DESIGNING COMPETITIVE BOTS FOR A REAL TIME STRATEGY GAME USING Q-LEARNING

Ola Al Naameh

University of Bonn, Germany

ABSTRACT

Video games provide rich test-beds for artificial intelligence methods; Real-Time Strategy (RTS) games in particular. These games require severe time constraints on player actions and a strong demand for real–time artificial intelligence which enables the scientists to solve real–world decision tasks quickly and satisfactorily. Therefore, designing an autonomous agent that performs well in strategy games is a very important task. This paper presents a Q-reinforcement learning model that use neural networks to learn to play the *Planet Wars* game. We used a simple feedforward neural network to estimate a Q-function which describes the best action to take at each game state. To train the network we used the deep reinforcement learning algorithm which made our learning agent able to perform very well against all the built-in agents.

***Index Terms*—** Game AI, Real Time Strategy Game, Reinforcement Learning, Q-Learning, Neural Networks.

1. INTRODUCTION

The computer games industry has become one of the most growing components in the entertainment business, sparking a critical interest in many research fields such as computer science and robotics. In addition, the games business is a multi-billion industry with a total consumer spend of 24 billion dollars in 2011 [3].

Real-Time Strategy games (RTS) is a sub-genre of strategy-based computer games in which participants compete to control areas of the map, gather resources and defeat their opponents. An adept RTS player requires maintaining proper control of their resources, units, and structures that are distributed in a playing area and also requires being able to quickly adapt to changing conditions and moves made by the opponent. Such games offer a large variety of fundamental problems for the AI researchers such as real-time planning, uncertainty in decision making, opponent modeling, spatial and temporal reasoning and resource management [2]. *Command and Conquer, StarCraft, Warcraft* and *Age of Empires* are examples of RTS games.

In 2010, *Planet Wars* was presented as an RTS game under Google AI Challenge. This game has been widely used as a test-bed for AI-techniques due to its simplification. The *Planet Wars* game has only one type of resource (the planets) and one type of unit (the ships). Therefore, we used *Planet Wars* to implement and test this project.

Most of the commercial video games have a built-in artificial intelligence agent that gives the human player the opportunity to play the game against the computer. Building such an anonymous agent is a challenging task considering that the artificial agent has to be a good player but the game still has to be winnable by the human player. Many studies have been done in this field and the most successful ones used reinforcement learning and operated on hand-crafted features. Other studies used reinforcement learning with high-level features that are extracted from raw data such as images pixels. The most recent advances used deep reinforcement learning where they achieved remarkable performance in playing *Atari*, using the minimal amount of prior information about the game. This has made us wonder whether a similar approach can be used to play RTS games. With this paper we demonstrate the use of Q-learning with neural networks to overcome the RTS games challenges and build an AI Agent that is able to learn successful control policies to play the *Planet Wars* game.

2. RELATED WORK

Many researchers have studied methods for learning how to play video games in general and strategy games in particular. In this section, we will review some related work in learning to play video games. The best work in the field of deep reinforcement learning is the approach that was used in DeepMind's paper [7] where they demonstrated how a computer learned to play *Atari 2600* video games by observing just the screen pixels. This paper and their 2015 follow-up [8] was our primary inspiration, with [7] [8] they achieved outstanding results that outperformed the human players. They applied the same model architecture to learn seven different games. In [6] experience replay was introduced which lets online reinforcement learning agents remember and reuse experiences from the past. This technique was one of the techniques used in [7] [8] and our paper. One of the methods that we did not implement but would consider as an extension was introduced in [9] which is Prioritized Sweeping. Prioritized Sweeping depends on training the model on examples with the largest error. Deep reinforcement learning was also used in [4] where they used it to train an AI agent to play *Tetris*. They used convolutional neural networks to describe the best action to take at each time step; however, their network failed to converge without the help of heuristics.

