The MWAL Algorithm for Apprenticeship Learning and Applications to Large MDPs

Robert Timpe

Spring 2011

Advisor: Prof. Schapire

Abstract

The goal of apprenticeship learning is to learn how to behave in an environment with an unknown reward function based on observations of the behavior of an expert. Early work was done by Abbeel and Ng [1] and was later improved upon by Syed and Schapire [2] in the MWAL algorithm. In addition to being less complicated and more efficient than earlier algorithms, the MWAL algorithm has the added advantage of not being upper bounded by the performance of the expert. That is, the MWAL algorithm is still guaranteed to do as well as the expert, but can perform better in some situations. Our goal is to explore the performance of the MWAL algorithm in both small and large MDPs. We first apply the algorithm to a simple grid world and show that while it does outperform a bad expert, it has trouble matching an optimal expert. We then apply the algorithm to the game Tetris to show that the technique of function approximation works reasonably well with this algorithm. While our choice of function approximators prevents the algorithm from playing well, our results indicate that a more sophisticated function approximator might have better results.

Introduction

When an agent has to make decisions in a (possibly randomized) environment, the environment is often modeled as a Markov Decision Process, or MDP. An MDP can be characterized as a set of states and transitions between them. The agent has a choice of various actions, which may result in a transition from the

agent's current state to some other (usually different) state. We will assume that the agent always knows what state it is in (i.e. the MDP is fully observable) and is allowed to choose its next action, but that the chosen action may not have the desired result with some known probability. That is, the agent may try to move North, but end up moving East with some probability. The MDP also has a reward function that specifies a reward for every possible state.

The goal of the agent in an MDP is to come up with an optimal policy. A policy is simply a function that tells the agent which action to take from each possible state. An optimal policy is one that maximizes the agent's expected reward. This is often done by computing the utility for each state and choosing a policy that maximizes expected utility. The utility of a state can be thought of as the expected sum of (discounted) rewards attainable from that state. When the reward function is known and the state space is small, there are well-known algorithms for finding an optimal policy (i.e. value iteration and policy iteration). However, these techniques fail when the state space is very large and/or the reward function is unknown (both common occurrences in real-world applications).

One attractive solution to the problem of finding optimal policies in large MDPs is reinforcement learning. In reinforcement learning, the agent attempts to learn an optimal policy by wandering around the state space. When it finds states with low (possibly negative reward) it knows to give them a correspondingly low utility, and similarly for states with high rewards. The hope is that the agent will eventually learn to avoid states with low reward and focus on states with high rewards.

Apprenticeship learning is closely related to reinforcement learning. However, it is assumed that the true reward function is unknown (so the agent can't just wander around, getting rewards since we might not even know what reward it should receive in each state). Instead, the reward function is modeled as a linear combination of "features", which should be easy to come up with for a given MDP [1]. Additionally, rather than try to learn a policy that is optimal with respect to this unknown reward function, the agent instead tries to imitate the performance of an expert. This framework was first proposed by Abbeel and Ng [1] who motivate its usefulness by pointing out that in many real life situations, it is difficult to provide an exact reward function, but easy to come up with features that play a part in determining the true reward. For example, they talk about the problem of coming up for a reward function for driving. It would be difficult to come up with a single, accurate reward function, but it is not hard to imagine a number of features that would be involved in its computation. In particular, they mention speed, number of collisions, and so on as examples of features that would be used. They then gave an algorithm that provides both an upper and lower bound on the performance of the agent with respect to the expert. That is, the agent will perform at least as well as the expert with respect to the unknown reward function, but it will perform no better.

Syed and Schapire [2], by looking at the problem from a game theory point of view, described the MWAL algorithm, which represents a strict improvement over the work of Abbeel and Ng [1]. This algorithm is both simpler and more efficient to

implement. But perhaps more importantly, it allows for the possibility that the agent might actually do better than the expert.

Our goal is to explore the performance of the MWAL algorithm on two different MDPs. The first, called Mines World [3] is a simple grid world with fewer than 100 different states. The agent can move North, South, East or West and wants to get to the goal state, which has a positive reward. However, there are states with mines in the way, and landing on a mine results in a large negative reward. In looking at the Mines world, we hope to answer two questions. First of all, we will explore the performance of the algorithm given experts of various skill levels to see whether the MWAL algorithm is truly capable of outperforming the expert. We will then repeat the experiments with some modifications to the reward features to see what kind of impact changing them has on the performance of the agent. Choice of features is central to the performance of the MWAL algorithm, and by exploring the effect of changing their values we hope to provide insight into how they should be selected.

