Applying Reinforcement Learning to Obstacle Avoidance

Josh Beitelspacher JOSH@CS.OU.EDU
University of Oklahoma, 308 Cate Center Drive Box 5242, Norman, OK 73072 USA

Abstract

This paper applies reinforcement learning techniques to an asteroids-type game. Both Q-Learning and Sarsa(λ) are used to learn obstacle avoidance. Action-value function approximation is provided by a backpropagation neural network. The combination of these two methods produces obstacle avoidance strategies that perform at nearly a human level.

1. Introduction

Reinforcement learning is easy to apply to simple problems, but using reinforcement learning for more complicated tasks is less straightforward. The most well known application of reinforcement learning is TD-Gammon (Tesauro, 1995). Other applications have shown a varying degree of success (Stone & Sutton, 2001).

TD-Gammon learned to play backgammon using temporal difference learning. A neural network was used to approximate the value of each possible state. As the program learned, it adjusted the weights of the network to better approximate the value function. The combination of neural networks and reinforcement learning can be applied to a wide variety of problems.

The major difficulty in any complicated reinforcement learning system is state representation and function approximation. Currently both of these difficulties have to be handled by trial and error. Therefore, the application of reinforcement learning to a new domain is a non-trivial process.

This paper describes a working system for obstacle avoidance in an asteroids-style game. This game is a good test bed for reinforcement learning because it requires a continuous state space with a large number of features. The system we devised uses Q-Learning or Sarsa(λ) to learn a control policy. Backpropagation neural networks are used for function approximation.

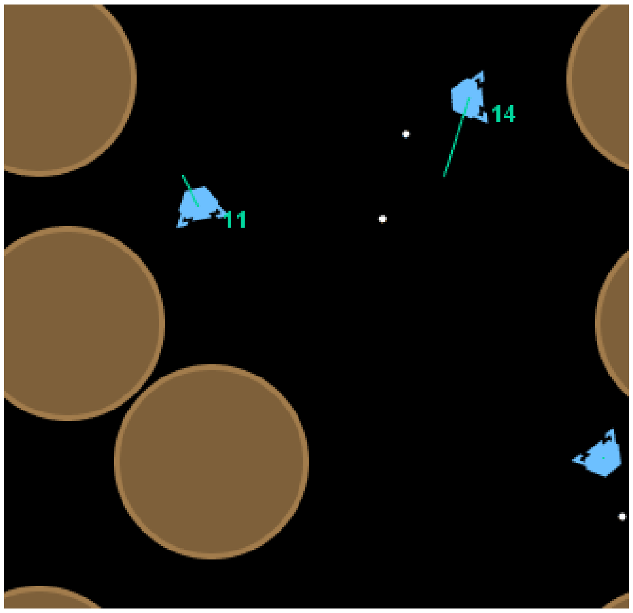

Figure 1. A screenshot from the Spacewar simulator. Note that a single obstacle is visible in all for corners of the screen. The white dots are bullets fired from ships.

2. Problem Definition

The problem domain is an asteroids-type game. In our version of the game, which we will call Spacewar, n ships and m obstacles exist in a two dimensional rectangular space. An object maintains its current velocity unless acted upon by an external force. The only external forces are provided by collisions or by a ship's thruster. Objects that go off one edge of the screen appear on the other edge.

Each ship is controlled by a player. At each timestep the player can turn left, right, thrust, or shoot. Combined actions are also possible. For example a player

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

can turn left, thrust, and shoot at the same time. Decisions are made every 1/30 second. Ships are damaged, and eventually destroyed, by collisions with other objects. The goal of the game is to outlive all over ships.

The simulator (Figure 1) was written in Java. The source code and compiled releases are available online at `http://spacewar.netbeetle.com/`. The simulator approximates all objects with a single circle. Collisions between objects are calculated as precisely as possible. If objects would have collided at any point during a timestep the collision will be detected even if the objects are not overlapping at the end of the timestep. The physics code has been thoroughly tested, and no exploits have been found.

