Reinforcement Learning for Game Development

Mike Riches, Ziniu (Aaron) Yang, Ruochen Hua

# 1 Introduction

Reinforcement learning is a well-known AI technique that trains an AI in a fashion similar to classic learning by applying punishment and rewards. Reinforcement learning can make games more fun to play – you can use it to teach an AI agent the best way to play a game, and you can use it to respond to human play styles in real time.

Reinforcement learning (RL) simulates the consequences of actions; in other words, it allows the agent to learn by cause and effect. It will remember mistakes, and it can be trained to recognize successful actions it should take depending on the state of the game. Reinforcement learning does this by storing, updating, and using a numerical value representing the success or failure of a particular action. Initially, the value of each action is zero – you have to train the AI using offline learning before it will seem “intelligent” enough to play a game.

Reinforcement learning can be tricky to implement in games – and difficult to see the benefit at first. But the benefits, if you can do it correctly, are potentially great. You can program an AI that appears to respond intelligently out-of-the-box – and then have it continue learning against the player. This gives an RL agent the potential to respond to player actions differently if given the same circumstances again in the future. This can provide a more varied and enjoyable gameplay experience. Properly applied, it could even allow you to dial up the difficulty in a way that is challenging to the player, but also seems natural and effective.

## 1.1 Implementation considerations

To implement RL, you need four major components: a learner, a game state representation (world), a policy, and a player. The learner trains against simulated or real gameplay situations and it updates the policy to reflect the consequences of its actions. The RL player inspects the world state and then chooses an action by reading from the policy. (It’s often convenient to implement the player and the learner as the same class, but you don’t always want them to be the same instance, for example with simultaneous learning and playing.) The world state simply tracks the current state of the game world, meaning that it tracks the specific parameters you defined when you programmed or configured your RL implementation. The world also recognizes state transitions, which all RL algorithms use to apply punishment and reward values.

## 1.2 Overcoming challenges with reinforcement learning

Reinforcement learning requires many iterations (called “epochs” by convention) in order for the agent to become effective at playing the game. Most game implementations of RL will need to include pre-baked RL agents to present a sufficient level of capability out-of-the-box.

Representing game state is another challenge. You must choose the game state parameters that your AI will be aware of. Choose wisely - leaving out the wrong parameter can cause the AI to seem unintelligent, whereas including too much information grows the complexity of your policy. Too much complexity leads to poor performance.

Here are just a few examples of game state parameters you might track:

* Player position and agent position on a grid
* Position of a racecar on a track - and its damage state
* How many zombies are left, and player health

# 2 Overview of Reinforcement Learning algorithms

RL algorithms are sufficiently complex as to warrant an overview explanation before discussing implementation. Two popular algorithms are used with reinforcement learning: Q-learning and SARSA, which are explained later in this article. Both use a similar overall workflow, but with different equations for computing the policy updates and slightly different learning goals.

## 2.1 Training sequence

Before we can define the training sequence, we need to define some common terms:

**Q-value:** The potential value of taking a particular action, given a particular state of the game. The higher the Q-value, the better the action; the more negative, the worse it is.

**Epoch:** One training iteration. An epoch can take many for-loops to complete. The inner for-loops are called iterations, and in that way we disambiguate.

**Action:** Thechoice theRL agent makes and that it learns from.

The overall reinforcement learning workflow occurs like so:

1. Initialize the Q-values to an arbitrary (but consistent) value. (We used zero.)
2. Perform as many epochs as specified by the operator, or try to meet a desired condition.
3. Store the resulting policy for use later (either for gameplay, training, or both).

One iteration in an epoch occurs like this:

1. Choose the best action, which involves consulting the policy.
2. Take that action.
3. Observe the reward or punishment value, if any.
4. Update the Q-value for that action in that game state using the observed reward (or punishment), and using the formula for the learning algorithm of your choice.
5. Update the game state, so we can use it in the next iteration.
6. Check for “game end” conditions. If game ends, break and end the epoch.
7. On the next turn, start again at Step 1.