We then apply the MWAL algorithm to the game of Tetris, which has a much larger state space. In doing so we hope to discover how well the MWAL algorithm does when combined with a simple approximation technique and provide direction for future efforts to apply the MWAL algorithm, whether to Tetris or other large MDPs. Specifically, we use a simple form of function approximation to represent the utility of each state. While this form of approximation does not result in the algorithm playing good Tetris, our results indicate that a more sophisticated form of function approximation might be more successful.

The MWAL Algorithm

Before we discuss the MWAL algorithm in detail, we will first define an MDP, which we define as in Syed and Schapire [2]. An MDP *M* consists of a set of states *S*, a set of possible actions *A*, a discount factor γ, an initial state distribution *D*, a transition function $\theta(s,a,s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ and a set of *k* features given by the features function $\phi : S \to \Re^k$. This is different from a classical MDP, where the reward function is known directly. In our case, we assume that the true reward function is $R^*(s) = w^* \cdot \phi(s)$ for some $w^* \in \Re^k$. The value of a policy π is defined as $V(\pi) = E\left[\sum_{t=0}^{\infty} \gamma^t R^*(s_t) \mid \pi, \theta, D\right]$. Similarly, Syed and Schapire [2] also define a "feature expectations vector" for some policy π as $\mu(\pi) = E\left[\sum_{i=0}^{\infty} \gamma^i \phi(s_i) \mid \pi, \theta, D\right]$. As Syed and Schapire [2] point out, this quantity might be more accurately called the "expected, cumulative, discounted feature values" and is used in lieu of a reward function to measure the effectiveness of a policy. Syed and Schapire [2] also assume that the expert follows some policy π_E (which need not be deterministic) and that the algorithm has access to information about this policy in the form of *m* independent trajectories in *M*, each of length *H* (the algorithm itself doesn't require this, it just makes the math easier). The *i*th trajectory is a series of states $(s_0^i, s_1^i, \ldots, s_H^i)$ visited by the expert. The expert's feature expectations vector can then be estimated as $\hat{\mu}_E = \frac{1}{m} \sum_{i=0}^{m} \sum_{t=0}^{H} \gamma^t \phi(s_t^i)$. For the MWAL algorithm it is also important to

estimate some of these quantities, so Syed and Schapire [2] define $\hat{\mu}$ to be an ε-good estimate of a feature expectations vector $\mu(\pi)$ if $\left\|\hat{\mu} - \mu(\pi)\right\|_\infty \leq \varepsilon$. Similarly, they call a policy $\hat{\pi}$ ε-optimal if $\left|V(\hat{\pi}) - V(\pi^*)\right| \leq \varepsilon$ where π^* is an optimal policy for *M*. Finally, we need to introduce the concept of a mixed policy. A stationary policy is a policy that consists of a single function mapping from states to actions. A mixed policy ψ is a probability distribution over Π, the set of all possible deterministic stationary policies for *M*. Since there are only finitely many policies in Π, we can number them from $\pi^1,\ldots,\pi^{|\Pi|}$ and treat ψ as a vector where $\psi(i)$ is the probability assigned to the i^{th} policy. We use the policy ψ by choosing a policy $\pi^i \in \Pi$ randomly with probability $\psi(i)$ at time 0, and then following that policy for the rest of that run through the MDP. Finally, we are ready to see the MWAL algorithm, which Syed and Schapire [2] define as

Algorithm 1 The MWAL algorithm

Given: An MDP\R M and an estimate of the expert's feature expectations $\hat{\boldsymbol{\mu}}_E$.
Let $\beta = \left(1 + \sqrt{\frac{2\ln k}{T}}\right)^{-1}$.
Define $\tilde{\mathbf{G}}(i, \boldsymbol{\mu}) \triangleq ((1-\gamma)(\boldsymbol{\mu}(i) - \hat{\boldsymbol{\mu}}_E(i)) + 2)/4$, where $\boldsymbol{\mu} \in \mathbb{R}^k$.
Initialize $\mathbf{W}^{(1)}(i) = 1$ for $i = 1, \ldots, k$.
for $t = 1, \ldots, T$ **do**
  Set $\mathbf{w}^{(t)}(i) = \frac{\mathbf{W}^{(t)}(i)}{\sum_i \mathbf{W}^{(t)}(i)}$ for $i = 1, \ldots, k$.
  Compute an ϵ_P-optimal policy $\hat{\pi}^{(t)}$ for M with respect to reward function $R(s) = \mathbf{w}^{(t)} \cdot \boldsymbol{\phi}(s)$.
  Compute an ϵ_F-good estimate $\hat{\boldsymbol{\mu}}^{(t)}$ of $\boldsymbol{\mu}^{(t)} = \boldsymbol{\mu}(\hat{\pi}^{(t)})$.
  $\mathbf{W}^{(t+1)}(i) = \mathbf{W}^{(t)}(i) \cdot \exp(\ln(\beta) \cdot \tilde{\mathbf{G}}(i, \hat{\boldsymbol{\mu}}^{(t)}))$ for $i = 1, \ldots, k$.
end for
Post-processing: Return the mixed policy $\overline{\psi}$ that assigns probability $\frac{1}{T}$ to $\hat{\pi}^{(t)}$, for all $t \in \{1, \ldots, T\}$.