2.1. Task Definition

Spacewar is a very complicated task. Rather than taking on the entire problem, this paper focuses on a subproblem. In order to win a Spacewar game, a player must be able to avoid collisions with other objects. The obstacle avoidance subproblem is critically important, and it is still a difficult task. The rest of the paper only discusses obstacle avoidance.

The obstacle avoidance problem is defined as attempting to move the ship in such a way that it will not collide with any obstacles. Performance at this task can be measured by the average amount of time before a collision.

We restrict the problem even further by setting the number of obstacles and the size of the space. We use six obstacles with a 50 meter radius and one ship with a 10 meter radius. These seven objects are randomly positioned in a 500 by 500 meter space such that there is a reasonable buffer distance between all the objects. The obstacles are also assigned a random velocity between 0 and 30 meters per second.

This task is ideal for reinforcement learning. It is simple enough to easily cast as a Markov decision process, yet the continuous state space make the task difficult enough to require the use of function approximation. Ideally, using reinforcement learning will also produce a control policy that is better than one based on heuristics alone. In fact, the interactions between obstacles would make it quite difficult to write a program to handle this task. The number of actions that can be performed in a game also makes it impossible to thoroughly search the state space.

2.2. Learning Algorithm

Q-Learning and Sarsa(λ) (Sutton & Barto, 1998) will both be used for reinforcement learning. Later we will compare the relative performance of the two algorithms. These two algorithms are very similar, and we expect them to produce similar results when run with identical parameters.

Reinforcement learning algorithms typically work on a sequence of states, actions, and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, \cdots, s_t, a_t, r_{t+1}, \cdots, s_n$$

The agent receives state information, decides which action to choose, and receives a reward based on the chosen action. The goal of a reinforcement learner is to maximize the sum of the rewards over an episode.

Q-Learning and Sarsa are both temporal difference (TD) learning methods. At all times they model a function $Q(s,a)$, where s is a state and a is an action. The value $Q(s,a)$ is the sum of the expected future rewards that will be received after taking action a from state s:

$$Q(s_t, a_t) = r_{t+1} + r_{t+2} + \cdots + r_{t+n}$$

Given state s_t and possible actions $a_0, a_1, \cdots, a_n$, we decide which action to choose by finding an action a_t such that $Q(s_t, a_t) \geq Q(s_t, a_j)$ for all j from 0 to n. This action is the optimal action according to the current policy.

After taking action a_t and receiving reward r_{t+1}, we find the next action to take by repeating the same procedure. The expected future reward following the next action is $Q(s_{t+1}, a_{t+1})$. If our function is optimal, the last expected reward should equal the actual one-step reward plus the new expected reward:

$$Q(s_t, a_t) = r_{t+1} + Q(s_{t+1}, a_{t+1})$$

While learning is taking place, the above equality will not hold. We will define the temporal difference error (TD_{error}) to be the difference between the expected rewards:

$$TD_{\text{error}} = r_{t+1} + Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

Instead, we will update $Q(s_t, a_t)$ to reduce the TD error. The amount to update is given by a step size parameter, α:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot TD_{\text{error}}$$

For continuous tasks, the sum of expected future rewards may be infinite. In this case it is possible to redefine the above equations to include a discount parameter, γ. If we include this parameter we get the following new equations:

$$Q(s_t, a_t) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n}$$

$$Q(s_t, a_t) = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$$

And we can define a new equation for the TD error:

$$TD_{\text{error}} = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

The system as defined above always takes the optimal action, but it is usually necessary for an agent to explore the state space more fully. This is typically handled by introducing another parameter, ε, the probability of choosing a random action at any timestep.

Sarsa(λ) has one additional parameter, the eligibility trace decay rate, λ. Setting this parameter to a non-zero value causes previously visited state-action pairs to receive a portion of the TD error on future timesteps. The amount of TD error that is given to previous actions is defined by λ. At $\lambda = 0$, TD error is only attributed to an action when it is taken. At $\lambda = 0$, TD error is attributed to all previously visited state-action pairs. Usually a value somewhere in the middle performs best. In our testing we set λ to 0 for the first test, and λ to 0.55 for the last test. It is also possible to use elgibility traces with Q-Learning, but it is not as clear how exactly that should work. In our testing Q-Learning is run without eligibility traces, so it is best compared to Sarsa(0).