Exactly what you choose to have it do for step 4 will determine how your agent learns from its successes and failures, and in turn, how it behaves as a result of what it learns. Next, we will look at specific reinforcement learning techniques you can apply here, including their relative strengths and weaknesses.

## 2.2 Q-learning algorithm

Q-learning is advantageous when your agent starts from a clean slate. It can learn even if you give it a high amount of randomness in the actions the agent tries to take initially. Q-learning uses a learning rate constant, epsilon, which determines how much the Q-value is updated for a given reward or punishment. It also uses a constant gamma, called a discount factor, which is used to weight current rewards against future rewards.

Let *s* be a state, let *a* be an action, let *r* be a reward value, and let *Q(s, a)* be the q-value for a given state, action pair. The equation for Q-learning is:

,

where *s’* is the new state.

## 2.3 SARSA algorithm

SARSA builds on Q-learning by considering not just the current action, but also the best action to take next after that. Q-values are considered to be a function of two state-pairs: *(s, a)* and *(s’, a’)*, and the reward value *r*. Thus, we have *Q(s, a, r, s’, a’)*, and this is where the name SARSA comes from.

Given the parameters stated above, the equation for SARSA is:

where *s’* is the new state and *a’* is the new action.

## 2.4 Action selection policies

One great point of flexibility in Q-learning is the ability to define action selection policies that lead to meaningful gameplay by your agent. You can experiment with always selecting the highest-scored action (the “greedy” choice), or select a different action some other portion of the time. One popular solution is to pick something different some percentage of the time by introducing randomness, which can help solve issues with overfitting and less-than-optimal-solutions. How you select a different action is up to you, for example: it can be completely random, you can choose the second-best action, or you could even train it to infrequently select a mediocre or bad action. Choosing a bad action could be done to make sure that is still a bad choice, for example, when playing against a new player.

# 3 Reinforcement learning example: Cat-and-Mouse

For our game application research project, we implemented both Q-learning and SARSA algorithms. We felt this to be a representative set of options to demonstrate how reinforcement learning could work in a real-time game environment. In our demo, the gameplay environment consists of a grid map, a cat (simple preprogrammed AI agent), a mouse (RL agent), walls, and the cheese (goal). Here are the game rules:

* The cat tries to catch the mouse.
* The mouse tries to get the cheese.
* The cat gets one point when getting the mouse. The mouse scores one point by getting the cheese.
* If the mouse gets the cheese, the cheese will be replaced randomly elsewhere on the map.
* If the cat catches the mouse, the game is over and a new epoch is started.
* Agents can’t move through walls, but can move diagonally across them.
* The edges of the map are treated like walls.

There are two types of mouse: a greedy mouse and a smart mouse. The greedy mouse doesn’t use RL to choose its action – it is preprogrammed to always move toward the cheese. (Note: This is not related to the “greedy” type of action choice. The greedy mouse is implemented simply as a point of comparison.) The smart mouse will use RL data to help itself make decisions, once we train it.

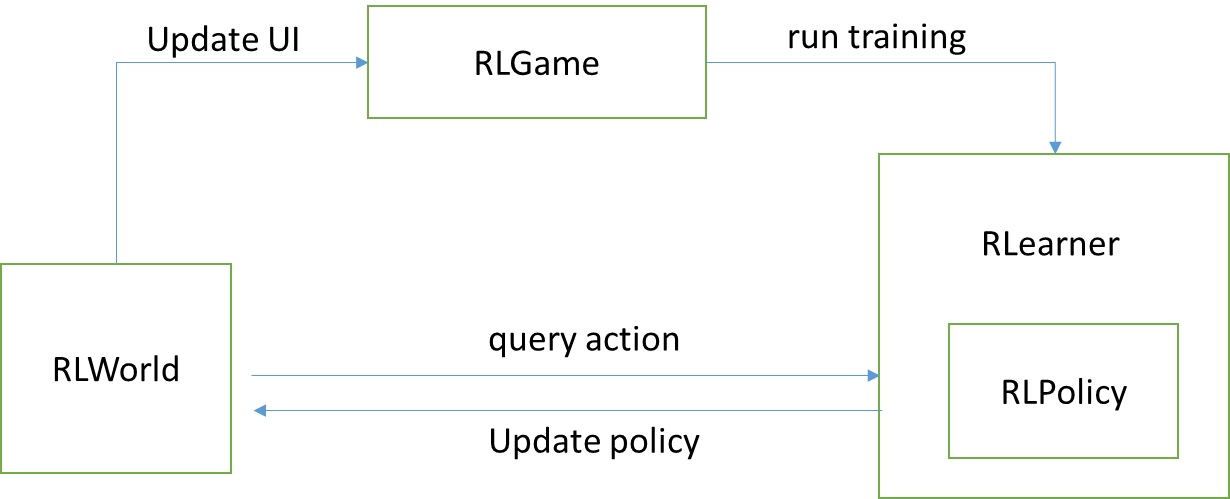
There are two parts in this demo; the first part is training. We give our reinforcement learning implementation some parameters, and it will run that specific RL algorithm setup for a number of epochs and record learning data. The second part is playing; after training, the AI agent will be called upon to play the game by using recorded learning data (policy) to move the mouse. UI updates are possible in both states. Training is displayed very quickly for the sake of brevity, and playing has the option to occur in real-time so that you can observe the choices being made. Both speeds can be adjusted.