Figure taken from Syed and Schapire [2]

We assume that the features are all in the range [-1,1] and that the weights are all positive and sum to 1. By making this choice, the algorithm can view the

problem of finding an optimal policy as a two player game. Syed and Schapire [2] provide more details, but the basic idea is that the agent tries to find a policy that performs at least as well as the expert while the environment tries to choose a reward function that causes the agent to perform as poorly as possible. This might seem strange, but since the true reward function is unknown, it makes sense to assume that it is being chosen by an adversary. The critical portion of the algorithm consists of steps 6-9 above. In these steps, the algorithm first normalizes the set of weights, then chooses an optimal policy based using them as a reward function. It then evaluates its newly chosen policy by computing the feature expectations vector. Finally, it updates the weights based on the difference between its feature expectations vector and the expert's.

This range of values for the features is what allows the MWAL algorithm to (possibly) outperform the expert. By making sure the weights are all positive but allowing the features to be negative we are making two assumptions. First, we assume that positive features are good and negative ones are bad. Secondly, we assume that the sign of the feature is correct (i.e. we don't change the fact that a negative feature is bad) and our job is just to guess the magnitude of the weight for each feature. So if an expert always gets a negative value for some feature, the MWAL algorithm can do better by getting a positive value. This makes the choice of features especially important – a good choice of features (and in particular, their signs) can give the agent quite a bit of information about the MDP outside of what it gets in the form of expert trajectories.

Note that steps 7 and 8 are not precisely defined, and there are a variety of ways that one might go about implementing them. We defer a description of step 7 until later because it was implemented differently in the two different environments. However, step 8 was implemented in the same way each time. Rather than attempt to compute the feature expectation vector directly, we approximated it by running the policy $\hat{\pi}^{(t)}$ repeatedly and computing the feature expectation vector for each run, then taking the average (similar to the way the expert's feature expectation vector was estimated).

The Mines World

The Mines world is a simple grid world. The code for this environment was obtained through the rl-library project and was originally written by Adam White [3]. In our version, it consists of a board 18 spots wide and 7 spots tall. Each square can have a blockade, a mine, a goal or be unoccupied. The agent can't enter a square with a blockade – if it tries to, it goes nowhere. If the agent lands on a goal or a mine state the game ends. If the state is the goal the agent gets a reward of 10, and if it is a mine, it gets a reward of -100 (all other states have 0 reward). In each state the agent has 4 possible actions – North, South, East or West. Figure 1 shows an example of an optimal policy.

Figure 1

Note that this world is slightly smaller than the one we will be using. The *'s represent blockades, the M's represent mines and the G represents the goal state. The arrows represent which action the agent should take in each state. In this example, actions are deterministic – i.e. if the agent tries to go East, it will always go East (unless the state it would end up in contains a blockade). The agent always starts in the upper left (state 1,1), so this is an optimal policy. Note that due to the way that policies are evaluated, the agent never learns what to do in certain states (i.e. states 1,10 and 2,10). In these cases the policy is not optimal, but it doesn't matter because the agent will never end up in these states if it starts in state 1,1 and follows this policy.

The version of the Mines world we use is slightly different in both its layout and its function. Figure 2 shows the version of the Mines world that we will use.

