Figure 3 shows our agent design. It uses an RBF state representation with linear neural networks for the value function and policy, which are trained by the TD error signal.

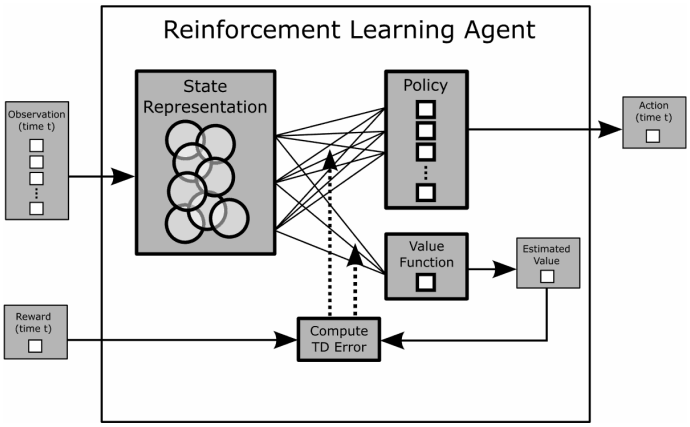

Figure 3: An agent that processes incoming observations into an internal state representation which provides more informative features. This agent uses linear neural networks to represent the value function and policy, which are trained by the TD error signal.

3 IMPLEMENTATION

Now we will compile many of the ideas from the previous section into a concrete software implementation. The goal here is to provide a practical tool for use in real applications. Its main intended users include engineers, roboticists, and game developers that need an out-of-the-box solution for their learning tasks. This tool, available online [2], 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 a concrete RL implementation in detail. It also functions as a platform for future research. For example, future work includes hierarchical data structures for states and actions (to improve scalability) and curiosity-driven exploration.

3.1 The Verve Library

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. The source code itself is heavily commented and unit tested. The downloadable distribution includes Python bindings and a set of example applications that validate the library's usefulness and provide example source code.

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, but this effect is more noticeable on simpler tasks that do not require much exploration.

Most of the features of this library are designed to solve the problems introduced in the previous section. The general architecture is same as the one developed above. 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 a dynamically-growing RBF system to combine sensory inputs. Actions are chosen using a roulette/softmax action selection scheme which maintains separate selection probabilities for each action. Agents can use any number of discrete and continuous sensors (discrete sensors take an index representing one of several distinct values, and continuous sensors take any real value between -1 and 1). Additionally, Verve agents incorporate more advanced features described in [3], including an internal uncertainty estimation, a learned predictive model for planning, and a curiosity drive to help improve the predictive model.

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.

For more specific details on the implementation (e.g., neural network learning rules), including the advanced functionality not described in this paper (e.g., planning with learned predictive models, curiosity), see [3].

3.2 A Code Sample

This section shows C++ 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. The first section of the code here defines an AgentDescriptor and creates an Agent and an Observation from the AgentDescriptor. The second part is a loop that continually computes the current Observation and reward, updates the Agent, and applies the Agent's chosen action to the environment.

```
// Define an AgentDescriptor.
verve::AgentDescriptor agentDesc;
agentDesc.addDiscreteSensor(4); // Use 4 possible values.
agentDesc.addContinuousSensor();
agentDesc.addContinuousSensor();
agentDesc.setContinuousSensorResolution(10);
agentDesc.setNumOutputs(3); // Use 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)
{
  // Set the Agent and environment update
  // rate to 10 Hz.
  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());

  // Compute the current reward, which is
  // application-dependent.
  verve::real reward = computeReward();

  // Update the Agent with the Observation and reward.
  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);
}
```

4 EXPERIMENTAL RESULTS

Here we provide a set of 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 test agents in a simple discrete maze environment and on two continuous control tasks: the pendulum swing-up task and the cart-pole/inverted pendulum task. In each case the task is specified simply by giving the agent a scalar reward value. The control policies are learned automatically.

4.1 2D Maze Task

This task tests an agent in a simple 2D maze environment (see Figure 4). 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. Additionally, the agent cannot cross interior wall boundaries. The experiment was run twice: once using discrete sensors, and once using continuous (i.e. radial basis function) sensors. Figure 5 shows the agent's learning performance.