The only difference between Q-Learning and Sarsa(0) is the handling of exploratory random actions. In Sarsa(0) a random action, a_{t+1}, will be used in the update rule described above. In Q-Learning if the action a_{t+1} is random, it is not used in the update step. Instead, the optimal action according to the current policy, a^{π}_{t+1}, is used in the update step.

In order to apply this framework to obstacle avoidance it is necessary to define our states, actions and rewards. The state representation is the most complicated part of this process, and it is discussed in the next subsection. The possible actions are {no action, left, right, thrust, left $\wedge$ thrust, right $\wedge$ thrust}. For each timestep on which no action is carried out, a reward of 0.1 is given. For each timestep on which any other action is carried out, a reward of 0 is given. On the final timestep of the game a reward of -10 is given to penalize the collision.

The discount rate, γ, is set to 0.9, therefore the max cumulative reward is 1. The exploration probability, ε, is set to 0.01, and the step size, α, is varied over the course of the experiments. Although the Spacewar program accepts commands every 1/30 second, we will only update our command at 1/3 second intervals. This will hopefully increase the speed of learning, without damaging the quality of the results to too high a degree.

2.3. State Representation

When using a neural network with reinforcement learning, the choice of a good state representation is critical. Both reinforcement learning and neural networks impose restrictions on the state representation. In order for learning to occur all the restrictions must be considered.

In order for reinforcement learning algorithms to find optimal policies, they require the state representation to be Markovian. In order to be Markovian the state must retain all information necessary to predict the future. It is usually not possible to store a true Markov state, so an approximation has to suffice.

In the Spacewar domain, a Markov state would record the positions and actions of all ships at all previous timesteps. We need this information to accurately predict how the opponents will respond. For example, a player may be aggressive in some circumstances and cautious in others. If we want to predict the actions of that player, we need to remember how he acted in similar situations before.

In the obstacle avoidance subproblem there are no opponent ships, so we don't need to record the complete history of the game. Obstacles are controlled only by physics. The movement of the obstacles can be predicted given their positions and velocities. Therefore, to maintain the Markov property all we need to retain is the current positions and velocities of all the objects.

While the positions and velocities are all that is required for reinforcement learning, it may not be the best state representation for a neural net. Neural nets are powerful function approximators because they have the power to generalize well. A network that has never seen a state can return a value that is appropriate to that state if it has been trained on many similar states. Over the course of training a network is able to determine the relative importance of all the inputs and weight them accordingly.

Training is made much more efficient if states that are similar have representations that are also similar. This was not the case in the representation described above. For example, the representations for a state is different depending on the positions of all the objects. If the position of every object is offset by the same amount, this representation will look drastically different although the relative positions of all the objects would be identical.

Egocentric coordinates are a simple solution to this problem. If the position of every object is defined relative to the ship, then the network will have a much

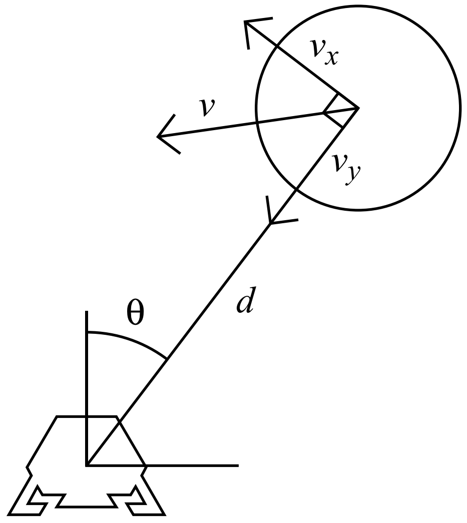

Figure 2. The state representation for a single obstacle. An obstacle is described by θ, d, v_x, and v_y.

better ability to generalize. This representation makes similar states look much more alike. A single obstacle position can be represented by four variables: θ, the angle to the obstacle; d, the distance to the obstacle; v_x, the velocity of the object around the ship; and v_y, the velocity of the object toward the ship. Velocities are relative to the current ship velocity. These parameters are shown graphically in Figure 2.

