PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF COMPUTER SCIENCE
POSTGRADUATE PROGRAMME IN COMPUTER SCIENCE

Q-TABLE COMPRESSION FOR REINFORCEMENT LEARNING

LEONARDO ROSA AMADO

Dissertation presented as partial requirement for obtaining the degree of Masters in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Felipe Meneguzzi

Porto Alegre
2018

REPLACE THIS PAGE
WITH THE
PRESENTATION TERM

MSC - Q-TABLE COMPRESSION FOR REINFORCEMENT LEARNING

RESUMO

Algoritmos de aprendizado por reforço são usados para computar agentes capazes de agir em ambientes onde não se tem conhecimento prévio. Porém, esses algoritmos não são capazes de convergir em ambientes onde o espaço de estados é muito grande com possível grande fator de ramificação. Nessa dissertação, desenvolvemos técnicas para comprimir o espaço de estados usando um auto-encoder. Para lidar com grande fator de ramificação, apresentamos técnicas capazes de reduzir este fator. Como aplicação, usamos um jogo de estratégia em tempo real, onde se tem um grande espaço de estados e um grande fator de ramificação. Nós mostramos que algoritmos tradicionais de aprendizado por reforço não são capazes de competir nesse tipo de ambiente. Nós detalhamos como implementar as técnicas desenvolvidas e discutimos trabalhos futuros.

Palavras Chave: Aprendizado por reforço, Espaço de estados, Auto-encoders.

Q-TABLE COMPRESSION FOR REINFORCEMENT LEARNING

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 dissertation, 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 present the application of 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.

Keywords: Reinforcement Learning, State space, Auto-encoders.

LIST OF ACRONYMS

RL	Reinforcement Learning
RTS	Real-time Strategy
ANN	Artificial Neural Network
DNN	Deep Neural Network
DBN	Deep belief Network

LIST OF ALGORITHMS

LIST OF FIGURES

LIST OF SYMBOLS

LIST OF TABLES

CONTENTS

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 [25]. 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 [26].

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 [29] 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.

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 fail to converge due to the large state and action space. Thus our main contribution is a novel technique for dealing with large branching factors in a multi-agent scenario, and the application of deep auto-encoders to encode a binary representation. The usage of an auto-encoder in reinforcement learning is not novel on its own, however using it to compress a large binary state-space instead of the raw video signal distinguishes our work from existing approaches. 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 [19]. Ultimately, 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.

This dissertation is structured as follows. First, we discuss the necessary background to understand this paper. Second, we detail our problem as a reinforcement learning problem. Third, we explain in detail our approach. Fourth, we explain our implementation and demonstrate the experiments we made. Fifth, we discuss related work to our approach. Finally, we conclude our work and discuss the future of our work.

2. MACHINE LEARNING

In this chapter, review the basics of machine learning. First, we provide a quick overview of machine learning types. Then, we detail neural networks. Finally, we quickly summarize Deep auto-encoders.

2.1 Machine Learning

Machine learning studies the construction of algorithms capable of building models from data [1]. Mitchell provides a formal definition of machine learning as follows [13]: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E." There are three types of machine learning algorithm [24]:

- Supervised Learning.
- Unsupervised Learning.
- Reinforcement Learning.

In supervised learning, the objective is to infer a function or model using training data labeled with the expected output given a set of input features [16]. Features in machine learning, are selected properties of the data used to represent the data. For example, the features of a pixel could be the RGB values for that pixel. Training examples are conjunt of data using the features of a problem. Using the training examples, a supervised learning algorithm produces a hypothesis function. After training, the learned function should be able to compute the expected output given a new set of feature values never seen by the learning algorithm. For example, suppose we are trying to learn a function capable of predicting how much a house costs, based on the size of the house. To train the algorithm, we use a few examples of house pricing, containing the price and the size of the house. In this example, the size of the house is our feature value, and the price of the house is the output we want to predict. We train the supervised learning algorithm using the examples of house pricing provided. The algorithm learns a function capable of predicting the house, based on its size.

The idea of unsupervised learning is to infer a function to describe a hidden pattern on unlabeled data. Similar to supervised learning, in unsupervised learning the algorithm is given a training data set. However, the data in this data set does not have an expected output. An usual task of unsupervised learning, is to cluster the data into different categories. For example, suppose we are given a data set containing multiple instances of a category of flower. Each instance contains the characteristics of a flower, such as color, size and number of petals. All these flowers are currently set in the same category. However, there are many differences between the flowers in this category, leading to the idea of dividing this category into two new sub-categories. Since the instances are not classified using these new sub-categories, we apply an unsupervised learning algorithm to cluster the instances into two sub-categories. After executing the algorithm, the instances are clustered and we know which flower belongs to which sub-category.

Unlike supervised and unsupervised learning, in reinforcement learning no training set data is given. Usually, the environment of reinforcement learning is a Markov Decision Process, where an agent acts an gather signals from the environment. The objective in reinforcement learning, is to maximize the expected utility of acting in an environment by experimenting the possible actions in this environment.

Machine learning tasks do not differ only based on the input given, but also on the desired output for each task. Some of the categories based on output are [4]:

- in a **classification task**, the algorithm must determine which class an instance belongs to—the possible classes are already known beforehand;
- a **regression task** is similar to classification, but instead of a discrete value, the algorithm tries to find a function that maps an input to a continuous value;
- in a **clustering task**, the objective is to find a hidden pattern in a set of inputs, dividing those inputs in multiple classes–those classes are not known beforehand;
- in a **density estimation task** the objective is to find the distribution of inputs in some space; and
- in a **dimensionality reduction task**, the objective is to simplify inputs to a smaller representation, storing almost the same amount of data.

There are many algorithms of machine learning to deal with each one of these task. One of them, are the neural networks.

2.2 Neural Networks

In this section, we explain briefly neural networks and deep auto-encoders. First, we provide a brief background of Artificial Neural Networks (ANN). Then, we explain in more depth a type of neural network used to find efficient encodings for data, called deep auto-encoder.

2.2.1 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 input. The nodes combines input from the data with a set of coefficients (or weights) that amplifies our suppress an input. An ANN comprises three types of layers [6]:

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. The hidden layer is not obligatory. There are only hidden layers in networks with 3 or more layers.

These layers are interconnected, sending information from one layer to another, as shown in Figure 2.1. The inter-layer connections have weights. These weights affects the values that one layer sends to the other, changing the impact of the value on the next layer.

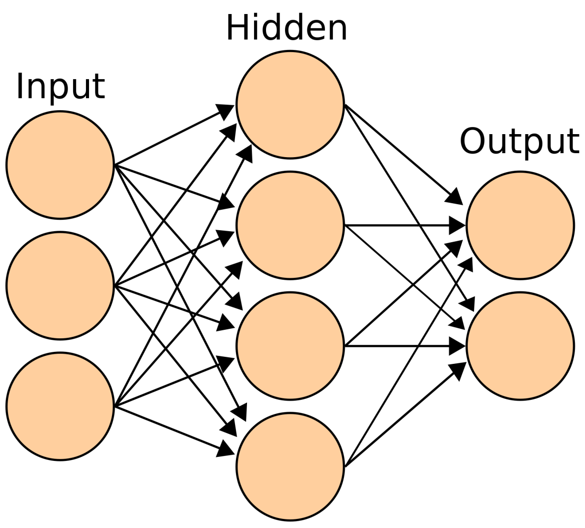

Figure 2.1 – Neural network with three layers.