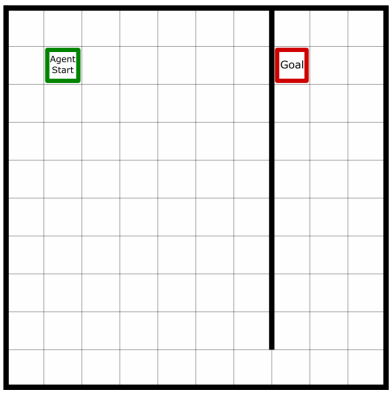

Figure 4: The layout of the environment in the 2D maze #1 task.

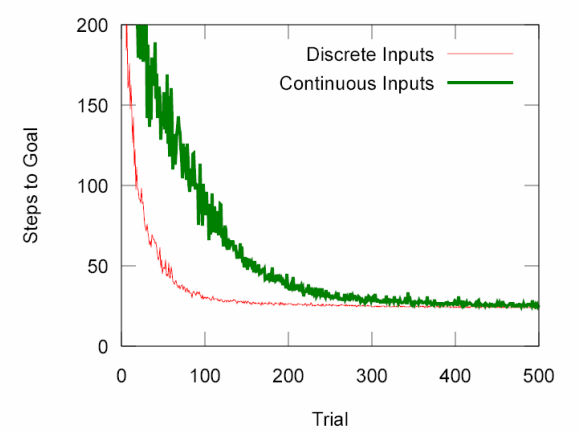

Figure 5: Learning performance on the 2D maze #1 task using the following parameters: policy learning multiplier = 1, position input discretization (for discrete inputs plot only) = 10, continuous sensor resolution (for continuous inputs plot only) = 15, number of runs averaged = 50.

Figure 6 shows the agent's learned value function over time. These images represent the agent's learned value of each state. (Lighter areas correspond to places that are more valueable to the agent.) 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.

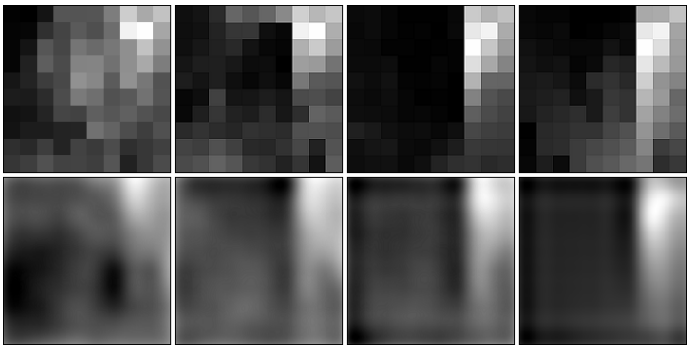

Figure 6: Learned value functions observed at the end of the 2nd, 5th, 10th, and 100th trials of the 2D maze #1 task, tested with discrete (top) and continuous (bottom) sensors. The following parameters were used: policy learning multiplier = 1, position input discretization (for the discrete sensors) = 10, continuous sensor resolution (for the continuous sensors) = 15.

4.2 Physical Control Tasks

We now cover results from two physical control tasks. 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 [9]. To simplify the process of constructing physically simulated environments, we used Open Physics Abstraction Layer [10]. 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.

4.2.1 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 7). 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.

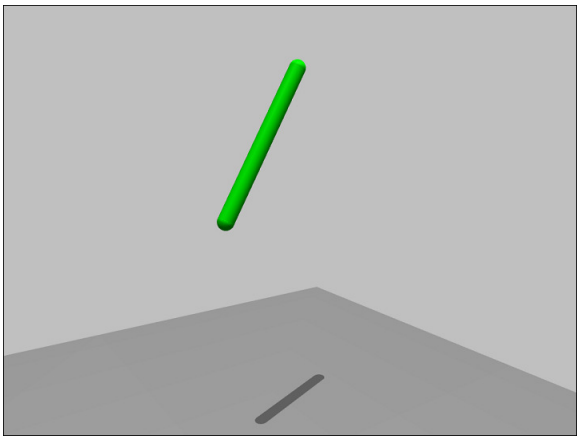

Figure 7: A physically simulated pendulum suspended in midair.

Figure 8 shows the agent's learning performance on the pendulum swing-up task. It reaches nearly optimal performance in about 60 trials.