It is also possible to sort the obstacles by distance, such that the first obstacle in the state representation is always the closest obstacle to the ship. This modification was added after the first set of tests were run, and time constraints made it impossible to re-run the tests.

Angles and distances seem like relatively abstract information to give a neural net, but they have been used successfully in other work (Stone & Sutton, 2001). The complete state description is therefore 24 floating point numbers, four numbers for each of the six obstacles.

Objects can wrap around the edge of the screen, so it is possible to measure the distance between objects in an infinite number of ways. When defining the egocentric coordinates, the shortest possible distance from the ship to each obstacle is used. This means that obstacles that appear on opposite sides of the ship may actually be close together even though they appear far apart.

The state representation described here does not include this information in any way, therefore it is non-Markovian. However, this fact only influences obstacles that are far away from the ship. Obstacles close to the ship move in a predictable manner, and it is still possible to learn a good policy for obstacle avoidance.

2.4. Function Approximation

When we use Q-Learning or Sarsa(0), standard multilayer backpropagation neural networks (Mitchell, 1997) are used to approximate the $Q(s,a)$ function. At all times six actions are possible {no action, left, right, thrust, left $\wedge$ thrust, right $\wedge$ thrust}. As described by Thrun (1995) it is possible to represent this function with a single neural network. However, we chose to use six different networks, one for each possible action.

Each network had 24 input neurons that took their values from the state representation described previously. These inputs were scaled so that they were roughly between -1 and 1. The input layer was fully connected to a layer of 20 hidden sigmoid units, and the hidden layer was connected to a single linear output unit.

To find the best action at any time, all six nets were evaluated and the action corresponding to the net that returned the highest value was chosen. To update a $Q(s,a)$ value, the net corresponding to action a was updated using the backpropagation algorithm.

In contrast, when we used Sarsa(λ), a different network structure was required. Eligibility traces can not be handled efficiently when multiple networks are used. In order to use eligibility traces with any function approximation system, it is necessary to have a single vector of parameters, $\vec{\Theta}$, that are used in the approximation of the $Q(s,a)$ function. When we use a single network, this vector is composed of all the weights of the network. Each weight, w_{ji}, also has an associated eligibility trace, e_{ji}.

In order to use a single network, we needed to modify our network structure. We used the same input layer as discussed above, but modified the output layer to include one neuron for each potential action. The number of hidden neurons was also increased to give the network more representational power.

With this setup, the optimal action could be found with the evaluation of a single network. On update steps, backpropagaton was done using only the neuron representing the chosen action. The additional neurons on the output layer were simply ignored.

Instead of updating the action-value function using backpropagation, we want to make use of eligibility traces to directly set our parameters. Backpropagation is still used to find the gradient of network weights with respect to the TD error, but instead of using the normal backpropagation update rule,

$$w_{ji} \leftarrow w_{ji} - \frac{1}{2}\alpha\frac{\partial TD^2_{\text{error}}}{\partial w_{ji}},$$

we will add in eligibility traces according to the following equations (Sutton & Barto, 1998):

$$e_{ji} \leftarrow \gamma\lambda e_{ji} + \frac{\partial Q(s_t, a_t)}{\partial w_{ji}}$$

$$w_{ji} \leftarrow w_{ji} + \alpha \cdot TD_{\text{error}}\frac{\partial Q(s_t, a_t)}{\partial w_{ji}}.$$

This is easily accomplished by noting the following relation:

$$\frac{\partial Q(s_t, a_t)}{\partial w_{ji}} = -\frac{1}{2 \cdot TD_{\text{error}}}\frac{\partial TD^2_{\text{error}}}{\partial w_{ji}}.$$

When these algorithms are used in such a way, eligibility traces become a feature of the neural network instead of a feature of the learning algorithm. In our tests, we gave the network an eligibility trace discount rate of 0.5. However, when comparing with other results of Sarsa(λ), it is more appropriate to say that λ was 0.55, or $0.5/\gamma$, where $\gamma = 0.9$.

