# Reinforcement learning applied to RTS games

Paper XXX

## ABSTRACT

Reinforcement learning algorithms are often used to compute agents capable of acting in environments without any prior knowledge. However, these algorithms struggle to converge in environments with large branching factors and their large resulting state-spaces. In this paper, we develop an approach to compress the number of entries in a Q-value table using a deep auto-encoder. We develop a set of techniques to mitigate the large branching factor problem. We apply such techniques in the scenario of a Real-Time Strategy (RTS) game, where both state space and branching factor are a problem. We empirically evaluate an implementation of the technique to control agents in an RTS game scenario where classical reinforcement learning fails and point towards future work.

## CCS Concepts

•**Computing methodologies** → **Multi-agent systems;**

## Keywords

Reinforcement Learning, Auto-encoder, Multi-agent

## 1. INTRODUCTION

Reinforcement learning (RL) is a subfield of machine learning that focuses on maximizing the total reward of an agent through repeated interactions with a stochastic environment [9]. This process occurs as an agent interacts with the environment multiple times, exploring the state-space of the environment and evaluating the reward of executing different actions. In order to converge to an optimal policy, an agent using traditional reinforcement learning approaches must explore the entire state space, executing every possible action. Consequently, these algorithms can take a long time to to find optimal actions for all configurations of the environment in environments where the state space, or the number of possible actions in each state, is very large. However, this is intractable for complex problems such as multiplayer computer games. Here, the number of possible states is large that there is neither sufficient storage capacity to store, nor sufficient time to visit all possible states [10].

To avoid dealing with large state spaces, it is possible to create a compact state representation of an environment, focusing only on the most important features. With compact state representations, it is possible to learn complex tasks in high-dimensional spaces, if the compact state represents the most important features of the state. However, creating a compact state representation to represent huge state spaces is a challenging task. There are currently two ways two overcome this limitation. First, we can apply machine learning techniques to generalize the state, assuming we can define features capable of representing the state. Second, we can map similar states into the same state so that an agent can learn to act in states it has never visited using information from such similar states. Recent developments on deep learning [13] have yielded mechanisms to reduce dimensionality and find efficient encodings, using auto-encoders. In this paper, we develop an approach that uses a deep auto-encoder to compress the state-space and create an efficient encoding capable of efficiently representing the state-space. Even with a small state-space, agents must try every pair of state and action multiple times before it has confidence on a learned policy. This, in turn, subjects the system to a combinatorial explosion when stored actions represent combinations of multiple agent's actions, so the auto-encoder technique alone does not ensure that the algorithm can visit all combinations of actions from multiple agents. In this paper we address this problem with by sharing the experience of multiple similar agents spread over the state-space.

Thus, our contribution is a set of techniques to use reinforcement learning in environments with multiple agents, and large state-space and branching factor. Recent research applies reinforcement learning to simpler games that allow the input to be the raw low resolution image output from the game. However, these games have a relatively simple state-space, which is not the case of RTS games. We thus apply reinforcement learning using all available information in a complex game where traditional reinforcement learning algorithms are not viable. We evaluate these approaches empirically in a multi-agent domain with a large state-space generated by the scenario of a Real-time Strategy (RTS) game called MicroRTS [7].

In this work, we aim to train an agent capable of playing the game competitively against other artificial intelligence players, using reinforcement learning techniques and a deep auto-encoder. To do so, we must develop a state representation. We explain each detail necessary to ensure a that a reinforcement learning algorithm can be converge and play the RTS game competitively.

## 2. BACKGROUND

### 2.1 Q-Learning

Q-Learning is a model-free reinforcement learning algorithm that does not need the transition function to learn the optimal policy. The agent learns to act in the environment

by testing the possible results of performing an action in a certain state, learning the utility of performing an action $a$ in a state $s$. The main idea of Q-Learning is to learn the utility of executing an action when the agent is at a particular state, rather than learning the utility of each state directly and then computing a policy. The pair of state-action is represented by a Q-Value. A Q-Value $Q(a, s)$ contains the utility of executing action $a$ on the state $s$.