One of the best learning approaches that used reinforcement learning was TD-gammon. They used reinforcement learning to learn how to play the game with a model-free reinforcement learning algorithm similar to Q-learning and a multi-layer perceptron with one hidden layer to approximate for the value function [13]. Perhaps the most similar prior work to our own approach is neural fitted Q-learning (NFQ) [10]. NFQ is an algorithm for efficient and effective training of a Q-value function represented by a multi-layer perceptron and based on the principle of storing and reusing transition experiences. Their method is evaluated on three control problems. Q-learning has also been previously combined with experience replay and a simple neural network in [6] and again to address a control problem which is the problem of automatically acquiring control policies by robots for task achieving. Q-learning has also been used in strategic games like *Civilization IV* [15] to place new cities on the map. The use of *Planet Wars* as a Q-learning platform was introduced by [5], where they implemented a Q-learning agent to play *Planet Wars* to test the ability of reinforcement learning in spatial reasoning and high-level resource management.

3. BACKGROUND

We will first discuss the *Planet Wars* game and then we will present an overview of reinforcement learning, Markov Decision Processes (MDPs), Q-Learning and some basic ideas that we will use later in our Q-Learning implementation.

3.1. Planet Wars

The goal of *Planet Wars* is to conquer all the planets on the map or eliminate all the opponents. The game takes place on a map on which several planets are scattered through the galaxy. Each planet has a number of ships and a growth rate (which indicates the growth ratio of the ships' number on the planet), in addition to that planets can be controlled by any player or remain neutral. Players can take over the planets by sending ships from their controlled planets to the other ones. The number of ships required to take over a planet is the same as the number of ships that defend the planet. In *Planet Wars*, the player has a very large set of possible actions. For example, the player could steal a neutral planet (take over a neutral planet), he could wait until the enemy takes over the neutral planet and loses its ships and then take the planet from him on the next turn, he could also redistribute his ships to be ready for defense or attack or combine forces of multiple planets to take over a planet or defend one of his own. Redistribution can be seen in Fig. 1.a, the red player in this image is redistributing its power. Combining forces of multiple planets can be seen in Fig. 1.b, the red player is combining its forces to attack the blue player. The difficulties of this game lie in the unbounded number of possible moves and the large number of available planets. This agent will therefore require a long time to be trained. In addition, there is the problem of finding a metric which best evaluates the player strength.

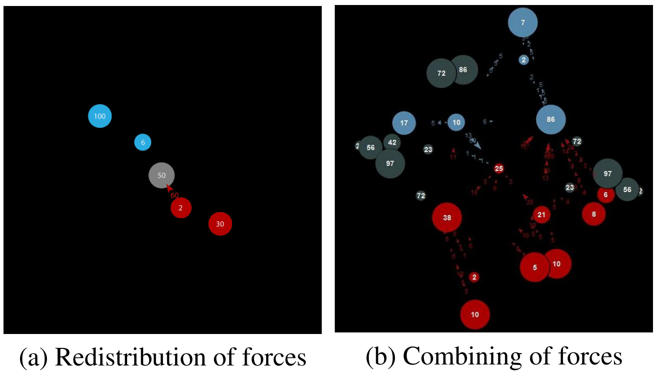
(a) Redistribution of forces (b) Combining of forces

Fig. 1. Screen-shots of a run in *Planet Wars*. The blue planets belong to player1, the red planets belong to its opponent and the gray planets are neutral planets. The triangles are fleets and the number in the triangles represents the ships. The number inside the planets means the growth rate of the number of ships in it.

In order to train the agent to make the best possible moves, we chose to use Q-learning with neural networks. Our approach is similar to the one in [7] but we used a different neural network structure. In order to grapple with the large number of planets available, we added a restriction to the number of planets in each map and we used *Planet Wars* open source code [1] that was developed in Google AI Challenge 2010. This code has 4 main agents: the *Random* agent which makes completely random moves, *AllToWeak* agent where all the agent planets attack the weakest enemy planet, *AllToCloseOrWeak* agent where the agent planets attack either the closest enemy planet or the weakest one and finally the *Evolved* agent which plays according to manually tuned weights (in practice we found that the *Evolved* agent is the best one among all the previously mentioned agents).