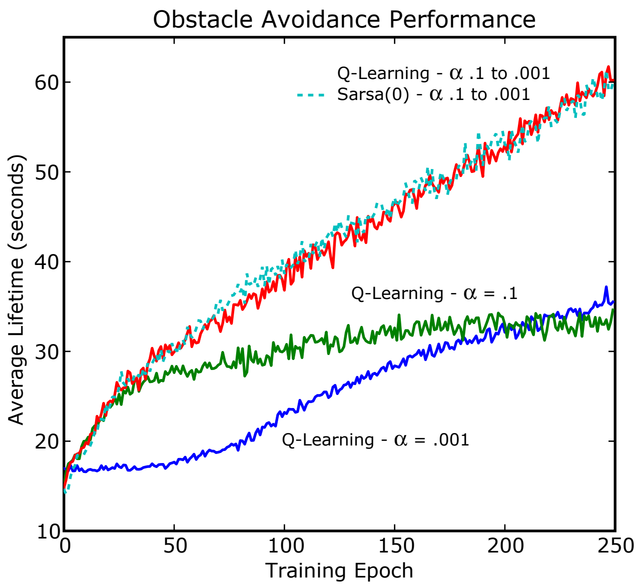

Figure 3. Results of Q-Learning and Sarsa(0). Each epoch consists of 4000 games.

3. Experimental Evaluation

3.1. Methodology

The performance measure for obstacle avoidance is average lifetime. The performance of each algorithm will be plotted over 1,000,000 games. The initial positions of objects in the environment can have a large impact on how long a ship can survive, so the data will not form a smooth curve. In fact, it is hard to see any pattern when all games are plotted. In order to visualize the data better, we will divide a run of 1,000,000 games into 250 epochs of 4000 games each. Performance over each set of 4000 games is averaged, and these 250 points are plotted.

Results are not averaged over multiple runs due to time constraints in making repeated runs. However, the results presented are typical of all the experience we have.

The best naive algorithm for this problem is to do nothing. If the ship can not be moved in an intelligent way, it is much better off doing nothing than moving randomly. A ship that stays still averages about 18 seconds, while a ship that moves randomly averages about 10 seconds.

We hypothesize that a machine learning algorithm should be able to do substantially better than either of the naive methods. Also, we hypothesize that Sarsa(λ) should learn significantly faster when λ is not set to 0.

3.2. Results

The first algorithm tested was Q-Learning with α set to 0.1. This algorithm performed consistently better than random almost immediately, and began to outperform “do nothing” at about 40,000 games. Most of the learning happened during the first half of the run, after which results improved very little.

In the next test we lowered the learning rate, α, to 0.001. This was an attempt to allow the algorithm to continue learning longer than on the previous attempt. This change drastically reduced the amount of learning in the first half of the run, but it increased learning in the second half enough to pass the first algorithm near the end of the run. It is worth noting that for a long period of time, this algorithm used the naive strategy of doing nothing.

The next run used a variable learning rate. The learning rate started at 0.1 and linearly decreased to 0.001 over the course of the run. The goal of this run was to duplicate the early success of the first run, with a high learning rate, and the late success of the second run, with a lower learning rate. It succeeded on both points.

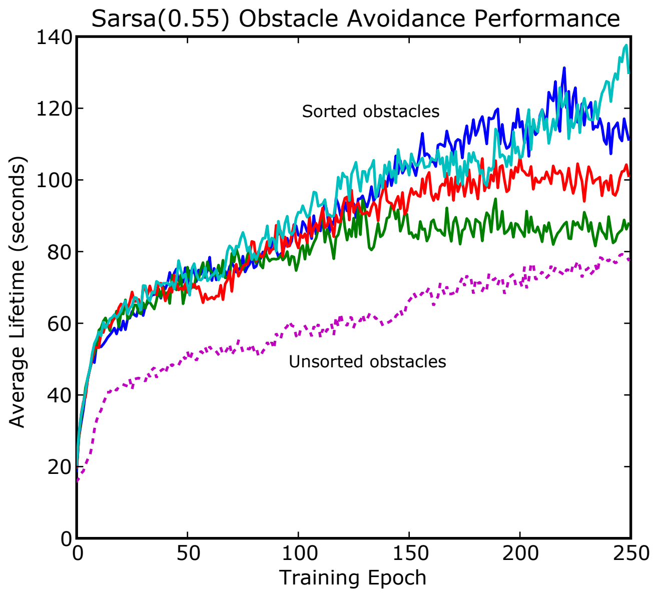

Figure 4. Results of four runs of Sarsa(0.55). Each epoch consists of 4000 games.