Since the agent does not know the transition function, the agent will learn how to act through checking the possible results of performing an action in a certain state, thus the algorithm learns the utility of performing an action $a$ in a state $s$. The utility of a state can be described as the highest reward that is possible to obtain in a state. The utility of a state can be written as the following equation:

$$U(S) = max_a Q(a, s) \quad (1)$$

we use equation 1 instead of Bellman equation, to update a Q-Value, by removing the transition function from the equation. The update rule of the state of the Q-Learning algorithm can be written as the following equation:

$$Q(a, s) = Q(a, s) + \alpha(R(s) + \gamma max_{a'} Q(a^{'}, s^{'}) - Q(a, s)) \quad (2)$$

were $\alpha$ is the *learning rate*, which ranges between 0 and 1. When 0, nothing is learned, and when 1, the learned value is fully considered. $\gamma$ is the discount factor, which ranges between 0 and 1. When 0, future rewards are irrelevant, and when 1, future rewards are fully considered. Equation 2 implements the Q-Learning update as part of Algorithm 1, based on the specification from [14].

The Q-Learning algorithm requires a trade-off between exploration and exploitation. The algorithm can either choose to continue exploring different actions on states, or exploit the current computed policy to act in an environment. The exploration function can differ in each implementation, such as a balance between exploration and exploitation developed by [12], but in most cases it forces the agent to explore actions that were never tried in certain states, exploring at least once every possible action in every state. The exploration function controls how the agent behaves until the algorithm converges. Since every Q-Value starts as null, Q-Learning relies only on the exploration function to start exploring the environment. We denote the exploration function in Algorithm 1 as function $f$.

The SARSA (State-Action-Reward-State-Action) [8] algorithm is a variation of Q-learning, with a small change to the update rule. SARSA learns action values relative to the policy it follows, while Q-Learning does it relative to the exploration policy. Under some conditions, they both converge to the real value function, but at different rates. The update rule for SARSA is represented as follows:

$$Q(a, s) = Q(a, s) + \alpha(R(s) + \gamma Q(a^{'}, s^{'}) - Q(a, s)) \quad (3)$$

where $a'$ is the action taken in state $s'$. The update rule is applied at the end of each $s\ a\ r\ s'\ a'$ quintuple. The SARSA algorithm will act on the environment and update the policy based on actions taken.

## 2.2 Artificial Neural Networks

Artificial Neural Networks are a group of models, loosely inspired on the human brain, designed to recognize patterns. ANNs are composed of multiple nodes, organized in layers. A node is an artificial neuron responsible for computing the output. The nodes combines input from the data with a set of coefficients (or weights) that amplifies our suppress an input. An ANN contains three types of layers [1]:

1. An **Input layer** responsible for receiving the signal (data) that feeds the neural network.
2. A **Output layer** responsible for receiving the signal from the hidden layer (or input if there is no hidden layer in the network) and producing the output of the network. For example, in a classification problem, this layer will output which class the input belongs to.
3. A **Hidden layer** responsible for receiving the signal from the input layer. Every layer between the input and the output layer is considered a hidden layer.

These layers are interconnected, sending information from one layer to another subject to weights in the inter-layer connections. Artificial Neural Networks solve multiple machine learning task problems, such as classification, regression and dimensionality reduction. To solve different tasks, ANNs can use supervised, unsupervised and reinforcement learning algorithms.

A Deep Neural Network (DNN) is an ANN with multiple hidden layers between the input and the output layers. There are several types of DNNs, such as deep belief networks, deep auto-encoders, convolutional neural networks and deep Boltzmann machines [17]. These types are defined based on the architecture of these networks. In this work, we use a deep auto-encoder to encode the values in the Q-table.

## 2.3 Deep auto-encoder

A deep auto-encoder is a type of deep neural network capable of converting inputs to a more compact representation of itself [13]. A deep auto-encoder is composed of two connected symmetrical neural networks. The first network encodes the input into an internal representation and the second decodes the internal representation back to the original input.

**Algorithm 1:** Q-Learning pseudo code.

**Input:** percept, a percept indicating the current state $s'$ and reward signal $r'$
**Output:** An action $a$.
**Persistent**: $Q$, a table of action value indexed by state and action, initially zero ;
$N_{sa}$, a table of frequencies for state-action pairs, initially zero ;
$s$, the previous state, initially null;
$a$, the previous action, initially null ;
$r$, the previous reward, initially null;
**if** $isTerminal(s')$ **then**
  $Q[s', None] \leftarrow r'$ ;
**end**
**if** *$s$ is not null* **then**
  $N[s, a] \leftarrow N[s, a] + 1$ ;
  $Q[s, a] \leftarrow Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$ ;
**end**
$s \leftarrow s'$; $r \leftarrow r'$ ;
$a \leftarrow \arg\max_{a'} f(Q[s', a'], N[s', a'])$ ;
**return** $a$