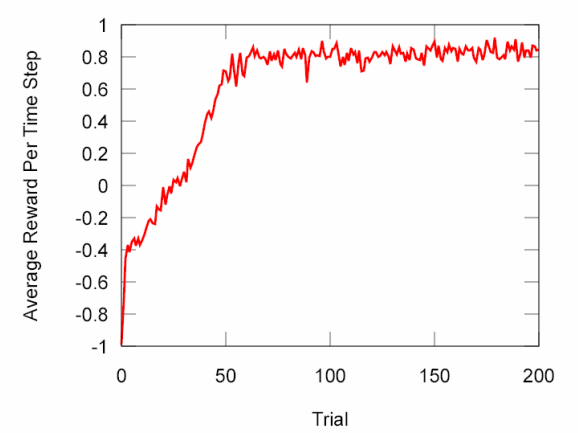

Figure 8: Learning performance on the pendulum swing-up task using the following parameters: physics step size = 0.01 s, agent step size = 0.1 s, pendulum mass = 1.0 kg, pendulum length = 1.0 m, ODE solver = "quickstep" (with 20 iterations per step), pendulum angle range = +/- 180 deg, pendulum angular 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 9 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.

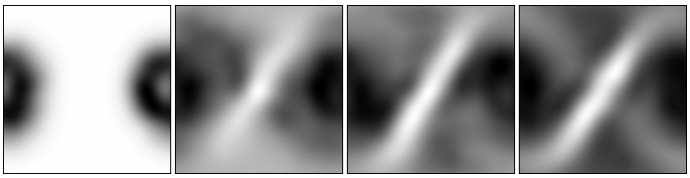

Figure 9: Learned value functions observed at the end of the 1st, 5th, 20th, and 100th trials of the pendulum swing-up task. The horizontal axis in each image is the pendulum's angle (i.e. the angle between the pendulum and vertical), whose range is +/- 180 deg and wraps directly from -180 to 180. The 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 10 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.

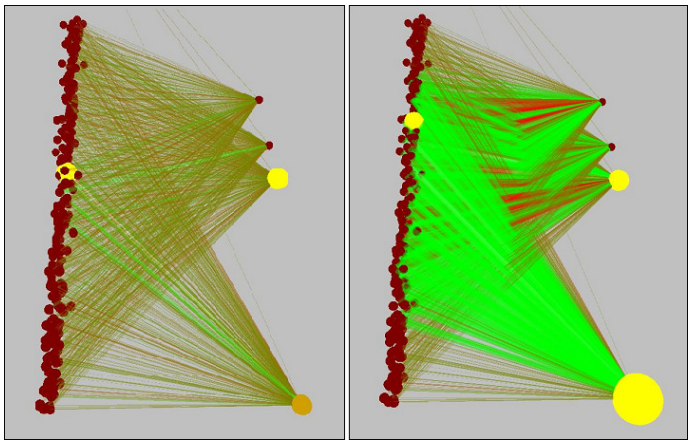

Figure 10: A visual representation of the pendulum agent's neural networks before (left) and after (right) learning. The neurons on the left side of each image are the state representation. The neurons on the top right represent the policy's three actions. The neuron on the bottom right represents the value function. Green connections are excitatory (positive); red connections are inhibitory (negative). Thicker connections have a larger weight magnitude.

4.2.2 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 11). 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).

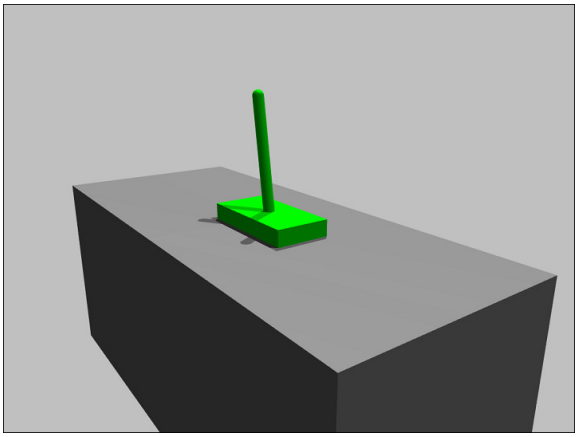

Figure 11: A physically simulated cart with an attached pole situated on a platform.

Figure 12 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.