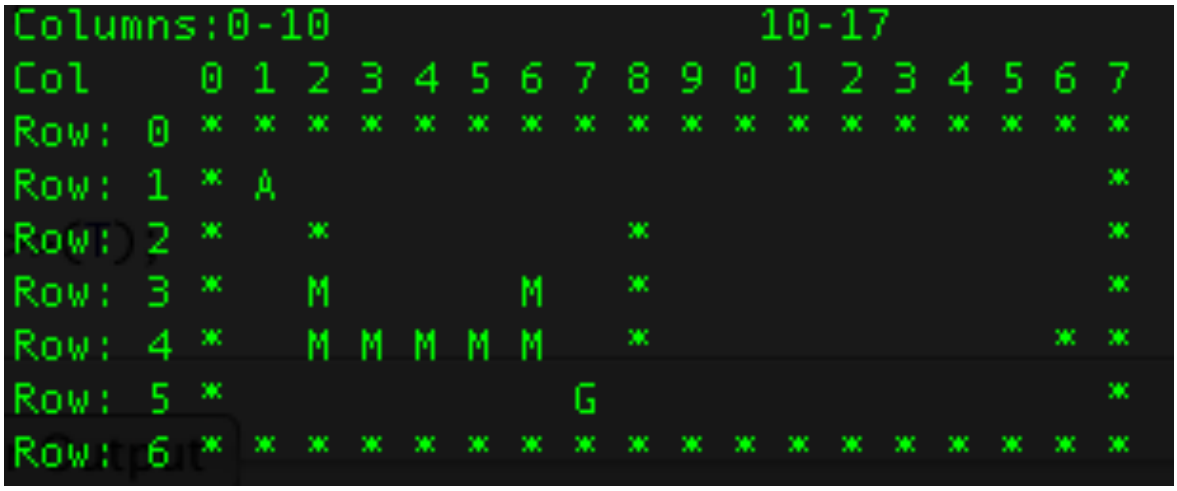

Figure 2

In this figure, the A indicates the position of the agent (in its starting position). We use a discount factor of .80. We also complicate the world by adding an element of randomness to the agent's movement. When the agent attempts to move in some direction, it will move in that direction with probability .8 and move in one of the "adjacent" directions with probability .1. So if the agent takes the action "move North" in state 2,4 it will end up in state 1,4 (its intended destination) with probability .8, state 2,3 with probability .1 and state 2,5 with probability .1. Note that there are (at least) three basic strategies that the agent might take, as illustrated in figure 3.

Figure 3 – the "bad" path is in red, the "middle" path is in yellow and the optimal path is in green

The "bad" path (shown in red) is very short and direct, but it brings the agent dangerously close to the mines. The "middle" path (shown in yellow) is the same length as the "bad" path, but is superior because it avoids going near the mines until the very end. The optimal path (shown in green) is able to avoid the mines entirely, although it is slightly longer than the other two.

We first apply the MWAL algorithm using a fairly simple set of three features. Feature 0 is based on the goal: it is 1 if the agent is on the goal state and 0 otherwise. Feature 1 is based on the mine locations: it is -1 if the agent is on a mine and 0 otherwise. Feature 3 is also based on mine location: it is -1 if the agent is adjacent to a mine (i.e. if there is a mine above, below or next to the agent, but not if there is a mine diagonally across from the agent) and 0 otherwise. Note that this third feature is not really part of the true reward function (i.e. its weight would be 0). However it was included because it captures the idea that the agent should avoid states where it

might accidentally end up on a mine in the next time step and because it improved the performance of the algorithm.

Step 7 from the MWAL algorithm (computing an optimal policy using the current reward weights) was implemented using simple value iteration. Since the Mines world has so few states, it is easy to iterate over all possible states to compute an optimal policy.

In order to explore the performance of the MWAL algorithm given different expert policies, we test it with three different sets of trajectories corresponding to the paths described above. The first expert (call it the bad expert) always takes the red path, the second expert always takes the middle path and the third always takes the optimal path. Figure 4 shows how the algorithm does with each of the three different experts.

	Bad	Middle	Optimal
Expert Average Reward	-78.0	-12.0	10.0
MWAL Average Reward	-27.0	4.5	8.79

Figure 4 – The reward for the MWAL algorithm was averaged over 1000 trials

The MWAL algorithm was most impressive when given a bad expert. Even though the expert usually just ended up landing on a mine, the agent managed to get

a reasonable reward. Figure 5 shows a typical policy that the MWAL algorithm came up with for the bad expert.