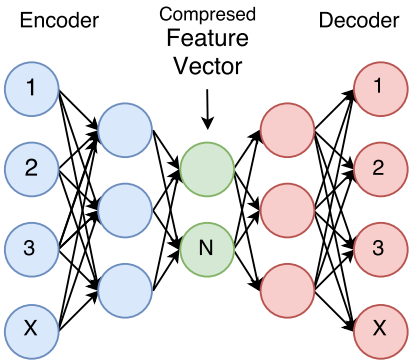


**Figure 1: Deep auto-encoder architecture.**

We illustrate the architecture of an auto-encoder in Figure 1, which comprises two neural networks connected by a middle layer containing a compressed representation. Here, X represents a variable number of nodes for both the input and the output layers, N represents a variable number of nodes for the layer that the connects both networks. Blue nodes represent the nodes of the encoder network, transforming the input in the compressed representation. The green nodes represent the nodes where the input becomes its compressed encoding. The red nodes represent the nodes of the decoder network, which, from the encoded representation, incrementally transforms the encoded data back to the original form.

Suppose we want to compress the encoding of an 28x28 image (a total of 784 pixels) into a compressed 30 pixel representation. The image is fed to the neural network as an array of binary values, where each pixel is fed to one of the input nodes. A sketch of the encoder is determined as follows:

$$784(input) \rightarrow 1000 \rightarrow 500 \rightarrow 250 \rightarrow 100 \rightarrow 30$$

In this sketch, the first hidden layer has more nodes than the actual image input. To represent the decoder, we provide the following sketch:

$$30 \rightarrow 100 \rightarrow 250 \rightarrow 500 \rightarrow 1000 \rightarrow 784(output)$$

Once we train a deep auto-encoder, the decoder network is no longer necessary, since this part of the network is only used to compute the error of the decoded data during training. Since it is impossible to know the expected output of an encoder (we want the encoder to solve this problem), we train the network to return the exact input as output. After training the auto-encoder, we can retrieve the compressed feature vector from the hidden layer in the middle of the network, i.e. the green nodes in Figure 1.

## 2.4 MicroRTS

MicroRTS is a simple implementation of a Real-time Strategy game, designed for the sole purpose of AI research. Developed by Ontañón [7], MicroRTS is a well structured implementation of an RTS game in Java. The advantage with respect to using a full-fledged game like Starcraft, and the main reason we choose MicroRTS, is the fact that MicroRTS is much simpler, becoming a useful tool to quickly test theoretical ideas, before trying on to full-fledged RTS games.

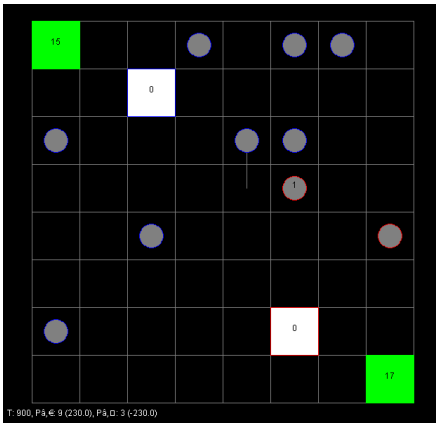

**Figure 2: MicroRTS game state.**

MicroRTS consists of two players trying to eliminate every single structure and unit of the enemy. Figure 2 illustrates a game state of the MicroRTS. There are 4 types of units in MicroRTS:

1. **Worker**. This unit is responsible for harvesting minerals and constructing structures. This unit can also fight, but does very little damage. Represented by the gray circles.
2. **Light**. Light units do little damage, but are extremely fast. This type of unit can only attack.
3. **Ranged**. Ranged units are capable of ranged attacks. They have moderate damage and moderate speed. This unit can only attack.
4. **Heavy**. Hard units are heavy attack based units. They do high damage, but are extremely slow. This unit can only attack.