3.2. Reinforcement learning

The idea of learning by interacting with our environment is probably the first to occur to us when we think about the nature of learning. Interactions produce a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. In this paper, we will consider an approach of learning from interaction. Reinforcement learning is about learning from interaction how to behave in order to achieve a goal [12]. In reinforcement learning, the agent interacts with an environment over a sequence of discrete time steps. The agent make choices called actions, these choices are made based on states and then the choices are evaluated by rewards. In our case, the environment is the *Planet Wars* game simulator, at each time step, the agent select an action a from the set of legal actions A. The action is passed to the game simulator and modifies the game state and the game score. The agent observes a vector of values that represent the current game state and receives reward r which represents the change in game score.

3.3. Markov Decision Processes

Markov Decision Processes (MDPs) is a mathematical framework for modeling sequential decision-making when in situations where outcomes are partly random and partly under the control of a decision maker [4].An MDPs model is represented by a tuple $< S, A, P, R >$ that contains:

1. A set of possible world states S.
2. A set of possible actions A.
3. A set of state transition probabilities $P : P(S'|s, a)$, the probability of transitioning from the state s to the state s' when action a is taken by the agent.
4. A real-valued reward function $R : R(s|a)$ a real-valued immediate reward for taking action a in state s [4].

At each time step, the agent observes a state from the environment then performs a certain action in the environment and gets a reward. The actions change the environment and lead to a new state where the agent can perform another action and so on. The actions are chosen according to a policy. The goal is to define a policy that maximizes the future rewards. We can express the total future reward at time step t as $\sum_{t'=t}^{T} r'_t$ where T is the termination time step. As the environment is stochastic we cannot be sure that the next time we perform the same actions we will get the same rewards, for this reason, we made an assumption in this paper that the future reward is discounted by a factor $\gamma = [0, 1)$ and defined the future reward as $\sum_{t'=t}^{T} \gamma^{t'-t} r'_t$.

3.4. Q-Learning

Q-learning is a form of model-free reinforcement learning technique. It can also be viewed as a method to find the optimal policy for action selection given an MDPs [14]. In Q-learning, we define a function $Q(s, a)$ that represent the maximum expected return achievable by following any policy after we perform action a in state s.

$$Q(s, a) = \max R_{t+1} \tag{1}$$

We can also write the Q-value of state s and action a in terms of the Q-value of the next state s' as

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a') \tag{2}$$

this is called the Bellman equation. The core idea behind Q-learning is to iteratively approximate the Q-function by assuming the old value and making a correction based on the new information

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \tag{3}$$

Where $\alpha : (0 < \alpha \leq 1)$ is the learning rate which control the difference between previous and new Q-value, $maxQ(s', a')$ is the estimation of the optimal future value. The neural networks are considered to be function approximators, so we can represent our Q-function with a neural network that can take the game state as an input and output the Q-value for each possible action we call this neural network a Q-network. Q-network can be trained with a squared error loss function

$$L = \frac{1}{2}(r + \gamma \max_{a'} Q(s', a') - Q(s, a))^2 \tag{4}$$

where $r + \gamma \max Q(s', a')$ is the target. Note that the targets depend on the network weights; unlike the targets used in supervised learning that are fixed before the learning begins. This algorithm was proposed by [7]. In section 4, we will explain more about Deep Q-learning Algorithm.

3.5. Experience Replay

Experience replay lets online reinforcement learning agents remember and reuse experiences from the past [11]. Experience replay can also reduce the number of action executions required by saving the learning trials [6]. The experience is a tuple that consists of the current state, the action, the reward and the future state $< s, a, r, s' >$. Like [7] and [8], we used experience replay to create a “replay memory” for our neural network to train on. This technique is used as the following, during the training, the tuple is stored in a replay memory and then random mini-batches are selected from the memory to train the network instead of using the most recent transition. Using experience replay will prevent the network from reaching a local minimum by averaging the behavior distribution over many previous states.

3.6. Epsilon-greedy policy

One of the challenges that arise in reinforcement learning is the balancing between exploration and exploitation. On one hand to maximize the reward the reinforcement agent must prefer actions that are explored or tried in the past and found to be effective in improving the reward. On the other hand, the agent also needs to discover and explore new actions that have not selected before. The agent has to exploit what it already knows in order to obtain a reward, but it also has to explore in order to make better action selections in the future. This is called the exploration-exploitation dilemma [12].