Artificial Neural Networks solve multiple machine learning task problems, such as classification, regression and dimensionality reduction. For example, in the case of a classification task, a neural network will train using provided training data, adjusting the weight of connections to properly predict the class of an input data. As the weights adjusts, the neural networks builds a model to represent the data and predict next inputs. 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 autoencoders, convolutional neural networks and deep Boltzmann machines [32]. These types are defined based on the architecture of these networks. Most networks are trained using the back-propagation method.

2.2.2 Back-propagation

Backward propagation of errors (back-propagation), is a common method for training neural networks [22]. The idea of back-propagation is that by feeding known input values, which we known the desired output, we can use the error between the network's output and the expected output to modify the network weights. Since back-propagation requires inputs where the expected output is known, it is considered a supervised learning algorithm.

To use back-propagation, first the network must have its weights initialized. Usually, the weights are chosen at random. After initializing the weights, we can divide back-propagation in four steps[5]:

- Forward propagation.
- Back-propagation of errors.
- Assigning blame to the weights.
- Weight update.

The first step, forward propagation, consists of using an input x_i with expected out y_i, and feeding it through the network using the actual weights. In the first iteration of the algorithm, the weights are random initialized values. The network then produces an output, based on the input given. This output then is used in the back-propagation step.

After generation an input in the forward propagation step, the output of the network is compared with expected output y_i. Comparing the two outputs, we can calculate the error using a loss function. After calculating the error between the output layer and the desired output, the error values are then propagated backwards, starting from the output layer. The process continues until each neuron has an error value which approximates its contribution to the original output.

The last two steps are assigning the blame to the weights and weight update. To assign the blame, the error derivatives for each weight are calculated by combining the input to each node and the error signal for the node. In the weight update phase, the weights are updated in a direction that reduces the error derivative (error assigned to the weight), metered by a learning coefficient. The learning coefficient dictates how fast the nodes learns.

The network is trained using a training set with multiple entries. The procedure described is applied to each one of the entries in the data set. At the end of the training set, the network's weights are capable of better predicting new entries of this model.

2.2.3 Deep auto-encoder architecture

Deep auto-encoder is a type of deep neural network capable of converting inputs to a more compact representation of itself [29]. 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.

We illustrate the architecture of an auto-encoder in Figure 2.2, 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.

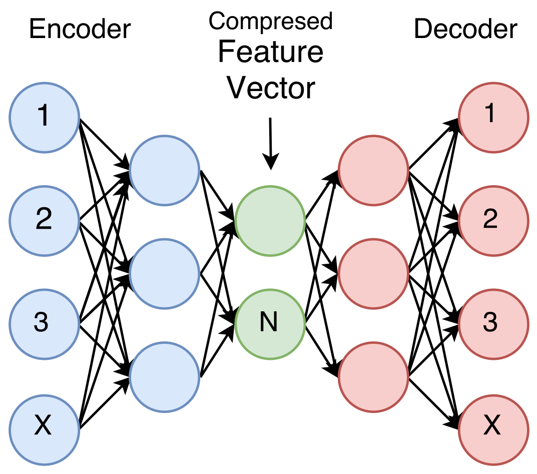

Figure 2.2 – Deep auto-encoder architecture.

Suppose the encoding of an image that has 784 pixels (a 28x28 image). On this architecture of Deep auto-encoders, we are only considering one input signal, so the image has to be on gray scale. 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 2.2.

3. REINFORCEMENT LEARNING

In this chapter, we review key background on reinforcement learning required by the contribution of this dissertation. First, we provide an quick overview of Markov Decision Process basics. Then, we introduce the q-learning reinforcement learning algorithm. Finally, we discuss generalization in reinforcement learning.

3.1 Markov Decision Process

Markov Decision Process (MDP) is a mathematical framework used to model decision making in environments where the outcomes are not entirely controlled by the agent. MDPs are composed of actions, states and rewards [21]. Actions are responsible for transitioning the environment from one state to another. States are a representation of the environment state. The reward is a numeric value that represents how desirable a state is. MDPs can be formalized by three main components [24]:

- A initial state S_0
- A reward function $R(s)$ that maps the states of the environment to a reward.
- Transition function $T(s, a, s')$ that defines the outcome, s', of an action a in the state s.

Within an MDP, an agent may choose an action a in a state s and the MDP randomly transitions to a state s' with probability $P(s'|s, a)$ given by the transition function $T(s, a, s')$. The agent advances a step by executing the action a, leading to a new state s', and returning a reward for achieving state $R(s')$. The reward function is a function that maps any state to a numeric value, that represents how desirable such state is. The transition function $T(s, a, s')$ returns a probability transitioning from state s to s', since in some MDP models actions do not always yield the same results, hence it is possible to achieve different states executing the same action in the same state.

Solving an MDP consists of finding a *policy* [2]. A policy π is a function that, given a state, returns the action to be executed in that state. A policy that always returns the action that provides the highest expected reward for each state is called an *optimal policy*, denoted by π^*. To calculate how good a state is, we must measure the utility of the state. The utility of a state represents the total reward obtainable in the future, if starting from this state. While the reward function represents how good a state is immediately, the utility represents how good the state is after visiting all reachable states and their rewards, within a certain temporal horizon. To calculate the utility of a state, we use the Bellman equation, defined in [2] and shown as Equation 3.1.

$$U(s) = R(s) + \gamma max_a \sum_{s'} T(s, a, s')U(s') \tag{3.1}$$

where γ represents a discount factor, a value that determines how valuable are future rewards. When γ is 0, rewards obtained in the future are irrelevant, rewarding policies with immediate rewards. When γ is 1, future rewards are fully considered, not penalizing policy with future rewards.

By calculating the utility of each state, it is possible to choose an action in a non-terminal state that returns the state with the highest utility. The process of extracting a policy based on the computed utility values is defined by Equation 3.2, below.

$$\pi^* = argmax_a \sum_{s'} T(s, a, s')U(s') \tag{3.2}$$

where a is a possible action in a given state s. With this equation, it is possible to calculate the utility of each state when choosing the best action. Instead of considering the set of every future states, we consider only the immediate following state. Then, the utility of a state is the immediate reward for that state plus the discounted utility of the next state, assuming that an optimal actions is chosen [24]. This equation is called Bellman equation [2]. With this equation, it is possible to solve an MDP. The Bellman equation is the basis for multiple algorithms that acts on MDPs. There are reinforcement learning algorithms that use the Bellman equation as basis to act and find optimal policies on MDP environments.

3.2 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:

$$U(S) = max_a Q(a, s) \tag{3.3}$$

we use equation 3.3 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:

$$Q(a, s) = Q(a, s) + \alpha(R(s) + \gamma max_{a'} Q(a^{'}, s^{'}) - Q(a, s)) \tag{3.4}$$

were α is the *learning rate*, which ranges between 0 and 1. When 0, nothing is learned, and when 1, the learned value is fully considered. γ 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 3.4 implements the Q-Learning update as part of Algorithm 1, based on the specification from [**?**].

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 [28], 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.