Units requires resources to be produced, but more than that, they require structures. There are three types of structures in the MicroRTS. Those are:

- **Base**. The main structure. This structure is responsible for producing workers. This structure is also where the workers return the minerals they harvested. The game starts with a base for each player. Can be attacked. Represented by the white squares.
- **Barracks**. Auxiliary structure. This structure is responsible for producing light, ranged, and heavy units. It can be built by the workers by using resources. Can be attacked.
- **Minerals**. Minerals can be harvested by the workers to obtain resources. Not an exactly structure, cannot be attacked. Minerals are finite, and each player starts with one source. Represented by the green squares.

The game is totally observable, so the player can see every enemy action. With those components, MicroRTS provides a good simplification of a full-fledged RTS game.

The decoder is not needed, as the objective is to encode the state representation, with no necessity of decoding it after. However, to train the Deep auto-encoder, it is necessary to train the neural network with both the encoder and

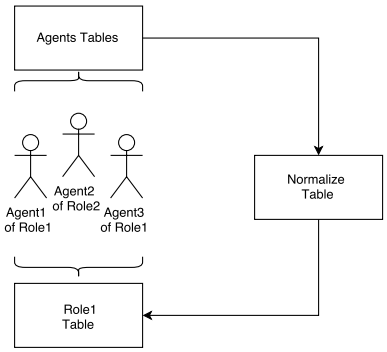


**Figure 3: Unit Q-learning Diagram**

the decoder network. This is necessary because to train the network, the expected output must be the same as the input and the expected output. Since it is impossible to know the expected output of an encoder (we want the encoder to solve this problem), we train the network to return the exact input as output. After training the auto-encoder, we can retrieve the compressed feature vector from the hidden layer in the middle of the network, as shown in Figure 1.

# 3. APPROACH

## 3.1 Unit Q-learning

Each unit in MicroRTS has approximately 5 possible actions in each state, so, even if this may be manageable for relatively small state-spaces for a single agent, it can become a problem when learning combinations of actions for multiple agents simultaneously. For example, this becomes an issue when ordering 10 units simultaneously, resulting in 9765625 possible actions, clearly a combinatorial explosion. To avoid dealing with such huge branching factor, we apply Q-Learning to each unit individually, executing parallel Q-Learning updates on each training episode, one for each unit. At the end of the training episode, units with the same role (such as workers) *share* their experience, building a new set of Q-values. The algorithm updates the Q-values of the units at the start of each training episode each iteration. This process is illustrated in Figure 3, where the units are the agents, the Agent table are the group of each table of the each agent and the role table is the agent tables normalize.

The experience sharing between the units is accomplished by merging the Q-tables of the units of the same role, by normalizing the Q-values that both agents visited. We define the normalization function as follows:

$$Q(s,a) = \frac{\sum_{i=0}^{agents} Q_i(s,a) * frequency(Q_i(s,a))}{frequency(Q(s,a))} \quad (4)$$

where $Q(s,a)$ is the new Q-value for all units for state $s$ and action $a$, and $Q_i(s,a)$ is the Q-value of unit $i$ for state $s$ and action $a$. The idea is that agents who visited more times a Q-value pair, are more able to determine the value of such Q-value. In Algorithm 2, the *mergeTables* method implement the update described by the equation. The algorithm consists of three steps:

- Assign a Q-table to a unit based on its type.
- Compose an action for the state, using one action for each unit.
- In the terminal state, merge the Q-tables of the units with the same type.

In Algorithm 2, the first *for* loop assigns the Type table to each unit and defines the action for the state by selecting each individual unit action. In the second *for* loop, if the current state is a terminal state, we merge the tables of units with the same type. Finally, in the last line of the algorithm, we return the composed action.

**Algorithm 2:** Unit Q-learning pseudo code.

```
Input: s, The actual game state.
Output: An action Action.
Persistent: QTables, a set containing one Q-table to
 each unit type ;
   U, set of player units ;
   s, the previous state, initially null;
Action = ∅ ;
for unit u ∈ U do
    if u.QTable = ∅ then
        u.QTable := QTables(u.type) ;
    end
    Action.add(QLearning(u,s)) ;
end
if isTerminal(s) then
    for type ∈ U do
        Q := mergeTables(type) ;
        QTables(type) := Q ;
    end
end
return Action
```

The worker unit of MicroRTS is specialized in harvesting resources, but can be used to offensively to attack and pressure the enemy. Since these two very distinct behaviors would be put in the same Q-table, we propose separate roles. We discuss how we implemented separate roles in Section 4.