For the next run we switched learning algorithms to Sarsa(0). Sarsa is a slightly faster algorithm, because on exploratory steps no neural networks have to be evaluated. The difference in the amount of time taken by this algorithm turned out to be insignificant, but the performance of this algorithm was almost identical to Q-Learning with the same parameters.

Finally, we ran four runs of Sarsa(0.55). On each of these runs we were able to learn a policy that performed as well as any previous run after around 1/10 the number of games. However, late in the runs a significant amount of variation developed. These runs used the same parameters as the previous runs, and the learning rate was varied in the same way. A single run was done in which the obstacles were not in sorted order. This run can be accurately compared to the previous results, while the 4 runs with sorted obstacles had a distinct advantage. The comparison would be much stronger if sorted obstacles had been used in all tests, but time constraints made that impossible.

Learning curves for all algorithm are shown in Figure 3 and Figure 4.

3.3. Discussion

The results obtained exceeded our initial expectations. All the algorithms tested significantly outperformed the simple strategies that we described. The performance of Sarsa(0.55) was impressive. Learning could be witnessed within the first 10,000 runs. This contrasted strongly with the other approaches in which it was difficult to visually decide if learning was taking place until around half way into the run.

The variable learning rate runs of Sarsa and Q-Learning appear to still be learning at the end of the run. An attempt was made to continue these runs, but the learning immediately slowed to a negligible level. We think this is because the learning rate was held constant after the first 1,000,000 games were finished.

The act of decreasing the learning rate actively stimulates learning. As long as the learning rate decreases the information learned in one game is unlikely to be completely erased in the next. Once the learning rate is held constant random fluctuations in the function approximators stop the system from improving.

This raises questions about the way in which the learning rate should be manipulated. For example, maybe learning would go faster if the learning rate changed slower at the end of the run than at the begin of the run. In our testing we only adjusted the learning rate linearly with the number of games played.

4. Future Work

The subproblem we are looking at is too tightly constricted. The system described in this paper only works for obstacle avoidance when there are exactly six obstacles. In order to change the number of obstacles an entirely new network would have to be learned. A more general solution would be able to handle any number of obstacles, as well as ships and bullets. This may not be that large of a problem, the learning speed makes it feasible to learn on each subproblem individually.

However, if this system were to be adapted to an real world system, we would need a way to greatly reduce the number of runs. The current algorithms are capable of learning in simulation only. So, even though it is possible to learn a good policy in a few minutes, we would still like to greatly reduce the number of runs.

Finally, the parameters of the learning algorithms were all set somewhat arbitrarily. It would be informative to systematically adjust these values and record the results.

5. Conclusion

This paper presented a reinforcement learning algorithm in a previously unexplored domain, and was able to give promising results. The algorithms were able to learn complicated control policies that would be quite difficult to implement by hand.

Spacewar is a fun and challenging domain to work with. It can be used as a competitive environment, a non competitive environment, or even a team coordination environment. The continuous state space and continuous time make the domain a great testing ground for the latest and most advanced algorithms. Hopefully more work will be done on this problem.

References

Mitchell, T. M. (1997). *Machine learning*. New York, NY, USA: McGraw-Hill.

Stone, P., & Sutton, R. S. (2001). Scaling reinforcement learning toward RoboCup soccer. *Proceedings of the 18th International Conference on Machine Learning* (pp. 537–544). Morgan Kaufmann, San Francisco, CA.

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

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, *38*, 58–68.

Thrun, S. (1995). An approach to learning mobile robot navigation. *Robotics and Autonomous Systems*, *15*, 301–319.