Reward is calculated in this way: by default, when the mouse get the cheese, the reward is 50. Also by default, when the mouse is caught by the cat, the “reward” is -100 – and in this case, we say the punishment value is 100.

## 3.1 Implementing Cat-and-Mouse

We used Steve Rabin’s SML learning framework (which is built on Microsoft’s “Tiny” DirectX animation sample) and C++ to implement this demo program. At a high level, our reinforcement learning program consists of five modules: RLearner, RLPolicy, RLWorld, RLGame and RLAgent. The RLearner and RLPolicy classes are responsible for running the RL algorithm and storing learned data. RLWorld maintains some state information for the game world. RLGame is a “straw-man” game state manager, responsible for controlling the game loop and feeding parameters selected via the UI to the reinforcement learning algorithm. RLAgent is a finite state machine (FSM) agent class that moves cat and mouse characters around on the map, allowing us to watch the action unfold.

The following diagram provides an overview of our demo architecture:



Next, we will explain each module in detail.

**RLPolicy:** This class is used to store learning data. The data consists of a lot of pairs. Each pair consists of a particular game state, with a set of Q-values. “Game state” in our example implementation means the position of the mouse, the position of the cat, and the position of the cheese. Each position is an x, y value pair, for a total of 6 floats. We use std::vector<float> to represent a single game state. Each state is mapped to 8 mouse actions – one of 8 directions to move – and each action has a Q-value representing how good the action is. We also use a std::vector<float> to represent each group of Q-values. We chose to use std::map to represent game state/Q-value set pairs. The map data structure is advantageous – it allows us to store only non-zero data, meaning that we save lots of memory compared to using a fixed-size n-dimensional array.

**RLearner:** This class implements the major Q-learning functionality. It is built to run one of two reinforcement learning algorithms, Q-learning or SARSA. When it calls the member function RunEpoch(), it will query the m\_policy instance for the best action to take – take that action – and then update m\_policy with the result.

**RLWorld:** This class maintains the state of the game world. The following RLWorld APIs are used by the RLearner:

* vector<int>& GetNextState(int action, bool update)  
  Takes an action, then updates the state depending on the result. Returns the current game state.
* bool EndState()  
  Tests whether or not the game is over.
* float GetReward()  
  Calculates, and returns, the reward value of current game state.

**RLGame:** This class has a FSM. It works as a game state manager; it controls the training and playing process, it allows you to provide input (such as adjusting the process speed), it handles global messages, it prevents inappropriate state changes, and it updates the UI.

**RLAgent:** This is a simple FSM used to represent an agent’s presence in the game. The cat and the mouse agents each run a separate instance of this class. It has simple behaviors like walk, run, add waypoint, or teleport to a specific position.

