The Knowledge Engineering Review, Vol. 33, e22, 1–21.
doi:10.1017/S0269888918000280

Q-Table compression for reinforcement learning

LEONARDO AMADO[1] and FELIPE MENEGUZZI[1]

[1]*Pontifical Catholic University of Rio Grande do Sul, Av. Ipiranga 6681, Porto Alegre, RS, 90619-900, Brazil; e-mail leonardo.amado@acad.pucrs.br, felipe.meneguzzi@pucrs.br*

Abstract

Reinforcement learning (RL) algorithms are often used to compute agents capable of acting in environments without prior knowledge of the environment dynamics. However, these algorithms struggle to converge in environments with large branching factors and their large resulting state-spaces. In this work, 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 RL fails and provide a number of possible avenues of further work on this problem.

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 (Sutton & Barto, 1998). An agent learns the optimal action for each state (a policy) as it interacts with the environment multiple times, exploring the environment and evaluating the reward of executing different actions. In order to converge to an optimal policy, an agent using traditional RL approaches must explore the entire state space, executing every possible action. Consequently, these algorithms can take a long time 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. Multiplayer games are often a challenging case of study due to the difficult of predicting and adapting to the opponent action (Wendel *et al*., 2014). Thus, learning an optimal policy by visiting all possible state-actions pairs is intractable for complex problems such as multiplayer computer games. In many multiplayer games, the number of possible states is so large that there is neither sufficient memory to store, nor sufficient time to visit all possible states (Tesauro, 1992).

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, as long as they closely represent the actual state of such environment. 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 very large state spaces is a challenging task and requires an expert in the domain to properly select the important features. There are two common 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 in an alternative, smaller, state-space 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 (Vincent *et al*., 2008) have yielded mechanisms to reduce dimensionality and find efficient encodings, using auto-encoders. In this article, 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 smaller state-space, reinforcement learning systems must try almost every pair of state-action multiple times before they have 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 by sharing the experience of multiple similar agents spread over the state-space.

Our contribution is a set of techniques to use RL in environments with multiple agents, and large state-space and branching factor. These techniques can be summarized in two different approaches: reducing state space by compressing the state representation using neural networks to decide the important features; mitigating the branching factor by learning the best action of each agent as a singular problem and sharing the learned information between agents. Recent research applies RL to simpler games that allow the input to be the raw low resolution image output from the game. However, these games still have a relatively simple state-space, which is not the case of RTS games. We thus apply RL using all available information in a complex game where traditional RL algorithms fail to converge due to the large state and action space. The usage of an auto-encoder in RL 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 (Ontañón, 2013).

This work is structured as follows. We provide the necessary background to understand this paper in Section 2. We describe the MicroRTS game and provide a brief explanation of RTS games in Section 3. We develop our approach for multi-agent RL (MARL) using a compressed state-space in Section 4. In Section 5, we explain our implementation and the experiments we use to validate and evaluate our approach. Finally, we compare our approach to related work in Section 6, and conclude the paper discussing potential future in Section 7.

2 Background

In this section, we briefly review the necessary background used to develop our RL technique. First, we provide a brief introduction to neural networks and deep auto-encoders. Then, we introduce the Q-learning algorithm and briefly discuss generalization in RL.

2.1 Neural Networks and Auto-encoders

Artificial neural networks (ANNs) are a class of machine learning models composed of multiple nodes, organized in layers, responsible for computing an output based on the activation of nodes mediated by weights in the connections between such nodes. ANNs can 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 RL 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, based on the architecture of the connections, such as deep belief networks, deep auto-encoders, convolutional neural networks and deep Boltzmann machines (Zhang & Zong, 2015). Most networks are trained using variations of the back-propagation algorithm (Rumelhart *et al.*, 1988).

A deep auto-encoder is a type of DNN capable of converting inputs to a more compact representation of itself (Vincent *et al.*, 2008). A deep auto-encoder is composed of two connected symmetrical neural networks. The first network encodes the input into an internal, latent, representation and the second decodes the internal representation back to the original input. 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

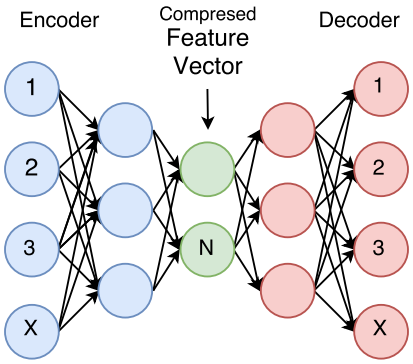

Figure 1 Deep auto-encoder architecture.

layers, *N* represents a variable number of nodes for the latent layer that the connects both networks. Blue nodes represent the nodes of the encoder network, transforming the input in the compressed representation. Green nodes represent the nodes where the input becomes its latent encoding. Red nodes represent the nodes of the decoder network, which, from the encoded representation, incrementally transform the encoded data back to the original form.