When the Q-network is initialized randomly; its prediction is also random. Therefore we can say that the agent will be exploring by choosing the action that has the highest Q-value. When the Q-Network is trained it returns more consistent Q-value and the amount of exploration will decrease. This is called a greedy exploration because the agent settles with the first action it finds. To solve this issue we used epsilon-greedy exploration. The epsilon-greedy policy is a strategy used in [7] where the agent takes a random action with a probability ϵ and choose the greedy action with probability $1-\epsilon$. This is done to ensure that the agent explores the search space and look for the optimal actions. In [7] the epsilon value decreases over the time from 1 to 0.1 and fixed at 0.1 thereafter.

4. DEEP REINFORCEMENT LEARNING

In [7], they used a deep neural network which operated on a huge number of RGB images and processes the training data using stochastic gradient updates. Their approach was very successful that it was able to outperform human players. This made us wonder how this algorithm will perform in a Real Time strategy game such as *Planet Wars*.

In this paper, we used the experience replay technique and the epsilon-greedy algorithm in the same way we explained in section 4.5 and section 4.6. We set our Q-network to work on a fixed length representation of histories and an epsilon value that start with 0.3 and decrease over the time. The full Deep Q-learning algorithm can be shown in Algorithm.1. According to [7], this approach has several advantages such as greater data efficiency because each experience might be used in several weight updates and breaking the correlations between the samples which will reduce the variance of the updates. On the other hand, the algorithm only stores a finite number of experience tuples in the replay memory which means that some important transition will be deleted. In addition to that, the algorithm will sample uniformly from the reply memory which gives equal importance to all transitions. These problems can be solved using Prioritized Sweeping, which can be considered as a modification for this project.

4.1. Dataset and Features

As we mentioned before, in [7] they used a huge set of RGB images to train their network which will take a massive amount of resources and a very long time. Considering that we had neither the time nor the resources we needed to operate on hand-crafted features.

The *Planet Wars* game has a lot of map structure variations so we need our feature representation to be independent

Algorithm 1 Deep Q-Learning with Experience Replay

```
Initialize memory D
Initialize batch size
Initialize Q-Network with random weights
Get initial state s
**repeat**
  With probability $\epsilon$ select a random action $a_t$
  otherwise select $a_t = \max_a Q(s,a)$
  execute the action $a_t$
  Get reward $r$
  Get new state $s'$
  Store the experience tuple $< s, a, r, s' >$ in D
  Take a random sample mini-batch of experience from D

  **for each** experience $< s_i, a_i, r_i, s'_i >$ in mini-batch **do**
    **if** $s'_i$ is a terminal state **then**
      Set target $t = r_i$
    **else**
      Set target $t = r + \gamma \max_{a'} Q(s'_i, a'_i)$
    **end if**
    Train the network using $(t - Q(s'_i, a'_i))^2$
  **end for**
**until** end of game
```

of the map. We decided to use features that are a combination of planet features and cluster features. The cluster is calculated by finding a set of our planets that can influence a specific planet or in other words, finding a set of our planets that are close to a specific planet. The planet-cluster combinations are calculated for each planet on the game map.

To be a discuss the features, we need to define a few heuristics that we designed to present our state information and select relevant data for better feature representation. The first heuristic is the score function which calculates a score value for each planet. This score indicates the state of a planet at t turns in the future that is to say that the score value will be positive if the planet is kept by its owner and negative otherwise. To calculate this value we use the following equation:

$$\begin{gathered} score_{p,t} = score_{p,t=0} + score_{p,t=1} + ... + score_{p,t} \\ score_{p,t=0} = S_p \\ score_{p,i} = \frac{score_{p,i-1}}{|score_{p,i-1}|} * growth_p + \frac{If_{p,i} - IE_{p,i}}{d * \log_2 i + 1}, i > 0 \end{gathered} \tag{5}$$

where t is the number of turns, d is the decay value, $p \in P$ is a planet from the game, S_p is the current number of ships in planet p, $growth_p$ is the growth of planet p and $If_{p,i}$, $IE_{p,i}$ are the incoming friendly and enemy fleets that need i more turns to reach planet p. We also presented the term $d * \log_2 i + 1$ which is a logarithmic decay. This term indicates that the farther the fleets are from a planet, the less influence they have on that planet. The second heuristic is the influence function