## 3.2 State encoding

Our state encoding approach consists of 3 steps. First, we design a binary representation for the state space we would traditionally store in the Q-table, and call this representation *raw encoding*. Second, we design an auto-encoder that takes as input the number of bits we chose for the raw encoding and narrows it into 15 number of neurons, creating a *canonical encoding*. Finally, we train the network using state samples. We execute Q-Learning using the canonical encoding to store values in the Q-Table.

Our raw encoding consists of a binary representation of 156 bits. The 64 first bits store the player units position. Each bit represents a position in the grid. Using Figure 2 as example, the following bit encoding represents the blue units:

$$Blue = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The next 64 bits are used to encode the enemy units, using the same concept. We must also represent the health points of the units, however, since it is hard to represent the health points of each individual unit, we chose to represent only the health points of the bases. We assign 4 bits to represent the health points of the player base, and 4 bits to represent the health points of the enemy base, since the base unit has 10 health points. To represent the resources a player has, we use 5 bits, since the maximum possible number of resources is 25. Five more bits must be used for the resources of the enemy player. To represent the unit carrying a resource, we design one bit if any unit is carrying a resource, and 1 bit if any enemy unit is carrying a resource. We do not define unit type in the representation. However, we design 1 bit to represent if the player has a barracks, and 1 bit to represent if the enemy has a barracks. The last 6 bits represent the action executed in that state.

To avoid dealing with a large Q-Table and be able to model Q-Learning in complex environments, we aim to build a state representation that focuses on only the most important features of the environment. However, discovering the most important features of an environment is not trivial and completely changes as the environment changes since features important in one domain may not be important or even exist in another domain. To discover theses features, we model a deep auto-encoder capable of reducing the state representation to a compact form, representing only the most important features.

The first layer of the network will receives a binary state-representation corresponding to our raw encoding. The number of nodes in this layer will be referent to the number of bits used to represent the state, 156 bits. Each bit represents a feature of the state. The goal is to reduce this 156 bits representation to a more compact one, which does not affect Q-Learning convergence. Through empirical tests [11], Tesauro recommends that the Q-Learning lookup table size does not surpass 10000 entries. Since we are working with binary representations, it would be ideal to compress to a number of bits that represent less than 10000 states. The number of bits to create a representation with less possibilities than 10000 is 13 bits, as $2^{13}$ is 8192 possible combinations of state. Since many states are impossible to reach, and will not be visited (since it depends on enemy behavior to generate all states), we increased the canonical representation to 15. Given the fact that we now know the exact number of bits to represent our compressed state, we model an auto-encoder that compress the state representation to a 15 bits representation. Having defined the canonical representation size, and the raw encoding, we must define the auto-encoder architecture.

We develop an auto-encoder to compress the 156 bit representation into a smaller representation. To test the canonical encoding, we developed an auto-encoder with the following layer architect architecture:

$$156 \rightarrow 160 \rightarrow 100 \rightarrow 50 \rightarrow 25 \rightarrow 15 \rightarrow 25 \rightarrow 50 \rightarrow 100 \rightarrow 160 \rightarrow 156$$

where each number represents a layer and the number of the nodes in this layer, and 15 is the size of the canonical encoding. To train the network, we fed approximately 15000 states using the raw encoding. These states were generated from matches from two random AIs. We used random AIs matches because they are able to provide a much greater variety of states.

# 4. IMPLEMENTATION AND EXPERIMENTS

## 4.1 Separate Roles

In the MicroRTS the worker unit has the ability to harvest resources and deliver them to the players base. However, this unit has also the ability to attack. Since the worker is the only unit the player can produce without constructing a barrack, it is a good unit for both harvesting and attacking. This means we can have workers performing different roles. If this strategy is to be followed, we must have workers performing two distinct tasks: harvesting resources, attacking the enemy base. We call them *harvesters* and *attackers* respectively.

In the previous section, we explained how different units have different Q-Tables. The same applies to different roles. Since both attackers and harvesters units have completely different tasks but are the same unit type, we must assign different Q-Tables for these units. The workers is the only unit that suffers from this problem, since other units have a only purpose. We could make workers only harvesters, and make the other units responsible for attacking. However, this could lead our approach extremely vulnerable to rush tactics, since there is a delay time to construct a barrack and produce stronger units.

Since we are designed two separated roles, we must design different reward functions for each of them. Each role must have a reward that teaches how to properly perform such role.