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 $a$
```

The SARSA (State-Action-Reward-State-Action) [23] 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:

$$Q(a,s) = Q(a,s) + \alpha(R(s) + \gamma Q(a^{'}, s^{'}) - Q(a,s)) \tag{3.5}$$

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. This behavior makes SARSA an on-policy algorithm, unlike the Q-Learning algorithm, which is off-policy. This means that SARSA learns the Q-Values associated with following the policy itself, while Q-Learning learns the Q-Values associated with following the exploitation/exploration policy.

3.3 Generalization in reinforcement learning

Reinforcement learning can never converge when dealing with huge state-spaces. Even after some training, if the basic Q-learning algorithm is executed and a new Q-value is found, the algorithm will not use any of the information to deal with the new state. This problem comes from the fact that reinforcement learning algorithms, we saw so far, are not able to generalize information from the domain. By generalizing, we mean estimating the utility of a state, by using information of similar states. If a state is constantly explored and a new state with a minor difference is found, the algorithms no way of estimating the utility of such state.

Function approximation can be used to generalize information from the state values. By using other representation method for the Q-function, instead of the Q-Table. The representation is an approximation because with only a linear function it is impossible to guarantee that the function represents the true utility function [24]. In a scenario with *n* features, the algorithm must learn a weighted linear function of the set of features $f1, ..., fn$:

$$\hat{\mathrm{U}}_\theta(s) = \theta_1 f_1 + \theta_2 f_2 + ... + \theta_n f_n \tag{3.6}$$

where $\hat{U}$ is the estimated utility. A reinforcement learning algorithm can determine the weights for the parameters θ, approximating $\hat{\mathrm{U}}$ to the real utility function. Instead of having a large number of values on a table, we have a function based in a much smaller number of parameters. With this function, an agent is capable of approximating the utility of a state it has never visited.

To apply this concept to reinforcement learning, we must adjust the parameters based on the real utility after each try. To do so, we use an error function to determine how much we must change each parameter. We define $u_j(s)$ as the observed total reward from state s onward in the jth trial, then the error is the half of squared difference of the predicted utility and the actual utility:

$$E_j(s) = (\hat{\mathrm{U}}_\theta(s) - u_j(s))^2/2 \tag{3.7}$$

For the linear function approximation $\hat{\mathrm{U}}_\theta(s)$, we have a simple update rule for each parameter:

$$\begin{aligned}
&\theta_0 \leftarrow \theta_0 + \alpha(u_j(s) - \hat{\mathrm{U}}_\theta(s)),\\
&\theta_1 \leftarrow \theta_1 + \alpha(u_j(s) - \hat{\mathrm{U}}_\theta(s))f_1,\\
&\theta_n \leftarrow \theta_n + \alpha(u_j(s) - \hat{\mathrm{U}}_\theta(s))fn.
\end{aligned}$$

Finally using Equation 3.6 and the update rule described above, it is possible to derive an update rule based on the Q-values of a domain. This update rule can be written as:

$$\theta_i \leftarrow \theta_i + \alpha(R(s) + \gamma max_{a'}\hat{Q}(a', s') - \hat{Q}(a, s))f_i \tag{3.8}$$

With this equation, it is possible to generalize Q-values, avoiding the need to visit all states. However, a strict state representation must be made to convert it to a linear function.

4. MICRORTS

In this chapter, we introduce the concepts necessary to understand the MicroRTS game. First, we provide a quick overview of what is a Real-time strategy game. Then, we introduce the details of the MicroRTS game. Finally, we formalize the MicroRTS problem as a reinforcement learning problem.

4.1 Real-Time Strategy Games

Real-time Strategy (RTS) games are complex domains that try to simulate a scenario of war between two or more factions. Normally played by two adversary players RTS games are complex because the huge amount of units to control and large amount of actions to take in each moment. In RTS, both players can perform actions simultaneously, rather than, acting taking turns as modeled by most computationally-studied games. For example, it is possible to issue an order to multiple units at the same time, creating a combinatorial explosion of action choices at every decision point. Due to these characteristics, building AIs for those games is a well known challenge. Modern games, such as Starcraft, are being subject of study to create complex AI implementations.

In most RTS games, the player starts with a base, a supply of resources (minerals, trees, gas), and workers that can harvest resources and build structures. The main objective of an RTS game is to destroy the enemy base, leaving the enemy with no way to rebuild its base. Starting with only workers, the objective is to build an army. Workers can build structures, like barracks, that can train military units to fight. When a player is ready to fight, he sends his units to find the enemy base. In Starcraft, for example, there are a vast number of buildings for each of three factions that can build different units, and allow workers to create other buildings. The possibility of units is vast, and there are multiple ways of playing the game.

Due to the complexity of Starcraft, a strong competitive scene was built around it. Starcraft was launched on March 31, 1998, and its competitive scene survived through 12 years, when a sequel was released. This game is still one of the most used subjects of study for developing AI for RTS games, and is still a remarkable challenge. However, due to the complexity of the game, testing theoretical ideas can be troublesome and time consuming.

MicroRTS was created with the intent to be a simple tool to test theoretical ideas before implementing them in full-fledged RTS games, like Starcraft. Programmed in Java, MicroRTS is simple, but preserves the main components of an RTS game.

4.2 MicroRTS

MicroRTS is a simple implementation of a Real-time Strategy game, designed for the sole purpose of AI research. Developed by Ontañón [19], 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.

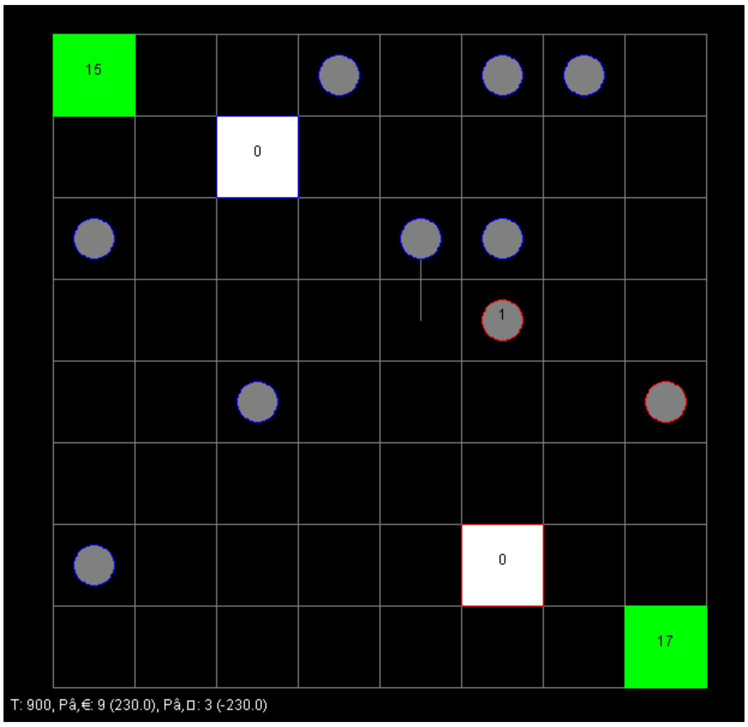

Figure 4.1 – MicroRTS game state.

MicroRTS consists of two players trying to eliminate every single structure and unit of the enemy. Figure 4.1 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.

4.3 Problem specification

The goal of a Reinforcement learning algorithm is to compute an optimal policy $\pi^* : S \rightarrow A$ that maps the current states s to an action a to be performed in s. Reinforcement learning algorithms, specially Q-Learning, are well known for their ability of retrieving optimal policies in environments with no previous knowledge. However, reinforcement learning algorithms converge very slowly in complex environments. This usually happens because the state-space of the environment is large and the number of possible actions is also very large.

Reinforcement learning problems can be modeled as a 4-tuple $\langle S, A, T, R \rangle$ [11]. To transform the MicroRTS scenario to a reinforcement learning problem, we must define three components:

- A representation of the MicroRTS states S.
- A set of possible actions A the player can perform in state s.
- A reward function R, that returns a numeric value to each state in S.

In the scenario of MicroRTS, a possible representation of a state is to list every unit, their position, their health and the player resources. Units can carry resources and have different types. The state must represent both the player units and enemy units. A player unit can be represented as pu_i and enemy units are represented by eu_i, where i is the unit id. Each unit is on a position of the map, so the state representation must account every tile of the map. This can be represented by xy_{pi}, where xy is the position in the grid, p is the player and i is the unit id. The unit can have different types, but this may not be necessary, since if dealing with a enemy unit we expect our unit to attack. Tracking every unit health can be hard to represent in a binary state representation, so it is best to only track the base health, as pb_{hp} and eb_{hp}, where hp is the number of health points of the base. The state representation must account the resources of each player, which can be done using p_{re} and e_{re}, where re is the amount of resources the player has.

Having defined the state space of the problem, we must define the possible actions of each state. In the scenario of MicroRTS, the possible actions of a state can be represented as a list, containing each possible action for unit of the player. Each unit in MicroRTS has in the worst case 9 possible actions in a state. We assume all states have at maximum 9 possible actions. A possible action a of the set of actions A, is the union of each action applied to each player unit, denoted as $\hat{a}_{ijp}$, where $\hat{a}$ denotes an unit action, i is the unit type, j is the unit id, and p a possible action applicable to this unit. An action a in a state with 3 units can be defined as $a = [\hat{a}_{0jp}, \hat{a}_{1jp}, \hat{a}_{2jp}]$.

The last component to be defined is the reward function. A reward function R should map any state s to a numeric value r, which represents how desirable is to achieve such state. On the MicroRTS scenario, defining a reward to each state can be a challenging task. To do so, we modeled the reward function as $R(s, a, s')$, where s' is the actual state, s the previous state and a the previous action. With the information of the previous state and action, we can define how a single agent should be rewarded, based on the impact of its actions. Looking at the previous and actual state, we can determine if it was the agent actions responsible, for example, killing a certain unit.

4.4 Discussion

In this chapter we introduced the basic concepts to understand an RTS game and the MicroRTS. The MicroRTS may be a simplification of modern RTS games, however, it stands as a challenge to develop reinforcement learning techniques to competitively play the game. During our tests, we found states where there were 3,500,000 (three million and five hundred thousand) possible actions. Only to test every possible action of this state, it would be necessary the same number of Q-Table entries. This alone would turn a classic Q-Learning implementation unfeasible in such scenario.

5. APPROACH

In this Chapter, we discuss our approach in two steps. First, we introduce the idea of applying q-learning individually to each agent. Second, we explain the architecture of the auto-encoder we built and the state representation used.

5.1 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 5.1, 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.

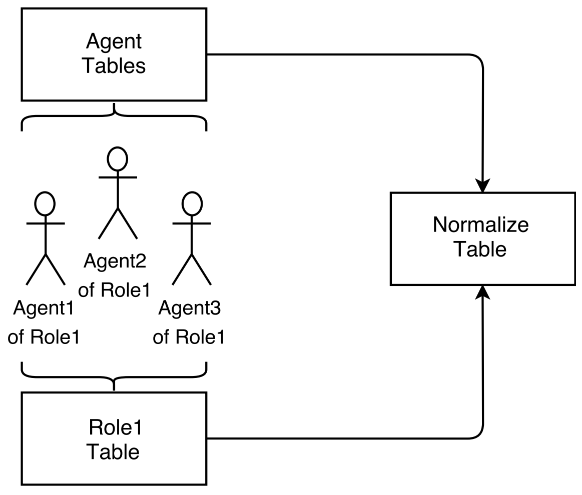

Figure 5.1 – Unit Q-learning Diagram

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:

$$Q(s,a) = \frac{\sum_{i=0}^{agents} Q_i(s,a) * frequency(Q_i(s,a))}{frequency(Q(s,a))} \tag{5.1}$$

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* section assigns the Type table to each unit. Furthermore, in this *for* we define the action for the state by selecting each individual unit action. In the second *for* section, 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;
$Action = \emptyset$ ;
for unit $u \in U$ do
    if $u.QTable = \emptyset$ then
        $u.QTable := QTables$(u.type) ;
    end
    $Action.add(QLearning$(u,s)) ;
end
if isTerminal(s) then
    for $type \in U$ do
        $Q := mergeTables$(type) ;
        $QTables$(type) := $Q$ ;
    end
end
return $Action$
```