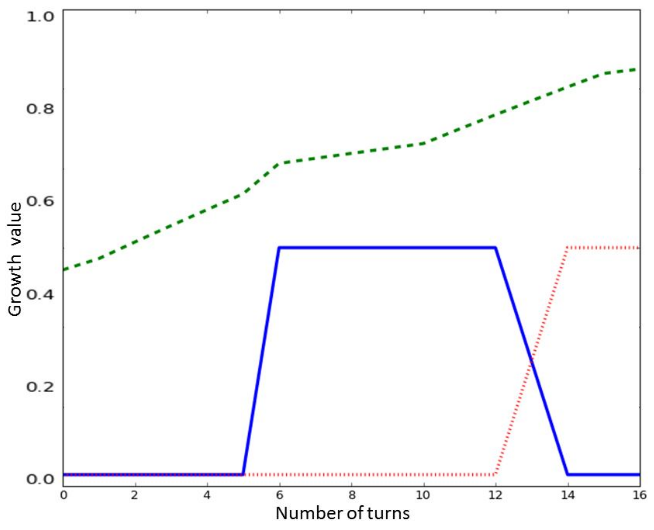

Fig. 2. The blue solid line is the QL Agent growth curve, the green dashed line is the desired growth and the red dotted line is the enemy curve. The plot on the left was produced for a game where the QL Agent lost.

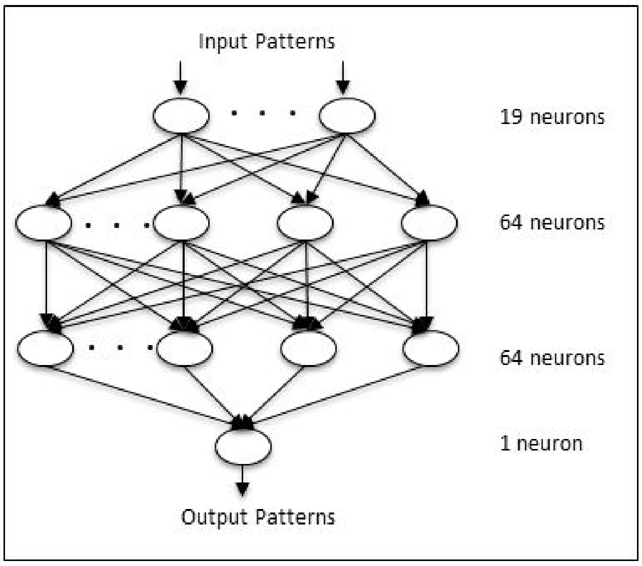

Fig. 3. An illustration of the multilayer feedforward neural network architecture used in the QL agent.

which is calculated by the following equation:

$$f_p = \sum_{i=1}^{P} score_p/distance_{p,i} \tag{6}$$

where P is a set of planets (friendly or enemy planets), $score_p$ is the score of planet p and $distance_{p,i}$ is the distance between planet p and planet i measured by the number of turns.

The cluster features are similar to the planet features the only difference is that they are calculated for all the planets in the cluster. The cluster features include the total number of ships, the total growth rate, the total score, the total influence of the friendly planets, the total influence of the enemy planets, the total mean influence of the friendly planets and finally the total mean influence of the enemy planets. On the other hand, the planet features include the number of the planet ships, the planet growth rate, the planet score, the influence of the friendly planets, the influence of the enemy planets, the mean influence of the friendly planets and finally the mean influence of the enemy planets.

In addition to the planet-cluster features we added the overall growth ratio of our planets, the overall growth ratio of the enemy planets, the overall of our ships ratio, the relationship between the planet and the cluster which take the values (0: the planet belongs to the enemy, 0.5: the planet belongs to us, 1: the planet in neutral) and finally the planet distance from the cluster. So the network input is a vector of 19 features for each planet cluster combination on the game map.

4.2. Rewards

The reward is based on three main parts, first the total growth ratio of the QL agent, second the enemy total growth ratio and third the desired growth which is a function that we designed to measure how the perfect growth in the game should be. The desired growth is calculated using the following formula:

$$1 + initGrowth - 130^{\frac{t}{800}} \tag{7}$$

where the $initGrowth$ is a constant value equal to 0.06 and t is the number of relevant turn.