## 4.2 Experiments

Using the implementation described in the previous section we now evaluate our approach in terms of its ability to converge to a policy in the MicroRTS domain, and how competitive the resulting policy is. To evaluate our encoding, we test the ability to train both an attacker and harvester work. To test this, we use our AI against a tweaked passive AI. The passive AI does not execute any action, however, our tweaked passive AI produces one worker that moves randomly through the map. The idea of this worker is to force our approach to explore states where there are enemy units besides the enemy base. We limit the amount of workers our base can produce, to two workers of each type. In Figure 4, we present the convergence rate of both unit types. The values represent how much the sum of all values of the Q-Table has varied in one training step to another. As we can see, the variance spikes at some points, possibly because of a new state found with a very positive reward, or a very negative one.

In Figure 5, we shown q-table size as the number of matches increases. As we can see, both units have a very close q-table size once it converges.

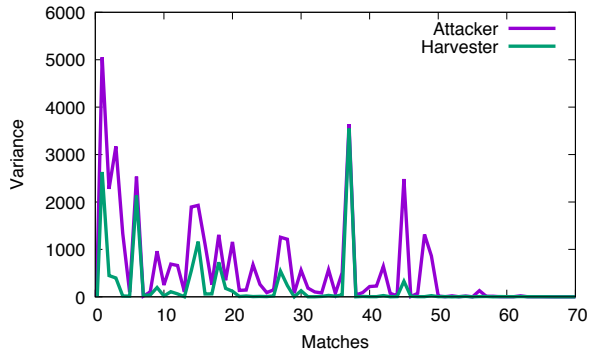


**Figure 4: Convergence of Attacker and Harvester**

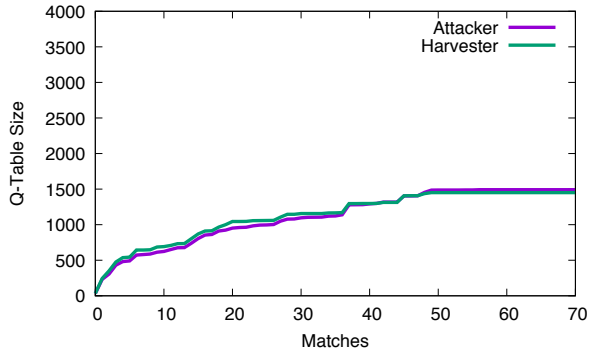


**Figure 5: Q-Table size of Attacker and Harvester**

To test if our approach is competitive, we test it against the following approaches:

- Passive: An approach that does not perform any action. We use it to test convergence time.
- Random: An approach that selects a random action for each unit.
- Random Biased: An approach that selects a random action for each unit. However, this approach prioritizes attacking and harvesting over the other actions.
- Heavy/Ranged/Light Rush: A hard-coded strategy that builds a barracks, and then constantly produces "Heavy", "Ranged" and "Light" military units to attack the nearest target (it uses one worker to mine resources).
- Worker Rush: This approach only uses worker units. One worker is assigned to harvest resources while the other workers attack. It is a very effective strategy due to the very high pressure it applies to opponents very early on in a match.
- Monte Carlo: An approach based on Monte Carlo tree search [7].
- NaïveMCTS: The approach built by Ontañón [7]. We use the default values set in MicroRTS.

To evaluate our approach, we play a number of games using our approach against each of the approaches above. In each match, we trained our agent against the opposing approach by playing 200 matches. After the 200 matches, we stopped learning new information and only followed the current learned policy. After training our approach, we evaluated 20 matches, analyzing wins, losses, draws, win rate and the score. The score is the number of wins minus the number of losses. All games were played in the standard 8x8 grid, the same used in all images of MicroRTS in this paper. Table 1, shows the results of our approach against each of these approaches.