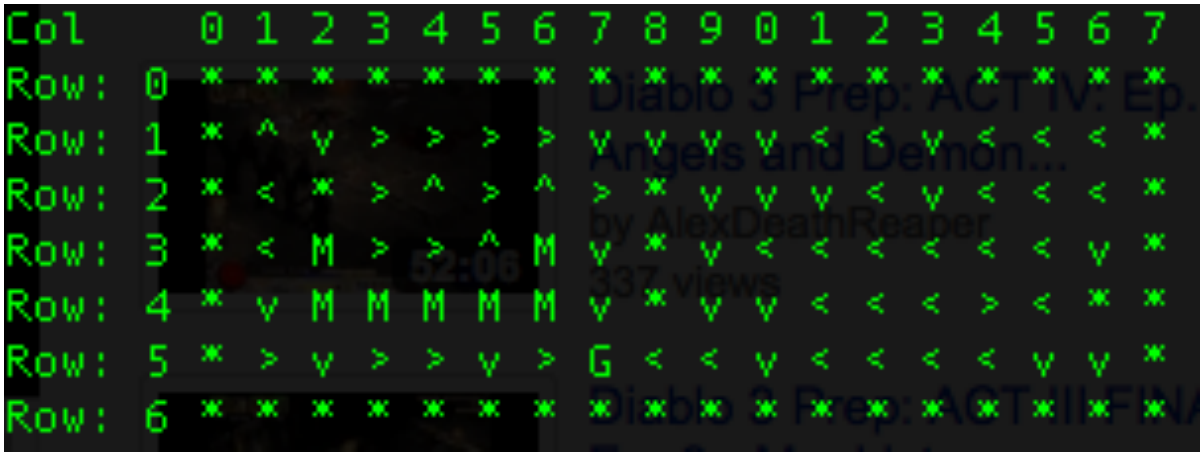

Figure 5

Note that the agent will try to take the middle path, even though the expert took the bad path. In fact, in this particular policy, the agent will never end up on the bad path because it tries to go North from the starting state. The agent will still occasionally wander into a mine, but it does a much better job of avoiding mines than the expert.

The performance of the MWAL algorithm when given an expert who took the middle path was similarly impressive. As figure 4 indicates, the agent still performed significantly better than the expert on average. Figure 6 shows a typical policy created by the algorithm.

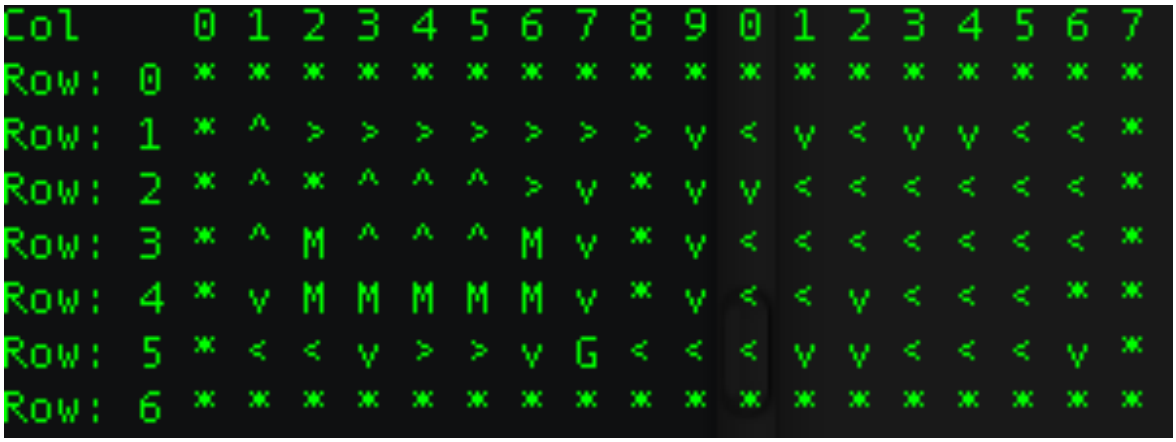

Figure 6

The policy is largely the same as the one created from the bad expert, but with several important differences. Firstly, the agent makes a bigger effort to avoid the bad path (i.e. in state 2,1 it will go North rather than West). Secondly, the agent makes more of an effort to avoid the mines on squares (4,3), (4,4) and (4,5). Unlike the policy in figure 5, the agent will get away from the mines as fast as possible. The most important difference, however, is that the agent will attempt to take the optimal path. Of course, the agent will still end up taking the middle path if it ends up going South when it tries to go East in state (1,7), but most of the time it will end up taking the optimal path and avoiding the mines completely. Note that the MWAL algorithm produces *T* different policies, some of which don't share all of the above mentioned features. But as the average reward in figure 4 shows, the agent is nevertheless fairly effective at avoiding the mines and getting to the goal state safely.

The agent's performance was slightly less impressive when given an optimal expert. Figure 7 shows a typical policy created by the MWAL algorithm in this case.