The worker unit of the MicroRTS game can harvest 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 Chapter 8.

5.2 State encoding

Our approach consists of 3 steps. First, we design a binary representation for the state space we would traditionally store in the Q-table. We define this as *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. These 156 bits are based on the formal description we provided in Section 4.3 The 64 first bits are designated to represent the player units position. Each bit represents a position in the grid. Using Figure 4.1 as example, the following bit representation would represent 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 represent the enemy units, using the same concept. Furthermore, we must represent the health points of the units. Since it is hard to represent the health points of each unit, we decided 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 [27], 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. If we used competitive AIs, we would get a much smaller number of visited states, since these AIs tend to follow one single strategy.

In Figure 5.2, we illustrate the architecture of the auto-encoder we designed. We only illustrate the input layer, the output layer, and the layer responsible for the canonical encoding.

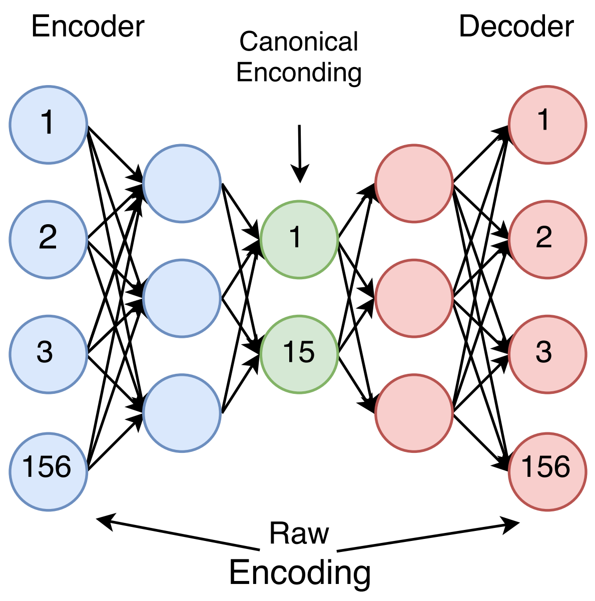

Figure 5.2 – Auto-encoder architecture.

5.3 Discussion

In this chapter, we explained the techniques used in our approach to effectively apply reinforcement learning in the MicroRTS scenario. Addition of the state encoding and the unit Q-learning, is what makes our approach competitively. It is important to note that the reward function is what forces a certain agent to follow a policy. In this work, we defined the a very specific reward function for each unit type. We could use simpler forms of reward function, but we wanted to avoid wrongful rewarding since we are using unit Q-learning. There is a set of techniques capable of calculating a reward function by observing other agent actions. This technique is called inverse reinforcement learning. We do not apply these techniques to our work, but we believe they could be applied to MicroRTS, since there is already many developed AIs. We propose inverse reinforcement learning as future work to extend our approach.