## 3.2 Experimenting with the Cat-and-Mouse implementation

In this section, we will go over several experiments we ran using our reinforcement learning example. These experiments show how reinforcement learning works and they are good examples of how certain factors can affect the result. Since the end state of a game is the cat catching the mouse, the cat’s score is always equal to the number of epochs. The winning rate is equal to the mouse score over the total score; the total score is the sum of the mouse score and the cat score. The algorithm we used for these experiments is Q-learning.

In our results ‘playing’ is used to indicate gameplay mode, where the policy is used but not updated. This is used to evaluate the agent’s learning effectiveness.

### Experiment 1: Greedy mouse vs. untrained smart mouse.

In this experiment, we ran ‘playing’ 5000 times using the greedy mouse, and then ran ‘playing’ 5000 more times with a completely untrained RL agent mouse.

|  |  |  |
| --- | --- | --- |
|  | Score | Winning rate % |
| Greedy mouse | 3856 | 43.87 |
| Untrained smart mouse | 614 | 10.93 |

The results show that a pre-programmed “greedy mouse” AI has much better performance is than an untrained smart mouse, which can only move randomly.

### Experiment 2: Different amounts of training epochs

In this experiment, we trained the agent 0, 5000, 10000, and 50000 epochs. After each training phase, we ran ‘playing’ 5000 times, and recorded the performance of the RL agent mouse given that amount of training.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epochs | 0 | 5000 | 10000 | 50000 |
| Score | 624 | 822 | 1193 | 6227 |
| Wining rate % | 11.09 | 14.11 | 19.26 | 55.46 |

So we see that as the training time increases, the mouse performance continues to get better and better. When trained 10,000 times, the RL agent is still worse than a “greedy mouse,” but by the time we got to 50000 epochs the RL agent’s winning rate overpassed ‘greedy mouse’ by about 11%.

### Experiment 3: Different maps

In this experiment, we see how the RL agent performs on different types of terrain. In each of the following three maps, we ran ‘training’ for 20000 times and then ran ‘playing’ for 5000 times, then recorded the RL agent’s performance:

* Map 1 is a large open area with only a few walls.
* Map 2 looks like a box with several exits.
* Map 3 consists of a few connected corridors which are parallel to one other.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Map 1 | Map 2 | Map3 |
| Score | 1158 | 2147 | 43619 |
| Wining rate% | 6.27 | 30.04 | 89.71 |

Here, we got some interesting data. The agent performance is entirely different in these three maps, even though we set all learning parameters the same. So why is this the case?

To figure this out, we observed the agent’s behavior directly. We saw that in Map 1, which is very open terrain, it’s very easy for the cat to follow the mouse and finally corner it at the edge of the map. Without adequate terrain features, the mouse has nothing it can learn to use when avoiding the mouse. Reinforcement learning doesn’t do well in this open area.

But in Map3, which has tight corridors, our RL agent mouse’s performance is excellent. That’s because it’s easier for the mouse to run along the corridor, then use the transition to the next corridor to escape the cat. Another reason is because the map is repetitive; a lot of game states will appear over and over again on this map, which is a perfect environment for RL to update its policy and use reinforcement to learn efficiently.

### Experiment 4: Punishment and reward values

In this experiment we ran the learning algorithm in the same circumstances several times, changing only the punishment and reward values, and observed its learning behavior. We used Map 3 for this experiment. First, we set the punishment to 100 and the reward to 50 (our defaults), trained 20000 times, and finally we ran ‘playing’ 5000 times. Then we tried the same process with a punishment value of 10 instead of 100.

|  |  |  |
| --- | --- | --- |
|  | Punish:100, Reward:50 | Punish:10, Reward:50 |
| Score | 77710 | 41415 |
| Wining rate% | 93.98 | 89.22 |

We got some very interesting results in this experiment. Notice the behavior of the mouse. When the punishment is set to 100, the RL agent mouse considers that avoiding the cat is more important than getting the cheese. So the mouse tends to get away from cat, and won’t take risks to get the cheese when the cat is around. But when the punishment is reduced to 10, getting the cheese becomes more attractive, and our RL agent mouse will try to go for the cheese even if it will be caught by the cat. We found that higher punishment causes the mouse to stay alive longer, allowing it to consistently get a bigger score and achieve a higher winning rate over time.