As an example, consider the encoding of an image that has 784 grayscale pixels (a 28×28 image). 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 values computed by the hidden layer in the middle of the network, i.e. the green nodes in Figure 1.

2.2 Reinforcement Learning

A 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 formally composed of actions, states and rewards (Puterman, 1994). 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. RL algorithms usually operate on MDP environments.

Q-learning is a model-free RL algorithm that does not need the transition function to learn the optimal policy (Watkins & Dayan, 1992). 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 in 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 $Q(a,s)$ represents the utility of executing action a in the state s considering the future utility that can be achieved by execution such action.

Since the agent does not know the transition function, the agent learns how to act by trying the possible results of performing an action in a certain state, thus the algorithm learns the utility of performing an

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

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:

$$U(S) = max_a Q(a, s) \tag{1}$$

Using Equation (1), we can compute the maximum utility a state can achieve without knowing the transition function. With this we can update the Q-value using the following equation:

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

Here, α is the *learning rate*, which ranges between 0 and 1. When $\alpha = 0$, nothing is learned, and when $\alpha = 1$, new learned values are fully considered. γ is the discount factor, which ranges between 0 and 1. When $\gamma = 0$, future rewards are irrelevant, and when $\gamma = 1$, future rewards are considered as being worth the same as immediate rewards. Equation (2) implements the Q-learning update as part of Algorithm 2.2, based on the specification from Watkins and Dayan (1992).

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 Tokic (2010), but in most cases it forces the agent to explore actions that were never tried in certain states, trying to execute every possible action in every state at least once. The exploration function controls how the agent behaves until the algorithm converges. Since every Q-value starts without any value, Q-learning relies only on the exploration function to start exploring the environment. We denote the exploration function in Algorithm 2.2 as function *f*.

RL can struggle to converge when dealing with huge state-spaces. Even after some training, if the basic Q-learning algorithm is executed and a new state is found, the algorithm will have to experiment all possible actions to be able to attest the utility of all possible state-actions pairs for such state. This problem comes from the fact that RL algorithms we saw so far are unable to generalize information from the domain. By generalizing, we mean estimating the utility of a state, by using information of similar states that have been previously visited. Thus, if the agent visits a new state that is very similar to a previously visited state, it has no way of estimating the utility of the new state, regardless of the number of times other similar states have been visited.

Function approximation can be used to generalize information from the state values, by using other representation methods for the Q-function, instead of the Q-table. Using only linear functions, the representation is an approximation because with only a linear function it is impossible to guarantee that the

function represents the true utility function (Boyan & Moore, 1995). In a scenario with n features, the algorithm must learn a weighted linear function of the set of features $f1,...,fn$:

$$\hat{U}_\theta(s) = \theta_1 f_1 + \theta_2 f_2 + ... + \theta_n f_n \tag{3}$$

where $\hat{U}$ is the estimated utility. A RL algorithm can determine the weights for the parameters θ, approximating $\hat{U}$ 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 RL, 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:

$$E_j(s) = (\hat{U}_\theta(s) - u_j(s))^2 / 2 \tag{4}$$

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

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

Finally, using Equation (3) 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:

$$\theta_i \leftarrow \theta_i + \alpha(R(s) + \gamma max_{a'} \hat{Q}(a', s') - \hat{Q}(a, s))f_i \tag{5}$$

With this equation, it is possible to compute a function capable of assigning weights to each feature of the state space. With such function, we can compute the value of a new state only based on it's features, without the need of experimenting the outcome of performing different actions in such state.

3 MicroRTS

We now provide a quick overview of what a RTS game is, detailing a specific instance of such type of game called MicroRTS. We then proceed to formalize the problem of playing MicroRTS as a RL problem.

3.1 RTS Games

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 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 automated players for such games is a well known challenge. Modern games, such as Starcraft, are being subject of study to create complex artificial intelligence (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 number of possible units is vast, leading to multiple ways of playing the game.