6. IMPLEMENTATION AND EXPERIMENTS

In this Chapter, we present the details of the implementation of our approach, and the experiments we executed in our approach. This chapter is divided in two main sections. First, a section discussing every implementation detail necessary to understand our approach. Second, each experiment we ran in our implementation and how we evaluated it.

6.1 Implementation

In this section we detail how we implemented our approach. First, we introduce the idea of separate roles. Then, we develop the necessary reward functions for Q-Learning.

6.1.1 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.

6.1.2 Reward function

To apply reinforcement learning to any scenario, we must design a reward function[12]. to translate events in the simulated environment into a numerical perception representing the desirability of an individual state of the game. Such reward function must meaningfully represent the relative desirability of all possible states in the environment. In the case of MicroRTS, the reward function must be an approximation of how close to victory the agent is.

However, since we are training each agent individually to reduce the combinatorial problem of possible actions, we must also design the reward for each agent individually. Suppose we design a reward function $R(s)$ that rewards the agent using the current state. In the Figure 6.1, suppose the units assigned by the red numbers 1 and 2 are attackers. Suppose the unit 2 in going to attack the red unit in front of it, killing it. In the next state, the enemy player has one less unit, so the next state is better than the previous state. The unit 2 will be rewarded for its attack. However unit 1 did not attack any unit. Since both units use the same reward function, both units are going to be rewarded for killing this enemy unit. With this reward, unit 1 will be rewarded erroneously, not learning how to properly act in the environment. To fix this issue, we must design a reward function $R(s, a', s')$, where a' represents the previous action, s' the previous state and s the actual state. The reward function rewards an unit based on the actual state and the last action this unit performed, avoiding erroneous rewarding an unit for an alteration in the state not performed by the unit.

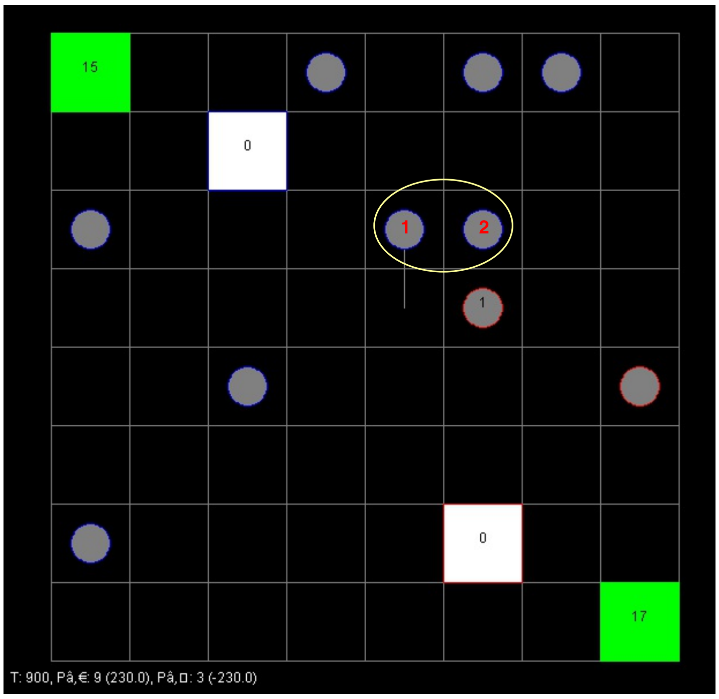

Figure 6.1 – MicroRTS reward example.

Using a reward function as $R(s, a', s')$, we develop three distinct reward functions. One reward function for the *Base* unit, responsible for producing other units. A reward function for the *harvester workers* and a reward function for *attackers*, which include workers and every other unit.

The objective of the harvester workers is to constantly harvest resources and return them to the base. To do so, we design the following reward function:

The attacker units must be constantly moving towards the enemy base and attack any unit found. We design the following reward for the attackers:

The objective of the base is to build workers. The base is responsible for managing resources. In Table 6.3, we detail the reward function we built to train the base unit. Since building

Table 6.1 – Harvester reward function

Situation	State	Previous Action	Previous State	Reward
Base reward	Any	Any	Any	-3
Harvest	Any	Harvest	Any	+10
Deliver resource	Have a resource	Deliver	Any	+10
Attack any unit	Any	Attack	Any	+5
Move towards resource	Distance to resource	Move	Distance to resource	+5
Move towards base	Distance to base	Move	Distance to base	+5

Table 6.2 – Attacker reward function

Situation	State	Previous Action	Previous State	Reward
Base reward	Any	Any	Any	-3
Attack any unit	Any	Attack	Any	+10
Killing an unit	N enemy units	Attack Base	N-1 enemy units	+10
Killing the base	No enemy base	Attack Base	Enemy base	+20
Move towards enemy base	Distance to base	Move	Distance to base +1	+5

a barracks costs 5 resources, the last line in the table represents a reward for storing resources. Without this specific reward, the base would never attempt to save resources to build a barracks, always producing new workers. However, further tests proved that a work rush approach is more efficient, and having a reward to force building a barracks slows convergence. These results and remarks are discussed in the next chapter.

Table 6.3 – Base reward function

Situation	State	Previous Action	Previous State	Reward
Base reward	Any	Any	Any	-3
Produce worker	Worker <1	Produce Worker	Worker <1	+20
Do not produce many workers	Worker >0	Produce Worker	Any	+10
Produce attackers	Any	Produce Attacker	Any	+15
Less units than the enemy	<units than enemy	Produce Unit	<units than enemy	+5
More units than the enemy	>units than enemy	None	>units than enemy	+15

In Figure 6.2, we display the behavior of each role learned using these reward functions. The yellow line represents the behavior of the harvester unit, and the red dotted line represents the behavior of the attacker units. The base behavior is not displayed since the base is unable to move.

6.1.3 Q-Learning Constants

In this section we define both Q-Learning constants, γ and α. The α constant sets the learning rate of the algorithm, and γ sets the discount factor. For gamma we use a constant of 0.8.

For alpha, we define a function based on the frequency that a state has been visited. We define this function as $e^{-(x-1)/5}$. We chose this function because we want the learning rate to

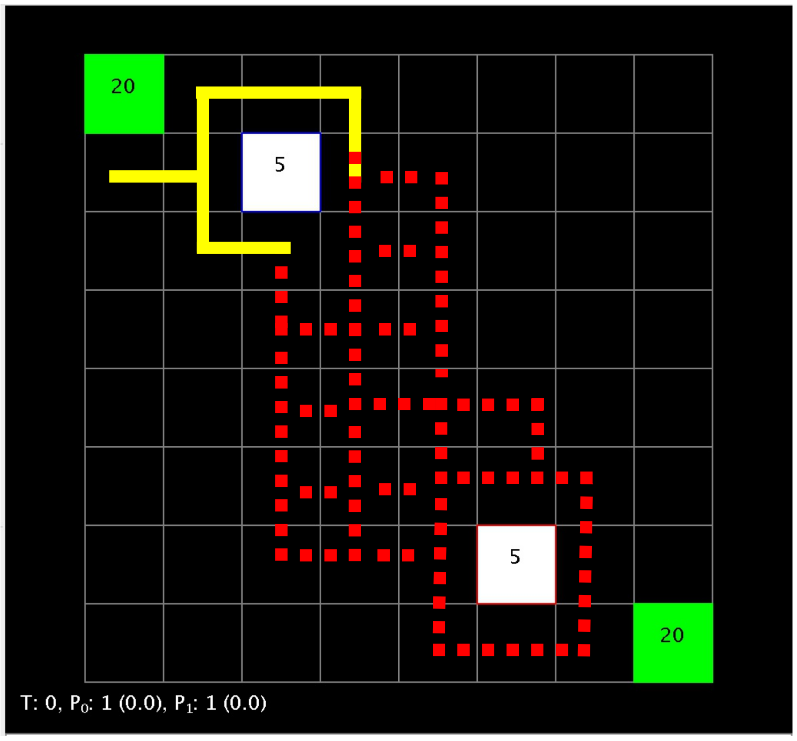

Figure 6.2 – MicroRTS movement trace.

decrease as the number of times a state has been visited grows. This causes the algorithm consider more information found in the earlier training tests. The alpha variation is shown in Figure 6.3.