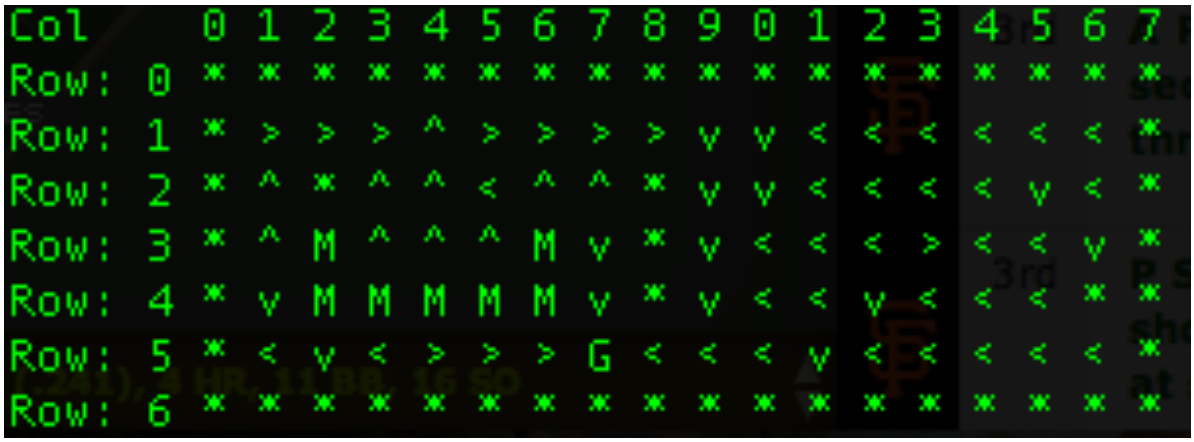

Figure 7

This policy is again similar to the one produced by the agent when given the middle expert, but with several important differences. The agent's actions in states (3,3), (3,4) and (3,5) are the same as before, but it does a slightly better job of avoiding the mine is square (3,6) (i.e. it goes North in square 2,6 rather than East). Also, the agent will go North in square (2,7) and thus never takes the middle path. This policy still isn't optimal (i.e. the agent would be better off going North in square 2,5), but it does a much better job of avoiding mines than the previous two policies shown. Despite this improved performance, however, it is still worth pointing out that the agent fails to perform completely optimally, as the average reward in figure 4 shows.

Note that the values of the features we used were somewhat extreme – they were always -1, 0 or +1. As figure 4 shows, this produced reasonable results. However, given the algorithm's heavy reliance on the values of these features, one might wonder what effect changing them would have. Of course it is the algorithm's job to guess the weights for each feature, but it is natural to wonder whether changing their values to closer match the true reward function has any effect. To

test this, we ran the same experiments, but with modified feature values. Feature 0 (whether this is a goal state) was changed to be .1 on the goal state and 0 otherwise while feature 1 (whether this is a mine state) was left the same (i.e. still -1 for a mine and 0 otherwise). This more closely reflects the true reward function where the reward for landing on a mine is 10 times larger in magnitude than the reward for the goal state. Feature 2 (whether there is a mine in an adjacent square) was changed to be -.2 if there is a mine nearby and 0 otherwise. This reflects the fact that being next to a mine is bad, but not nearly as bad as being on top of a mine. The results of this experiment (again in the form of average rewards) are shown in figure 8.

	Bad	Middle	Optimal
Expert average reward	-78.0	-12.0	10.0
MWAL average reward	8.68	7.69	8.35

Figure 8

We will not give a detailed description of the policies produced by this change other than to note that they are very similar to the one shown in figure 7 (which makes sense since they have about the same average reward). The main thing to note about these results is that they are all in about the same range. Interestingly enough, the adjustment we made to the values of the features was enough to allow the agent to perform near optimally even when given a very bad

expert. Given that the algorithm is responsible for guessing the weights of the features it is somewhat surprising that changing their magnitude has such a dramatic effect. On the other hand, it makes intuitive sense that giving the algorithm more information about the true reward function (in the form of the different feature magnitudes) would allow the agent to achieve better performance. Even though the agent is still not able to achieve optimal performance, it seems that changing the values of the features makes the agent much less sensitive to the performance of the expert.

It is also worth noting that there is some variation in the average reward, and the algorithm with the bad expert actually has the highest average reward. This is somewhat surprising, but the difference in average rewards is not huge and a larger number of trials might even out such differences.

Tetris

The game of Tetris was invented by Alexey Pajitov in 1984 and has been the subject of both theoretical [4] and practical [5] [6] [7] [8] research. The game of Tetris is played on a 10 x 20 grid where each square can be either occupied or empty. The game proceeds by presenting one of seven possible pieces (each formed from four contigous blocks in various patterns), which then starts to "fall" down the board. The player can then rotate and translate the piece as he sees fit until it either hits the bottom of the board or lands on an occupied block. The player scores points by completing rows, which are then removed, and higher rows (if present) moved down to fill the gap. In our formulation, the player is rewarded a score equal to $2^{i - 1}$

when *i* rows are cleared. The game continues until a piece lands with one or more of its blocks outside of the game area. As in the Mines world, the implementation of the game of Tetris was obtained from the rl-library project [3]. See Breukelaar, Demaine, Hohenberger, Hoogeboom, Kosters and Liben-Nowell [4] for a rigorous definition of Tetris.