## 3.3 Other factors we considered for Reinforcement Learning

There are other factors that can influence effective training and RL agent performance.

1. -value in Q-learning: Determines how quickly the learning can occur. When we turn down this value, it takes more instances of the same consequence to update the Q-value to a similar level.
2. -value in Q-learning: Determines the importance of future reward compared to immediate reward.
3. Probability of exploration: We implemented a random parameter to make the agent sometimes take random action instead of querying learning data. So our “smart” mouse does not use an entirely “greedy” policy.

## 3.4 Conclusion

In these experiments, we learned some interesting things. First, we demonstrated how Q-learning can indeed improve the performance of an RL agent. The effectiveness of the policy gets much stronger when more training is done. Theoretically, when trained enough by Q-learning, an agent can always perform optimal actions. We also saw that the game environment can have a big impact on the effectiveness of learning. We investigated the impact of weighting the punishment versus the reward, and discussed how Q-learning parameters affect the agent learning process.

# 4 Current applications of reinforcement learning in the industry

It is important to acknowledge specific examples of how reinforcement learning has been successfully applied to shipped game titles, and also discuss some examples of how the industry is currently exploring reinforcement learning as a tool for improving gameplay.

## 4.1 Board games

Reinforcement learning has a long history of use in board game titles. In fact, the basic techniques for solving reinforcement learning, which is the prediction method called temporal difference learning (TD), was invented for the purpose of making a program that has the ability to learn how to play checkers. With the creation of TD algorithms, one of the first successful implementation of it became a backgammon learning program call TD-gammon, developed in 1992 by Gerald Tesauro. The success of it resulted in many research projects in this area, and reinforcement learning then saw use in other kinds of board games such as chess and go.

One of the advantages of using reinforcement learning as the AI of the board games (instead of traditional hand-coded AI) is its simplicity of implementation. If we need to make an AI for a board game, we need to know the rules of the game, and then we also need to supervise the AI agent while it plays the game give it predetermined moves we think it should take, given particular game states. For a simple board game like Tic-Tac-Toe, a procedural or option-based implementation does not seem that bad because there are not that many different situations you can encounter in the game. It is also easy to find patterns in this game. But if we want to make an AI for the board game chess, making a hand-coded AI is a nightmare because the number of situations we can encounter is exponentially greater, and because the rules are far more complicated. However, with reinforcement learning, we only need to figure out a way to program in the rules of the game – and then add reward functions. Reward functions may be the hardest part of the whole program, but it is much easier than hand-coded AI and quite tenable. After that, we just need to give some time, and opponents, for the AI to train itself.

Another issue with hand-coded AI in board games is that the techniques are not adaptable. If we are able to find one tactic to beat the AI, most likely we can use that tactic every time and keep defeating the AI every time. Also, a hand-coded AI may act weird in situations the game creator did not consider – this becomes common for a complex board game. It will act weird every single time because without a learning mechanism, the AI cannot change its tactics. It cannot learn from the games it played in the past. With reinforcement learning, the AI agents will be adaptable because they will record every game they played in the past and update their policy. Next time, the same tactic may not work because the AIs will change their preferred moves.

The third advantage to reinforcement learning is that the hand-coded AI is all your players will ever get after the program released, unless you (the developers) make patches for the program and update the AI. But if you’ve already shipped, it’s more likely the AI won’t get any improvement at all, or only minor improvement to fix specific bugs reported by players. It’s possible to program in a different selection of difficulty levels but it will still feel repetitive after enough playthroughs. With an AI that uses reinforcement learning it can feel more and more intelligent, it can be more and more challenging, and it will provide different game play experience in different matches.

## 4.2 Strategy games

Actually, strategy games and board games are pretty similar in some aspects, so reinforcement learning can also be applied in strategy games - and it can also provide new features and experiences.

Here is an example in Wargus. First, a description: Wargus is a Warcraft 2 mod that allows you to play the game in the Stratagus engine. (This mod is not a copy of the game, so if you want to try it out, you’ll need an original copy of Warcraft 2.) With Wargus, we can experiment and apply custom AI to Warcraft 2.