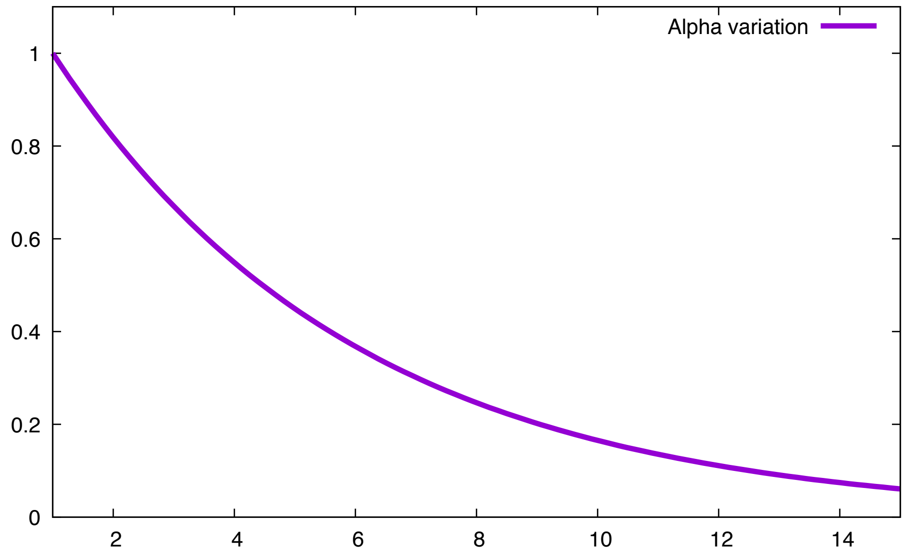

Figure 6.3 – Alpha Variation

6.1.4 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 6.4, 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.

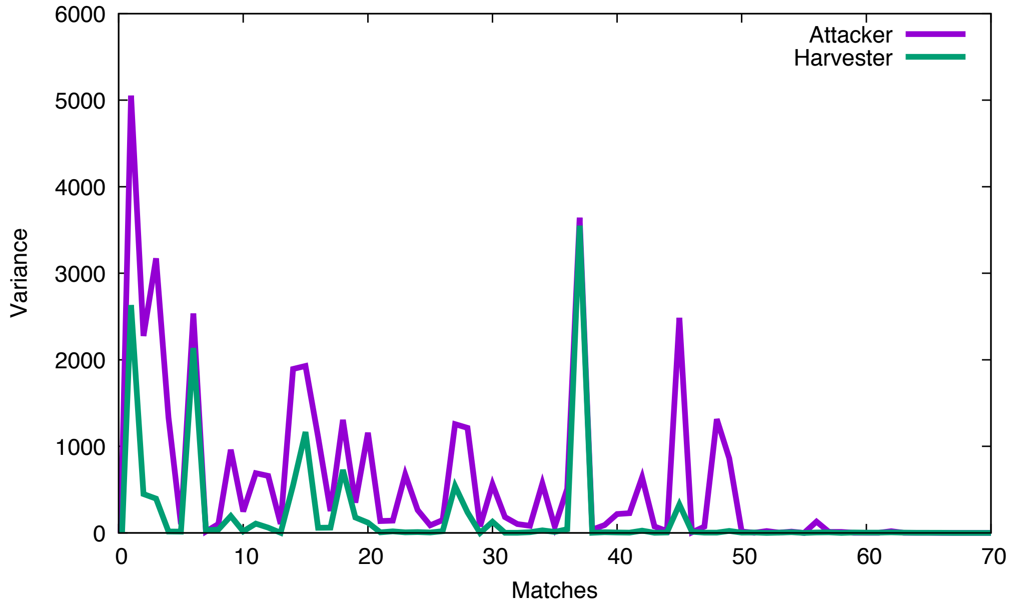

Figure 6.4 – Convergence of Attacker and Harvester

In Figure 6.5, 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.

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.

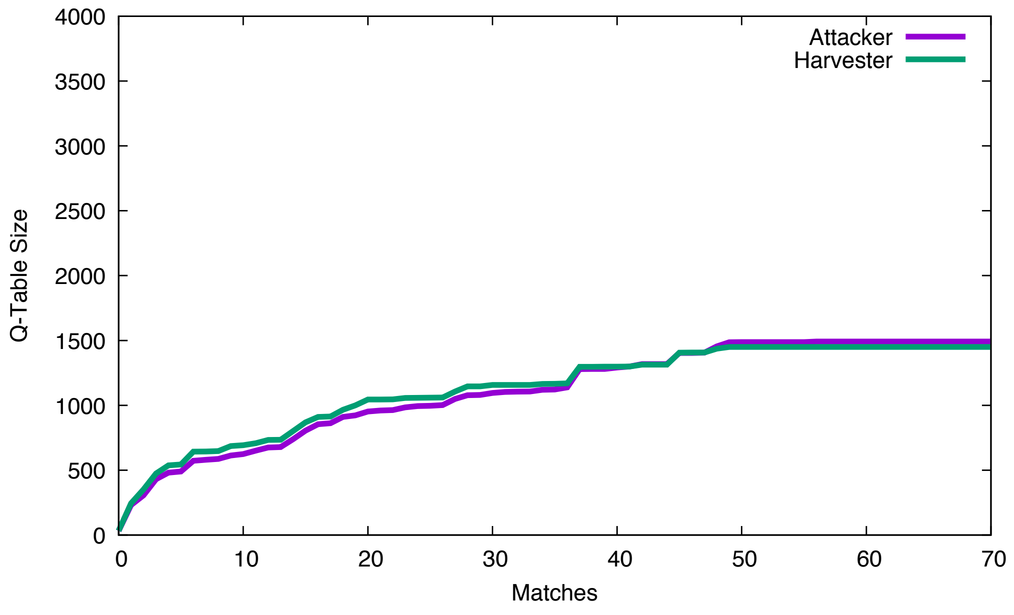

Figure 6.5 – Q-Table size of Attacker and Harvester

- 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 [19].
- NaïveMCTS: The approach built by Ontañón [19]. We use the default values set in MicroRTS.

To evaluate our approach, we perform five different testing scenarios. We train our approach against one specific strategy and then we use our trained AI to play against each of the available strategies. We use this train method for the following strategies: Passive, Random, Light Rush and Worker Rush. We do not train using Heavy Rush, Ranged Rush and Monte Carlo because they are very similar to Light Rush. Furthermore, Worker Rush behaves similar to NaïveMCTS, and acts much faster than NaïveMCTS, leading to shorter training times. Finally, we perform a scenario where we train against each strategy and face this strategy in a face-off match. 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 dissertation. Table 6.4 to Table 6.7 shows the results when training with only one approach. , Table 6.8 shows the results of our approach when trained with each approach and played against the same approach.