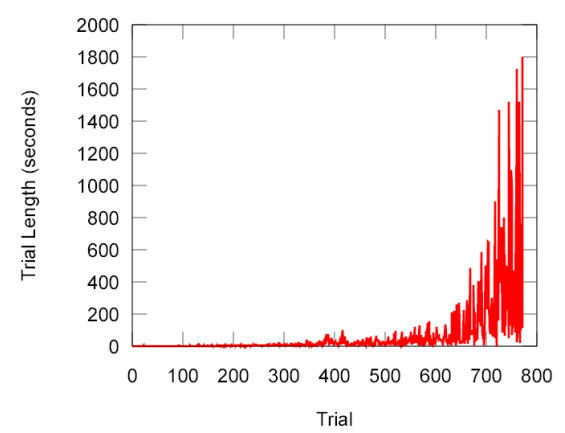

Figure 12: Typical learning performance for a single run of the cart-pole task using the following parameters: physics step size = 0.01 s, agent step size = 0.05 s, cart mass = 1.0 kg, pole mass = 0.1 kg, pole length = 1.0 m, coefficient of (static and kinetic) friction between cart and ground = 0, ODE 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.

5 CONCLUSIONS

This paper has discussed some of the key issues involved in designing general purpose RL agents. The result of this discussion was an agent design that uses temporal difference learning and an RBF state representation. After that we introduced the Open Source library Verve, a C++ implementation of the agent designed here. Finally, we showed results from several experiments that validate the library's effectiveness.

There are several limitations in the current implementation which will be addressed in future work. The main limitation is that the computational space and time requirements grow exponentially with the number of inputs. This is mainly due to the exhaustive state representation that combines all inputs into a higher level representation. This might be solved by using modular hierarchical policies, allowing agents to operate on low- and high-level sensory inputs and actions.

Another limitation is that the 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.

A third limitation is that the agents have no temporal state representation, so they cannot predict future events at specific times. One possible solution is to use a tapped delay line scheme [8], enabling the agents to learn temporal correlations between events.

An avenue of investigation currently underway is curiosity-driven exploration [11, 12, 13], a topic related to the field of developmental robotics. Rather than rely solely upon random action selection, an intrinsic curiosity drive can motivate agents to explore "interesting" parts of the state space. This trains predictive models (used for planning) more efficiently. Such curious agents decide for themselves which situations are worth exploring.

The open-endedness of the Verve library makes it applicable to a wide variety of tasks. Virtual characters in computer simulations (e.g., video games, virtual reality training scenarios) could learn to animate themselves. User interfaces for personal computing devices could adapt themselves through trial-and-error learning. Agents controlling real machines (e.g., mobile robots, unmanned aerial vehicles) could train themselves with minimal human teaching. These agents could even learn primarily in simulations before being transferred to physical robots, making the training process cheaper, safer, and faster. Any problem that can be formulated as an RL problem is a potential application for Verve agents.

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

ACKNOWLEDGMENTS

This work was partially supported by a grant from the Air Force Office of Scientific Research.

REFERENCES

1. Sutton, R.S. & Barto, A.G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.
2. Verve project website. http://verve-agents.sourceforge.net (accessed 2-5-06).
3. Streeter, T. (2005). Design and Implementation of General Purpose Reinforcement Learning Agents. Unpublished Master's Thesis, Iowa State University.
4. RL Toolkit project website. http://rlai.cs.ualberta.ca/RLAI/RLtoolkit/RLtoolkit1.0.html (accessed 2-5-06).
5. Reinforcement Learning Toolbox project website. http://www.igi.tugraz.at/ril-toolbox/general/overview.html (accessed 2-5-06).
6. PIQLE: a Platform Implementing Q-LEarning algorithms in JAVA project website. http://www.lifl.fr/~decomite/piqle/index.html (accessed 2-5-06).
7. SALSA (System using Artificial Life to Study Adaptation) project website. http://www.cs.indiana.edu/~gasser/Salsa (accessed 2-5-06).
8. Haykin, S. (1999). *Neural Networks: A Comprehensive*

Foundation, 2nd Edition. Prentice-Hall.
9. Open Dynamics Engine project website. http://www.ode.org (accessed 2-5-06).
10. Open Physics Abstraction Layer project website. http://opal.sourceforge.net (accessed 2-5-06).
11. Oudeyer, P.-Y., and Kaplan, F. 2004. Intelligent adaptive curiosity: a source of self-development. In Berthouze, L., et al, eds., *Proceedings of the 4th International Workshop on Epigenetic Robotics*, volume 117, 127-130. Lund University Cognitive Studies.
12. Schmidhuber, J. 1991. Curious model-building control systems. In *Proceedings of the International Joint Conference on Neural Networks*, vol. 2, 1458-1463. IEEE.
13. Streeter, T. 2006. Curiosity-Driven Exploration with Planning Trajectories. *In Proceedings of the Twenty-First National Conference on Artificial Intelligence (to appear).*