The desired growth is added to help the agent learn to expand. The enemy growth ratio and the desired growth ratio can both decrease the reward. If the enemy total growth ratio is bigger than QL Agent total growth ratio it means that the enemy has more planets and ships this will give our agent a bad reward and also if the total growth of the QL Agent is less than the desired growth this means that our agent is not performing as good as it should be and this will decrease the reward value as well. The reward is given with the following equation:

$$Reward = (growthR - growthE) + (growthR - growthD) \tag{8}$$

where $growthR$ is our agent growth ratio, $growthE$ is the enemy growth ration and $growthD$ is the desired growth. Fig. 2 shows the QL agent growth curve, the desired growth curve, and the enemy growth curve for one game before the training is done. Notice that the plot in Fig. 2 shows how the growth curve for our agent falls far behind the desired growth and at the end of the game, we started losing against the enemy.

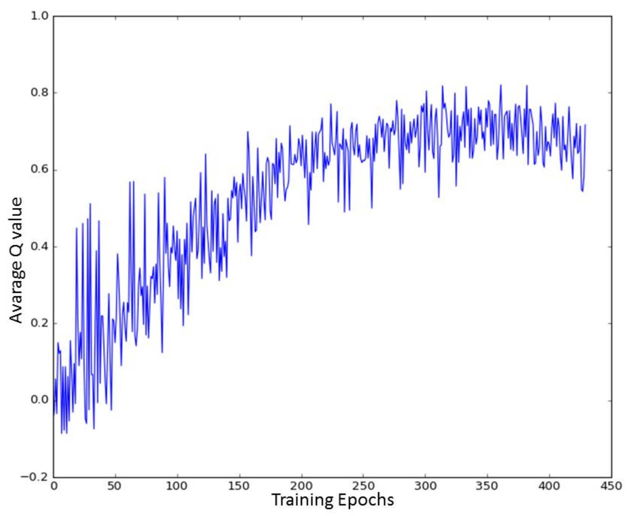

Fig. 4. The average predicted Q-value of a set of states on the *Planet Wars* game. Each epoch corresponds to 5000 mini-batch weights updates. This plot is generated after 100 game while playing against the *Random* agent.

4.3. Network architecture

There are several possible ways of parametrizing Q using a neural network. In [7] they suggested using an architecture in which there is a separate output unit for each possible action and only the game state representation is an input to the neural network. Each one of the outputs is mapped to the predicted Q-values of the individual action for the input state. This approach has the advantage of calculating the Q-values for all possible actions in a given state with one forward pass through the network. However, this architecture has a restricted number of outputs that need to be set beforehand. In our case, this will not work because we have a massive number of possible actions in the *Planet Wars* game, even if we limited the number of actions to one only, which is sending ships, we still need to decide the destination of the ships. Therefore we instead used another architecture where there is only one output for the neural network.

We used a simple feed-forward neural network, where the input of the neural network is a vector of 19 features for each planet-cluster combination on the game map. The first hidden layer has 64 neurons, the second hidden layer has also 64 neurons and the output layer consist of a single output neuron that indicates the percentage of success from which planet-cluster combination to send ships, that is to say if we have planet-cluster combinations, the network will always output a value, we take the best output (the best planet-cluster combination) and send ships from cluster to planet. After deciding the destination of the ships we send 50% of the existing ships. Fig.3 shows the network architecture.

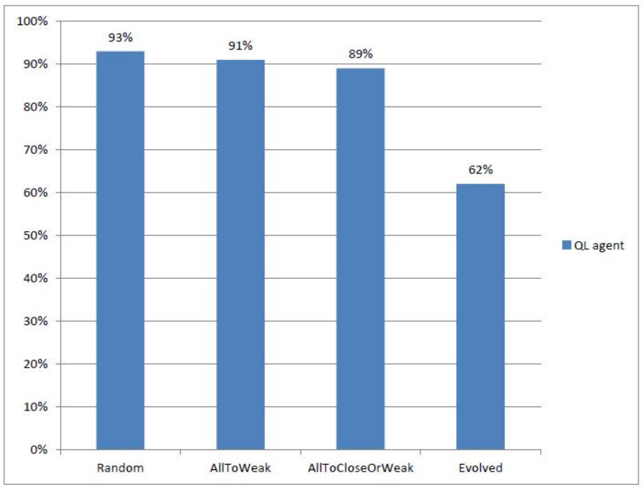

Fig. 5. The percentage of victories the QL agent achieved when playing against each of the built in agents. This data is collected by playing 1000 game with each of the built-in agents after the training process is done.