**Table 1: Results against multiple AIs**

| AI | Wins | Draws | Losses | Win rate | Score |
|---|---|---|---|---|---|
| Passive AI | 20 | 0 | 0 | 100% | + 20 |
| Random AI | 20 | 0 | 0 | 100% | + 20 |
| Random Biased AI | 20 | 0 | 0 | 100% | + 20 |
| Heavy Rush | 20 | 0 | 0 | 100% | + 20 |
| Light Rush | 20 | 0 | 0 | 100% | + 20 |
| Ranged Rush | 20 | 0 | 0 | 100% | + 20 |
| Worker Rush | 9 | 4 | 7 | 45% | + 2 |
| Monte Carlo | 17 | 3 | 0 | 85% | + 17 |
| NaïveMCTS | 6 | 6 | 8 | 40% | - 2 |

As we can see in Table 1, our approach consistently outperforms all competing approaches besides Ontañón's NaïveMCTS, against which we lose slightly more often. Our AI constantly defeated all AIs that required building barracks, due to high early pressure using workers as attackers. Most draws were due to our workers were unable to follow a clear policy to destroy the remaining units after destroying the enemy base, and ended up dying fighting the remaining enemy units in single combat. Against NaïveMCTS, we were unable to train 200 matches due to the long time it takes to complete a game. We trained using 200 matches against the Worker Rush AI, and then 10 matches against the NaïveMCTS.

# 5. RELATED WORK

## 5.1 Distributed Reinforcement Learning

Q-Learning and SARSA algorithms ensure that they will eventually compute an optimal policy [15]. However, in some environments, this computation can take too long. In [4], Mnih develops a new reinforcement learning architecture, the Deep Q Network (DQN). Using a deep neural network to encode images as input, DQN was able to outperform a human professional in many Atari 2600 games. These images are directly extracted from the Atari games, with the purpose of feeding it to an agent capable of learning by only receiving game image as the input, similar to a human being. Training the DQN in a single machine took a long time, on the order of 12-14 days to train an agent using a GPU to play a single game.

To improve convergence time in complex environments, In [6], Nair describes a distributed architecture that enables to scale up DQN by exploiting massive computational resources. This architecture, called *Gorila* (General Reinforcement Learning Architecture), is composed of four main components:

- *parallel actors* consisting of agents responsible for performing new actions on the environments, generating new behavior;

- *parallel learners* consisting of agents trained from stored experience obtained from the Parallel Actors;
- a *distributed neural network* to represent the behavior policy; and
- a *distributed replay memory* to store the sequential acts of each Parallel Actor.

To speed up convergence, multiple agents are instantiated to act in multiple instances of the same environment. Each agent is given a slightly different exploration policy, to ensure that the agents explore different states, providing more useful data. After each episode, the data of each agent, called replay memory, is stored on a distributed database. With this procedure, more data is generated, due to the use of multiple agents, and the state space is explored more efficiently, since the exploration policies are slightly different.

The work described in [5] and [6] share many similarities with ours. We use deep neural networks to encode the state space. However, in the scope of our problem, we do not use the game image as state representation. Instead, we use a pure binary representation we built, using a reward function we designed. Additionally, we have multiple units as agents, which vastly increases the number of possible actions. Our approach is more centered around relaxing the problem to fit reinforcement learning in, than emulating the difficulties of playing with the same information a human player would have.

## 5.2 Multi-agent Reinforcement Learning

Multi-agent reinforcement learning (MARL) is a technique applied to environments where multiple agents interact with each other and the environment. Such environments are usually cooperative and include agents who have individualized perceptions and policies. In cooperative environments, the goal of reinforcement learning is to compute an optimal policy that coordinates the actions of every agent. As the number of agents increase, so does the number of possible combinations of actions. The main difficulty of applying reinforcement learning in multi-agent systems is the need to compute a global policy for all agents [2]. A global policy is a policy that provides actions for every agent simultaneously in each possible state. Computing a policy that takes account of every action of every agent can slow convergence substantially in environments with multiple agent, because the policy must compute every possible combination of actions.

In [16], each agent has its own policy, and coordinate to compute a global policy. Each agent applies its own Q-Learning algorithm, acquiring experience from acting in the environment, without communication or coordination. This is called independent learning [3]. With only independent learning, it is impossible to ensure that an optimal policy is going to be achieved. To illustrate the limitations of independent learning, Figure 6 illustrates a target tracking problem consisting of four sensors. Each sensor can scan in one of the four cardinal directions: North, South, West, East. The objective is to track targets in one of the locations in Figure 6. Tracking a target in each location has a reward. For location1 and location3 the reward is 40, and for location2 the reward is 60. However, to track a target, two sensors must be scanning the same area simultaneously. If location1, location2, and location3 always have targets to be tracked, then, by using the independent learning approach, sensor2 and sensor3 will potentially learn to sense location2, which has average expected reward is 60. However, the optimal policy is that sensor1 and sensor2 always sense location1 and sensor3 and sensor4 always sense location3, whose global expected reward is 80. Therefore, without the coordination of the sensors is not possible to ensure an optimal policy.