There is an application called CLASSQ-L that is able to play through a complete Warcraft 2 game, through Wargus, using the Q-Learning algorithm. Traditional implementations of reinforcement learning in strategy games usually track 3 major game state features using a 2-dimensional map, including:

* the number of resources and their (x, y) coordinates in a map
* information about each unit, including their class and their (x, y) coordinates
* information about each building and their (x, y) coordinates

This is the intuitive choice, however, the action and state space will be huge

considering the large amount of units and buildings that can appear in the same scenario. The units may need to execute commands such as:

* To construct a building, indicate the class of building and its coordinate
* To harvest a resource, indicate the class of resource and its coordinate
* Move a unit to an (x, y)-coordinate
* Attack an enemy unit or structure located in an (x, y)-coordinate

It’s easy to see how it would become pretty hard to use reinforcement learning to control the full scope of a real-time strategy game in this way. CLASSQ-L reduces the size of state-action space by having a separate Q-table for each class of unit, and then building and filtering useful state and action information that is customized for each class. Set up CLASSQ-L and train the AI for a little while, and the AI will learn how to deal with several tactics like Land-Attack, Soldier's Rush, Knight's Rush, and so on. It demonstrates that AI using reinforcement learning in strategy games has the ability to adapt to different tactics and find a solution to win the game. It is much better than just using a script AI or a simple hand-coded AI, which cannot respond when a player keeps using one tactic to win the game.

## 4.3 Fighting games

It may seem odd at first because fighting games are very different from board games and strategy games, but using reinforcement learning in fighting games is actually real and has been implemented effectively in a published game title called “Tao Feng: Fist of the Lotus.” The purpose of using reinforcement learning in a fighting game is to make the AI adapt to players' behaviors. It is a very good feature that can keep the game new and challenging.

In Tao Feng, the algorithm they choose is SARSA. (Recall that SARSA looks at action sequences, as we discussed in a previous section.) The developers' goal is to use reinforcement learning to develop a learning RL agent fighter. The fighter starts at a state that cannot even compete with a human beginner, but its skill becomes better and better when the game progresses.

To implement reinforcement learning in this game, they need to figure out how to choose features, actions and rewards that apply to this type of gameplay. There are mainly three groups of features: environment-related, opponent-related and agent-related. The environment-related feature is used for navigation in the arena, the opponent-related feature is used to react to opponent's action, and the agent-related feature will gives agent's actions continuity and consistency.

For the policy, instead of listing all the possible action sequences, they focus on atomic actions such as simple punches, kicks, and so on. With the atomic actions, they construct meta actions. Also, because timing is critical in fighting games, all the actions are associated with a time period – for example, “block25.”

For RL to work, the developers need some way to define a round in the real-time fighting game. Because the two fighters in the game are not acting in sync, they defined that if subsequent actions have been successfully selected, which means the previous action is completed, then a new training round is over. In this case, each round has a different duration; this is taken into account by considering the rate of the reward when learning.

In their tests the creators of Tao Feng found that by varying different reward functions, the agent can learn good policies, and some policies result in the agent displaying interesting actions. Also, using reinforcement learning helps identify weaknesses of the game engine and built-in AI because after training, the agent is able to find the “golden path” solution to defeating rule-based built-in AI quickly. It keeps winning the game in the same way over and over – just like experienced human players can do. So they also found out that reinforcement learning can be used as a good testing tool for other types of AI.

## 4.4 Racing games

For some games, it is possible that the methods to reach the goal is not readily known or available. For instance, in racing games, it is nearly impossible to know how to drive a car fast enough to give players a good challenge. Even if the game developers can find a professional driver to ask, or to record every action he/she takes, the scenarios in-game change rapidly and most of the time it won't be the same as the training session. In this case, the traditional methods will not work, and there is a better way to solve this problem using reinforcement learning.

To implement reinforcement learning in racing games, developers need to determine a way to represent reward functions, actions and other information that reinforcement learning needs to know about but with cars on a track. For example, the reward can be the time that the car takes to finish a race, or the time it takes to pass an area of the track; the actions can be similar to the actions in fighting games, using atomic actions like brake, accelerate, turn left/right, etc. Then we use these atomic actions to construct complicated actions like drifting. Also, like fighting games, every action should be associated with a time parameter. With racing games, you also have a degree parameter because in real life, drivers can turn the steering wheel to different angles at different rates. After all the pre-requisites are met, the next step is to train an AI driver on the track.