Table 6.4 – Results training with Passive.

Strategies	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	95%	+ 20
Heavy Rush	20	0	0	100%	+ 20
Light Rush	20	0	0	100%	+ 20
Ranged Rush	20	0	0	100%	+ 20
Worker Rush	0	3	17	0%	- 14
Monte Carlo	12	0	8	60%	+ 4
NaïveMCTS	0	2	18	0%	- 18

Table 6.5 – Results training with Random.

Strategies	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	4	1	15	20%	- 11
Monte Carlo	15	0	5	75%	+ 10
NaïveMCTS	0	1	19	0%	- 19

Table 6.6 – Results training with Light Rush.

Strategies	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	0	0	20	0%	- 20
Monte Carlo	17	3	0	85%	+ 17
NaïveMCTS	0	1	19	0%	- 19

As expected, in Table 6.4 to Table 6.7, we can see that our approach best perform when trained against the Worker Rush strategy. As we can see in Table 6.8, 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.

To further evaluate our metric, we analyze the response time for acting of our approach and every other AI strategy. We evaluate in two aspects, the average time the strategy takes to

Table 6.7 – Results training with Worker Rush.

Strategies	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	20	0	0	100%	+ 20
NaïveMCTS	4	7	9	20%	- 5

Table 6.8 – Results against multiple AIs

Strategies	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

return an action, and the max time during the 20 matches the approach took to return an action. In these tests, the opponent was always the Worker Rush strategy. We used the Worker Rush because it is a fast strategy that performs well. This should lead our approach to act slower, since the Q-Table will be bigger.

Table 6.9 – Time response for each approach.

Strategies	Average time (s)	Maximum time (s)
Passive AI	0s	0s
Random AI	∼0s	∼0s
Random Biased AI	∼0s	∼0s
Heavy Rush	0.001s	0.05s
Light Rush	0.001s	0.01s
Ranged Rush	0.001s	0.03s
Worker Rush	0.05s	0.1s
Monte Carlo	2.0s	2.303s
NaïveMCTS	2.0s	2.545s
Our approach	0.3s	0.511s

In Table 6.9, we show the response time of each strategy. As we can see, our approach was very fast and never surpassed the 1 sec of response time. Most approaches are nearly instant, except for the Monte Carlo and NaïveMCTS, that are slower than the other approaches. Combining the victory results with the time of response of our approach, we believe our approach is very competitive against the NaïveMCTS and the Worker Rush strategies.

7. RELATED WORK

In this chapter, we discuss and compare related work. First, we discuss work done on Distributed Reinforcement Learning. Second, we introduce work done Multi-agent reinforcement learning. Third, we summarize transfer learning. Finally, we discuss the reduction of state-space using deep learning.

7.1 NaïveMCTS

NaïveMCTS is the algorithm develop with MicroRTS in [19] to play RTS games. RTS games are domains where two players can simultaneously issue actions, affecting the state. Traditional adversarial algorithms, such as *minmax*, might underestimate or overestimate the value of a state because it acts using the assumption that each player acts on different turns. However, in [19], Santiago et al. empirically analyze the results of this assumption when dealing with MicroRTS, and it has small practical effect on the NaïveMCTS performance. So NaïveMCTS operates under the assumption that each player act on its own turn.

Furthermore, actions are durative in MicroRTS. This means an action can take more than a time step to complete, leading to states where neither player can issue an action. When expanding the search tree, NaïveMCTS takes into account durative actions and how much time steps each action takes, preventing from issuing an action when the unit is not available.

NaïveMCTS is designed to perform on environments that are deterministic and the zero-sum two player games, where one player acts as the *max* and the other player acts as the *min*. The *max* player must maximize a reward function R, while the *min* tries to minimize it. The difference of NaïveMCTS to standard Monte Carlo algorithms, is how NaïveMCTS expands the next node of the search tree. The process for defining the next node receives a node and define if it is a *max* or *min* node. After defining the type of node, the process must select one of the possible player action for this game state. To do so, NaïveMCTS uses naïve sampling to select of the possible actions of the game state. If the action is already expanded in the search tree, then the same process is applied to that node, going down the tree. Otherwise, a new node is created by computing the affect of the selected action a in the state s, until a decision point (a point where any of the players can issue an action).

Therefore, the two main differences of NaïveMCTS and MCTS algorithms is the naïve sampling, and dealing with durative actions. With these two approaches, NaïveMCTS is capable of competitively playing the MicroRTS.

7.2 Distributed Reinforcement Learning

Q-Learning and SARSA algorithms ensure that they will eventually compute an optimal policy [30]. However, in some environments, this computation can take too long. In [14], Mnih et al. 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 [17], 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.

Different from the parallel actors, the parallel learners read the stored replay memory, and update the policy, according to a given offline Reinforcement Learning algorithm. In [17], Nair uses a variant of the DQN, focused on distributed computation. After updating the policy, the Parallel Actors receive this policy, generating new exploration policies.

To prove the ability of learning faster, *Gorila* was compared to the DQN used in [14], that trained within 12-14 days. With 6 days of training, *Gorila* was able to obtain better results than the single machine DQN, outperforming it in 41 of 49 games. In [15], Mnih further extends this work, discussing the similarities of this architecture with ability of learning of the human brain.

The work described in [15] and [17] 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.

7.3 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 [7]. 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 [31], 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 [9]. 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 7.1 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 7.1. 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.

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.

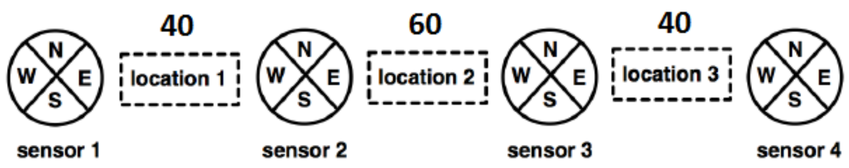

Figure 7.1 – Tracking problem.

7.4 Transfer Learning

Reinforcement learning algorithms are often slow due to the necessity of exploring the entire state space. When the domain is slightly changed, the algorithm requires a complete new training to adapt to those changes. The idea of transfer learning is that this complete re-training can be simplified, as knowledge acquired in a previous situation can be used as heuristics, speeding up the learning procedure on the new domain.

Transfer Learning can be defined as follows [20]: given a source domain (D_s) with its task to be performed on this domain (T_s), and a target domain (D_t) with its related task (T_t), transfer learning aims to improve the performance of learning the task's policy in the target domain, using knowledge obtained on learning the policy in D_s and T_s, where $D_s \neq D_t$ or $T_s \neq T_t$. Transfer learning algorithms can be summarized by two main characteristics [20]:

1. Which information is transferred. In the case of reinforcement learning, it could be the policy or the rewards, for instance.

2. How the information is transferred. This refers to which transfer learning algorithm is going to be used.

In [3], Bianchi describes the construction of a transfer learning algorithm based on SARSA. To transfer the knowledge, the actions of the source domain D_s are mapped to the actions of the target domain D_t. Using the Hebbian Rule [8], a Neural Network performs this mapping. The idea of this rule is that the weights that connect two neurons should be increased when their output is similar, and decreased when they are dissimilar.