5. RESULTS

5.1. Experimental Setup

We have performed experiments with four of the *Planet Wars* agents (*Random, AllToWeak, AllToCloseOrWeak*, and *Evolved*). In These experiments, we used the RMSProp algorithm with mini-batches of size 32. The behavior policy during the training was epsilon-greedy with a decreasing epsilon value from 0.3 to 0 and fixed at 0 thereafter. We used a replay memory of 5000 stored experience. Training the agent was over 1000 games with each one of the four built-in agents and with different variations of map sizes and structures.

5.2. Discussion

We noticed that starting with a small epsilon value was better in our case this might be true because the agent is training on hand-crafted features which are related to the game and not fully raw data.

In addition, we noticed that our agent performs better with big maps when playing against the *Random* agent, this sound obvious since the *Random* agent sends ships to random planets on the map and by playing on a big map it might take the agent a long time to conquer a planet. Furthermore, some of the built in agents perform better in certain map structures so, when the training is done on big maps with different structures, the QL agent will perform much better and the map structure will no longer be a problem in determining the winner of the game.

5.3. Experiments

For our experiments, we used the Q-value as an evaluation metric during the training process, as suggested in [7]. The Q-value represents an estimation of the maximum discounted future reward the agent can achieve when we perform action a in state s, and continue optimally from that point on. Fig. 4 shows the average predicted Q-value. Notice that the curve is increasing but it tends to be noisy.

Other experiments have been conducted proposing a direct comparison between the four built-in agents and our QL agent. The percentage of victories obtained by QL agent when it played against the four built in agents is shown in Fig. 5. Notice that the agent achieved very good results against the *Random, AllToWeak* and *AllToCloseOrWeak* agents and medium performance with the *Evolved* agent considering that it is the best one.

6. CONCLUSION

In this paper, we studied a Q-learning approach with feed forward neural networks as the Q-function to build a bot capable of playing an RTS game called *Planet Wars*. We used hand-crafted features that represent the game state as an input to the neural network and the deep reinforcement learning algorithm which was proposed in [7] to train the network. The QL agent performance was evaluated in multiple methods one of them included comparing it with other built in weak and strong agents and it achieved very good results with both kinds. These results show that using Q-learning can provide powerful tools which allow agents to adapt and improve quickly; even in complex scenarios such as strategy games.

The objective of this work was to test if using Q-Learning with neural networks can create competitive bots for RTS games. Our QL agent demonstrated the capability of Q-reinforcement learning in handling RTS games problems. In the future, we are interested in extending this line of research in several ways. For instance, we could achieve convergence on a more sophisticated heuristic reward function and find a better technique to determine the best percentage of ships to send. We can also add Prioritized Sweeping to the Deep reinforcement learning algorithm. Using these approaches, we may be able to improve the QL agent performance against the *Evolved* agent.

7. REFERENCES

[1] Google AI Challenge 2010. Planet wars source code. `https://github.com/alexjc/planetwars`, 2014.

[2] Michael Buro and Timothy M. Furtak. Rts games and real-time ai research. In *In Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS*, pages 51–58, 2004.

[3] Entertainment Software Association (ESA). Essential facts about the computer and video game industry, 2012.

[4] R. A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA, 1960.

[5] Michael A. Leece and Arnav Jhala. Reinforcement learning for spatial reasoning in strategy games. 2013.

[6] Long-ji Lin. Reinforcement Learning for Robots Using Neural Networks. 1993.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. 2015.

[9] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. 1993.

[10] Martin Riedmiller. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. 2005.

[11] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. 2015.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[13] Gerald Tesauro. Temporal difference learning and td-gammon. 1995.

[14] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992.

[15] Stefan Wender and Ian Watson. Using reinforcement learning for city site selection in the turn-based strategy game civilization iv. 2008.