Breukelaar, Demaine, Hohenberger, Hoogeboom, Kosters and Liben-Nowell [4] showed that the offline version of Tetris (in which the entire sequence of pieces is known) is NP-complete. Various techniques have been applied to the problem of playing Tetris, including genetic algorithms [5] and optimization algorithms [8]. Carr [9] provides a good overview of the various attempts that have been made to apply reinforcement learning to Tetris. Zhang, Cai and Nebel [6] have applied imitation learning (which is similar in spirit to apprenticeship learning) to a modified, competitive form of Tetris. But to our knowledge, no one has applied apprenticeship learning to playing Tetris.

We translate Tetris into an MDP in a straightforward way. The current state is just the current layout of the board (including the currently falling block). There are 6 possible actions: move the piece left by one square, move it right, rotate clockwise, rotate counter-clockwise, none and fall (makes the piece fall until it hits an occupied square). The piece falls one square each time an action is taken (unless the action is to fall). Note that unlike most real implementations of Tetris, the current state does not include the next available piece. Actions are completely deterministic; the block always does what the agent expects it to do. The next piece

is always chosen uniformly at random from the seven possible pieces. The discount factor is .90.

Since Tetris is played on a 10 x 20 board and each square can be either empty or occupied, there are on the order of 2^{200} possible states. This presents problems for step 7 of the MWAL algorithm (computing an optimal policy). It would be infeasible to iterate over all possible states, let alone store such a policy. Clearly some sort of approximation is needed to deal with such a huge state space. We attempt to solve this problem by using a simple form of function approximation. Function approximation can refer to any means by which the utility of a state is represented without using a lookup-table (as in value iteration). We use a relatively simple approach described by Russell and Norvig [10] that is based on a linear combination of features. This is similar to how we approximate the reward function, and indeed we use the exact same features in each case. We define the estimated utility of a state *s* as $\hat{U}(s) = w_0 f_0(s) + \ldots + w_k f_k(s)$ where the w_i are weights (which, despite the somewhat confusing terminology, are entirely separate from the reward weights) and the $f_i(s)$ are features. Once the weights are known, it is easy to compute the optimal action from any state by simply choosing the action that results in the state with highest utility (since actions are deterministic, there is no probability involved).

In order to actually compute a policy, we used a form of TD learning described by Russell and Norvig [10]. In order to learn a policy, the algorithm chooses some fixed policy and runs it for many iterations. After each step, the weights used to estimate the utility are updated according to the following equation:

$w_i = w_i + \alpha(R(s) + \gamma\hat{U}(s') - \hat{U}(s))$. Here s' is the destination state and s is the current state, and α is the learning rate (set to .1). $R(s)$ is computed using the current reward weights and $\hat{U}(s)$ is computed as above. The resulting estimate of the utility is then used as the basis for a policy (that is, the policy just chooses the neighbor state with the highest utility).

In order to approximate the utility of a state, 3 different features were used. The first feature is 1 if any row is complete and 0 otherwise. The second feature is the average row size (scaled to be between 0 and 1). The third feature is based on the height of the tallest column. It is close to 1 if the highest column is not very tall and close to 0 when the highest column is near the top of the board. These features are very simple and are almost certainly not capable of capturing all of the intricacies of Tetris, but they suffice to illustrate the performance of the MWAL algorithm in some simple experiments. Given a relatively bad expert, the feature expectations vector for a sample run of the algorithm is given in figure 9.

	Rows cleared	Average row size	Tallest column
Expert	0.0	1.449	2.639
MWAL algorithm	8.69e-11	2.223	6.824

Figure 9

As this table indicates, the MWAL algorithm is able to outperform the expert in each of the features (even managing to clear an occasional row, unlike the expert). The algorithm was also given input from a more capable expert, and the results are shown in figure 10.

	Rows cleared	Average row size	Tallest column
Expert	8.24	1.044	4.45253
MWAL algorithm	0.0	2.351	6.91

Figure 10

The MWAL algorithm manages to do just slightly better than before. However, this seems to be not because of an inherent flaw in the algorithm, but rather because of the overly simplified nature of the utility features. That is, the features do such a poor job of describing the utility of a given state that the algorithm has a hard time learning how to actually play Tetris. For example, the "tallest column" feature can be maximized (which corresponds to the tallest column being very low) by playing good Tetris (i.e. completing many rows and thus lowering the height of the tallest column). But without sufficient guidance (in the way of other features), the algorithm will have a hard time finding such a strategy. The state space of Tetris is so huge that many different features will be required to make the (possibly very long) sequence of moves necessary for such a strategy appear attractive to the algorithm.