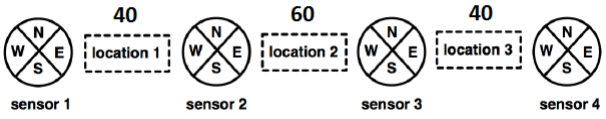


**Figure 6: Tracking problem.**

The MicroRTS case study properly fits the idea of this work. However, we do not deal with coordinating multiple agents. All agents have the same goal, and each agent type have it's own reward function. Our goal is not to coordinate agents with different goals, but rather use techniques to enable the use of reinforcement learning techniques, by compressing the Q-table.

# 6. CONCLUSION

In this work, we developed a set of techniques to reduce the number of entries in the Q-table of the Q-learning algorithm. Recently, much research has been done to use deep neural networks to improve the performance of reinforcement learning algorithms. However, most such efforts [5, 4], focus on transforming simpler domains in extremely complex domains using only the image as agent perception, increasing the similarity of how humans learn to act in such domains. Our approach, however, focuses on a very complex domain using all information available from the game, transforming a very complex domain in a simpler one.

We believe we achieve promising results, at a substantially lower computational cost, as our AI was competitive throughout all of the tested opponent strategies. Although, our approach requires training while others do not, the response time of our learned agent was much faster than the approaches that did not follow a hard-coded strategy.

We managed to use reinforcement learning in a very complex domain using two approaches that mitigate the problem of a large state-space combined with a combinatorially exploding number of actions due to multiple concurrent agents. Both approaches could be used in other scenarios. The auto-encoder can be used in any Q-learning scenario. And the unit Q-learning, requires a multi-agent scenario where there are roles defined for each agent.

For future work, we would like to address the following problems:

- Use denoising stacked auto-encoders.
- Learn the reward function.

Our auto-encoder could be improved by using a denoising stacked auto-encoder. Our auto-encoder is very simple, and it is possible we would be to achieve better results if we used a different type of auto-encoder. Furthermore, it would interesting to use the image of the game as a state

representation. However, this is a very challenging task due to the high dimensionality of the MicroRTS game image.

In our work, we currently design a complex reward function for each of the roles. This reward function can only be crafted because we have a good understanding of the MicroRTS domain. If we want to apply this set of techniques to other domain, we must design new reward functions, that in some domains can be even more complex. To avoid build a new reward function, we could use inverse reinforcement learning techniques to learn the reward function. Inverse reinforcement learning is a study that focus on learning the reward function of a domain by watching an agent perform in such domain. In the MicroRTS case, we have many computer controlled players already competitively playing the game. Our own reinforcement learning AI became very similar to the worker rush AI. We could use the worker rush strategy to extract a reward function. However, using a single reward function for both the harvester and the attacker can be a challenge. Retrieving a reward function from such scenario would be difficult challenge.

## REFERENCES


[1] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, Dec. 1989.

[2] C. Guestrin, M. G. Lagoudakis, and R. Parr. Coordinated reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pages 227–234, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[3] J. R. Kok and N. Vlassis. Collaborative multiagent reinforcement learning by payoff propagation. *J. Mach. Learn. Res.*, 7:1789–1828, Dec. 2006.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

[6] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015.

[7] S. Ontañón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In G. Sukthankar and I. Horswill, editors, *AIIDE*. AAAI, 2013.

[8] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.

[9] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[10] G. Tesauro. Practical issues in temporal difference learning. In *Machine Learning*, pages 257–277, 1992.

[11] G. Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, Mar. 1995.

[12] M. Tokic. Adaptive -greedy exploration in reinforcement learning based on value differences. In *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence*, KI'10, pages 203–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, New York, NY, USA, 2008. ACM.

[14] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[15] C. J. C. H. Watkins and P. Dayan. Technical note: q-learning. *Mach. Learn.*, 8(3-4):279–292, May 1992.

[16] C. Zhang and V. Lesser. Coordinating multi-agent reinforcement learning with limited communication. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, pages 1101–1108, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.

[17] J. Zhang and C. Zong. Deep neural networks in machine translation: An overview. *IEEE Intelligent Systems*, 30(5):16–25, 2015.