In the game Project Gotham Racing 3 (PGR3), reinforcement learning was successfully used for training its AI drivers. In their game, the developers of PGR3 used reinforcement learning to adjust AI drivers' steering behavior so that they can get the shortest possible lap time.

## 4.5 Future use of reinforcement learning in games

Reinforcement learning has a huge potential in game industry. Although it is not commonly used, there is a lot of research that show how reinforcement learning can associate with games in many ways and give the AI better performance.

Earlier we discussed how reinforcement learning can be applied to strategy games, for example, the CLASSQ-L application shows that AI trained with reinforcement learning can be used to make some simple decisions. However, there is a lot of research currently happening to show that the AI trained with reinforcement learning can not only make simple decisions like it does in Wargus, but it can also make high-level complicated decisions, like playing the game Civilization IV.

In Civilization IV, an AI must not only worry about resources, buildings, and units; it also needs to consider diplomacy with other AIs and/or players. A game of this complexity is difficult enough to give most human players pause, especially when they are first learning how to play. However, in [Christopher 10], the researchers find a way to use reinforcement learning to train the AI in order to make high-level decisions. During each game, the AI will observe current information, and make a decision according to the leader it represents. The whole process is repeated on subsequent steps up until the game ends. When the game is played again, the learning is continued and results are improved.

In comparison to earlier work, this research shows more of the potential reinforcement learning has for improving games. For example, unlike the planning approaches for strategy games, their approach tackles the complete problem rather than just resource gathering or attacking. Also, they use learning to adapt a policy based on the conditions of the current game and the strategy of their opponents. In contrast to game theoretic approaches, their method can learn to switch the strategy during the game rather than simply choosing a fixed policy for the whole game. Also, unlike previous learning methods, their research is about solving the more complex problem of playing a complete strategy game rather than optimizing individual parts of that game.

New research also shows how more types of games can involve reinforcement learning. For example, shooter games. There is now research showing that reinforcement learning can be used to learn the FPS bot behaviors of navigation and combat [Michelle 11], there is also research showing that reinforcement learning can be used to significantly generate adaptive team tactics [Megan], and it has been tested in the domination games in Unreal Tournament.

With all this research about using reinforcement learning in games, and research leading to implementations in successful titles, reinforcement learning technology is starting to see growth in its proven ability to improve game AIs. Reinforcement learning makes AI agents more dynamic, adaptable, and challenging, and it can be a good way to provide a better gaming experience to players.

# 5 References

[Imran 04] Imran Ghory. 2004. Reinforcement learning in board games. [www.cs.bris.ac.uk/Publications/Papers](http://www.cs.bris.ac.uk/Publications/Papers)

[Marco 07]Marco A. Wiering, Jan Peter Patist, and Henk Mannen. 2007. Learing to play board games using temporal difference methods. [www.ai.rug.nl/~mwiering/GROUP/ARTICLES/](http://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/)

[Ulit 12]Ulit Jaidee and Héctor Muñoz-Avila. 2012. CLASSQ-L: A Q-Learning Algorithm for Adversarial Real-Time Strategy Games. Artificial Intelligence in Adversarial Real-Time Games: Papers from the 2012 AIIDE Workshop AAAI Technical Report WS-12-15

[Thore]Thore Graepel, Ralf Herbrich, and Julian Gold. LEARNING TO FIGHT. Microsoft research.

[Christopher 10]Christopher Amato, Guy Shani. 2010. High-level Reinforcement Learning in Strategy Games.

[Michelle 11]Michelle McPartland and Marcus Gallagher. 2011. Reinforcement Learning in First Person Shooter Games. IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 3, NO. 1, MARCH 2011, Page 43

[Megan]Megan Smith, Stephen Lee-Urban and Héctor Muñoz-Avila. RETALIATE: Learning Winning Policies in First-Person Shooter Games