Although transfer learning is essentially different than our work, there are a few similarities to our approach. Transfer learning aims to simplify computing a policy by using an already computed similar policy on a similar domain. What we do, is to use multiple sources of training information and merge them, accelerating the training of an agent.

7.5 Deep Learning state representation

To ensure that a reinforcement learning algorithm converges to an optimal policy, the algorithm must explore the entire state space. Most reinforcement learning algorithms are limited to solving tasks in which the state space has low dimensionality [18]. Recent research tries to address this challenge using deep learning [10] to find states that can be considered similar.

In [10], Lange introduces a framework for combining deep auto-encoders and RL algorithms. The framework receives images as input, and feeds the images to a deep auto-encoder. The encoder converts the images to a lower dimensional feature space, representing the encoded image. This representation is then fed to the reinforcement learning algorithm, that returns the best learned action for this representation. This process is shown in Figure 7.2.

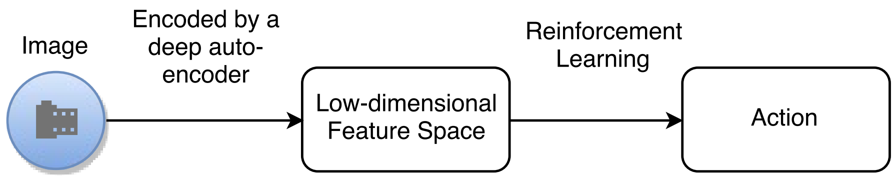

Figure 7.2 – Image encoding framework

Since the feature space is low-dimensional (2 features ranging between 0.0 and 1.0), the number of possible encodings is smaller than the number of possible images. This ensures that images that are too similar will be converted to the same feature space representation, avoiding creating a new state for images that have small noise captured by the camera.

We use a deep auto-encoder to encode our state space. Different from the works described in [10, 15], we do not encode high dimensional images. We design our own binary state representation.

7.6 Discussion

In this chapter we discussed related work to our approach. We believe what distinct our approach from other works, is that we aim to use reinforcement learning in a domain where it struggles to perform, rather than omitting information to increase the difficult of a domain. Much research has been done to apply reinforcement learning with only the information a human would have in simpler domains. Our work, however, uses every information available to learn how to act in the MicroRTS domain. With the auto-encoder, we aimed to solve the large state-space and generalization problems. With unit Q-learning, we aimed to solve the combinatorial problem of choosing one of the possible actions.

8. 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 [15, 14], focus on solving simpler 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.

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 a interesting challenge.

BIBLIOGRAPHY

[1] "Glossary of terms", *Mach. Learn.*, vol. 30–2-3, Feb 1998, pp. 271–274.

[2] Bellman, R. "A Markovian Decision Process", *Indiana Univ. Math. J.*, vol. 6, 1957, pp. 679–684.

[3] Bianchi, R. A.; Jr., L. A. C.; Santos, P. E.; Matsuura, J. P.; de Mantaras, R. L. "Transferring knowledge as heuristics in reinforcement learning: A case-based approach", *Artificial Intelligence*, vol. 226, 2015, pp. 102 – 121.

[4] Bishop, C. M. "Pattern Recognition and Machine Learning (Information Science and Statistics)". Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[5] Brownlee, J. "Clever Algorithms: Nature-Inspired Programming Recipes". Lulu.com, 2011, 1st ed..

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

[7] Guestrin, C.; Lagoudakis, M. G.; Parr, R. "Coordinated reinforcement learning". In: Proceedings of the Nineteenth International Conference on Machine Learning, 2002, pp. 227–234.

[8] Hebb, D. O. "The organization of behavior: A neuropsychological theory". New York: Wiley, 1949.

[9] Kok, J. R.; Vlassis, N. "Collaborative multiagent reinforcement learning by payoff propagation", *J. Mach. Learn. Res.*, vol. 7, Dec 2006, pp. 1789–1828.

[10] Lange, S.; Riedmiller, M. A. "Deep auto-encoder neural networks in reinforcement learning." In: IJCNN, 2010, pp. 1–8.

[11] L.P., K.; M.L., L.; A.W., M. "Reinforcement learning: A survey", *Journal of Artificial Intelligence Research*, vol. 4, 1996, pp. 237–285, cited By 2313.

[12] Mataric, M. J. "Reward functions for accelerated learning". In: In Proceedings of the Eleventh International Conference on Machine Learning, 1994, pp. 181–189.

[13] Mitchell, T. M. "Machine Learning". New York, NY, USA: McGraw-Hill, Inc., 1997, 1 ed..

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

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

[16] Mohri, M.; Rostamizadeh, A.; Talwalkar, A. "Foundations of Machine Learning". The MIT Press, 2012.

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

[18] Ng, A. Y.; Kim, H. J.; Jordan, M. I.; Sastry, S. "Autonomous helicopter flight via reinforcement learning." In: NIPS, Thrun, S.; Saul, L. K.; Schölkopf, B. (Editors), 2003.

[19] Ontañón, S. "The combinatorial multi-armed bandit problem and its application to real-time strategy games." In: AIIDE, Sukthankar, G.; Horswill, I. (Editors), 2013.

[20] Pan, S. J.; Yang, Q. "A survey on transfer learning", *IEEE Trans. on Knowl. and Data Eng.*, vol. 22–10, Oct 2010, pp. 1345–1359.

[21] Puterman, M. L. "Markov Decision Processes: Discrete Stochastic Dynamic Programming". New York, NY, USA: John Wiley & Sons, Inc., 1994, 1st ed..

[22] Rumelhart, D. E.; Hinton, G. E.; Williams, R. J. "Neurocomputing: Foundations of research". , Anderson, J. A.; Rosenfeld, E. (Editors), Cambridge, MA, USA: MIT Press, 1988, chap. Learning Representations by Back-propagating Errors, pp. 696–699.

[23] Rummery, G. A.; Niranjan, M. "On-line q-learning using connectionist systems", Technical Report, 1994.

[24] Russell, S. J.; Norvig, P. "Artificial Intelligence: A Modern Approach". 2003, 2 ed..

[25] Sutton, R. S.; Barto, A. G. "Introduction to Reinforcement Learning". Cambridge, MA, USA: MIT Press, 1998, 1st ed..

[26] Tesauro, G. "Practical issues in temporal difference learning". In: Machine Learning, 1992, pp. 257–277.

[27] Tesauro, G. "Temporal difference learning and td-gammon", *Commun. ACM*, vol. 38–3, Mar 1995, pp. 58–68.

[28] Tokic, M. "Adaptive ε-greedy exploration in reinforcement learning based on value differences". In: Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence, 2010, pp. 203–210.

[29] Vincent, P.; Larochelle, H.; Bengio, Y.; Manzagol, P.-A. "Extracting and composing robust features with denoising autoencoders". In: Proceedings of the 25th International Conference on Machine Learning, 2008, pp. 1096–1103.

[30] Watkins, C. J. C. H.; Dayan, P. "Technical note: q-learning", *Mach. Learn.*, vol. 8–3-4, May 1992, pp. 279–292.

[31] Zhang, C.; Lesser, V. "Coordinating multi-agent reinforcement learning with limited communication". In: Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems, 2013, pp. 1101–1108.

[32] Zhang, J.; Zong, C. "Deep neural networks in machine translation: An overview", *IEEE Intelligent Systems*, vol. 30–5, 2015, pp. 16–25.