Conclusion/Future Work

The performance of the MWAL algorithm in the Mines world is encouraging. It lived up to its promise and was able to outperform both of the first two experts (the "bad" and "middle" ones). Despite the fact that it did not fare quite as well given an optimal expert, the algorithm certainly seems to work well in small state

spaces. The algorithm also showed a somewhat surprising dependence on the magnitude of the reward features. The agent outperformed the expert regardless of the magnitude of the features, but changing them to more accurately reflect the true reward function allowed the agent to achieve much better performance in the presence of a bad expert.

The results for Tetris, however, were less impressive. While the algorithm did a good job of outperforming a bad expert, it was less successful when given a more advanced expert. Nevertheless, the success of the MWAL algorithm in the Mines environment suggests that the problem may be not with the algorithm itself, but rather with the function approximation used in step 7. Certainly the features used in our function approximation are extremely minimal and it is not hard to imagine that a more sophisticated set of features might result in improved performance. Thiery and Scherrer [7] provide a good overview of features that others have used in attempting to learn to play Tetris, many of which would likely be useful to our approach. While the rows completed feature is good for capturing the true reward function, it is almost certainly not enough to capture the intricacies of the utility function for each of the 2^{200} or so possible states. In order to for the algorithm to really learn to play Tetris we would likely need more features that capture the utility of states leading up to completing a row. Such features (we can think of them as "intermediate" features because they describe the states leading up to a reward state) would help the algorithm to reach good states and avoid bad ones. The average row size feature is an example of this kind of "intermediate" feature, but we would likely need many more to play good Tetris. As another

example of a good intermediate feature, Thiery and Scherrer [7] describe a "holes" feature, which is just the number of holes (open blocks surrounded by occupied blocks) on the board. Holes prevent rows from being completed and should thus be avoided. Even the maximum column height, already an intermediate feature, might benefit from another intermediate feature. Thiery and Scherrer [7] also describe a "landing height" feature, which is based on the height at which the last piece landed. In our set of features, the agent is motivated to keep the maximum column height low by the corresponding feature, but it is not directly encouraged to drop a piece as far as possible, which the "landing height" feature would accomplish. It is also worth pointing out that most of the states in a given run of Tetris will have a piece that is currently falling, but none of our features capture any information about the falling piece specifically (such as position relative to good landing sites, remaining squares to fall, etc.). Finally, we don't include any information about the next available piece even though this information is provided by the Tetris implementation. One might imagine using a different strategy for the current piece depending on the next available piece.

As the results in the Mines world showed, changing the magnitude of the features has the potential to greatly improve the agent's performance, especially when provided with a suboptimal expert. In Tetris the reward function is even more complicated and has a correspondingly larger potential for tweaking the feature magnitudes. Our features only varied in magnitude in a simple, linear fashion, and one could spend a great amount of time experimenting with different feature magnitudes to better reflect the true reward function.

Of course, a linear combination of features is about as simple as function approximation can get, and a more sophisticated form of function approximation (such as a neural network) might better capture the intricacies of the true utility function. Policy search is another technique for learning policies in very large MDPs, and might be a better fit than function approximation for our purposes.

Acknowledgements

Many thanks go to my advisor Prof. Rob Schapire who both introduced me to the field of artificial intelligence and provided me with guidance over the course of the semester. Without his support and advice I wouldn't even have known where to start. Thanks also go to my girlfriend and family for supporting me through this entire process.

References

[1] P. Abbeel, A. Ng (2004). Apprenticeship learning via Inverse Reinforcement Learning. *ICML* **21.**

[2] U. Syed, R. Schapire (2008). A Game-Theoretic Approach to Apprenticeship Learning. *Advances in Neural Information Processing Systems* **20.**

[3] RL-Library project. http://library.rl-community.org/

[4] R. Breukelaar, E. D. Demaine, S. Hohenberger, H. J. Hoogeboom, W. A. Kosters, and D. Liben-Nowell (2004). Tetris is hard, even to approximate. *International Journal of Computational Geometry and its Applications* **14.**

[5] N. Bohm, G. Kokai, S. Mandl (2005). An evolutionary approach to Tetris. *Proceedings of the Sixth metaheuristics international conference*

[6] D. Zhang, Z. Cai, B. Nebel (2010). Playing Tetris using learning by imitation. In *GAMEON*, pp. 23-27

[7] C. Thiery, B. Scherrer (2010). Building controllers for Tetris. *International computer games association journal* March edition.

[8] I. Szita, A. Lorincz (2006). Learning Tetris using the noisy cross-entropy method. *Neural computation* **18.**

[9] D. Carr (2005). Applying reinforcement learning to Tetris. *Bachelor's thesis, Computer Science Department, Rhodes University*.

[10] S. Russell, P. Norvig (2010). Artificial Intelligence: a Modern Approach (Third edition). *Prentice Hall* pp. 845-848.

This paper represents my own work in accordance with University regulations.

Robert Timpe