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.

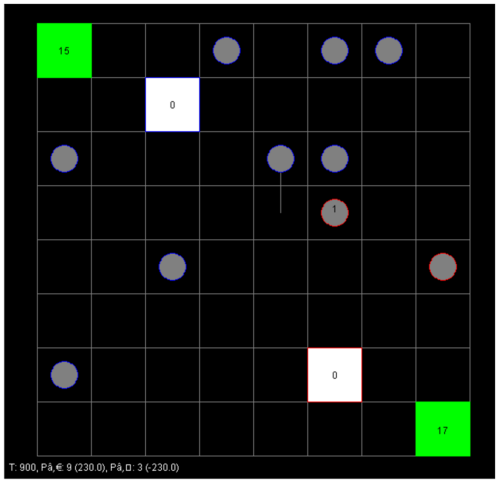

Figure 2 MicroRTS game state. RTS = real-time strategy.

3.2 MicroRTS

MicroRTS is a simple implementation of a RTS game, designed for the sole purpose of AI research. Developed by Ontañón (2013), 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. There is a frame work for implementing machine learning algorithms in Starcraft, called Torchcraft (Synnaeve *et al.*, 2016). We decided to use the MicroRTS due to the simplicity of the game.

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

1. **Worker**. This unit is represented by the gray circles, and is responsible for harvesting minerals and constructing structures. Although this unit can also fight, it does very little damage.
2. **Light**. This unit is represented by a small orange circle. Light units do little damage, but are extremely fast, having only the ability to attack.
3. **Ranged**. This unit is represented by a blue circle. Ranged units have the ability to attack from afar, having only the ability to attack.
4. **Heavy**. This unit is represented by a large yellow circle. They do high damage, but are extremely slow, also being able to sustain many attacks.

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 base is the main structure of the game and is responsible for creating workers, being represented by the white squares. This structure is also where the workers return the minerals they harvested. When the game starts, each player starts with a base and 5 resources.
- **Barracks**. The barracks are an auxiliary structure capable of producing all units except workers, represented by a gray square. The barracks can be built by the workers by using resources.
- **Minerals**. Minerals are single resource in MicroRTS, represented by a green square. Minerals can be harvested by the workers to obtain resources, they are not an exactly structure, as they have no owner or can be attacked. Minerals are finite, and each player starts near one source.

In this work, we work the totally observable version of MicroRTS, so the player can see every enemy action. With those components, MicroRTS provides a good simplification of a full-fledged RTS game.

3.3 Problem specification

The goal of a RL 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. RL algorithms, and Q-learning in particular, are well known for their ability of retrieving optimal policies in environments with no previous knowledge. However, RL 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.

RL problems can be modeled as a four-tuple $\langle S,A,T,R \rangle$ (Kaelbling *et al.*, 1996). To encode the MicroRTS scenario as a RL 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 for each state in S.

In the scenario of MicroRTS, a possible state representation 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 a numeric value to distinguish each unit. Each unit is on a position of the map, so the state representation must account every tile of the map. We represent the positions with xy_{pi}, where xy is the position in the grid, p is the player and i is the unit id. 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. Since there can be multiple bases, we track only the first base health. The state representation must account for the resources of each player, which can be done using p_{re} and e_{re}, where p_{re} is the amount of resources the player has and e_{re} is the amount of resources the enemy 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 in a state can be represented as a list, containing each possible action for unit of the player. 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 a 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 three units can be defined as $a = \left[\hat{a}_{0jp}, \hat{a}_{1jp}, \hat{a}_{2jp}\right]$.

The last component to be defined is the reward function. A reward function R maps every state s to a numeric value r that represents how desirable is to achieve such state. On the MicroRTS scenario, defining a reward for 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 current state we can determine if it was the agent actions responsible for a certain outcome, for example, killing a certain unit.

4 Approach

In this section, we discuss our approach as two distinct techniques. First, we introduce the application of Q-learning individually to each agent. Second, we explain the architecture of the auto-encoder we built and the state representation used.

4.1 Unit Q-learning

At each state, a single unit in MicroRTS has approximately five possible actions. This is manageable for relatively large state-spaces for a single agent, but it becomes a problem when learning combinations of actions for multiple agents simultaneously. During a game of MicroRTS, it is very likely for a player to control 5 units, resulting in 3135 possible actions for a single state. Computing an optimal policy is very complex for larger state-spaces with this branching factor, and the problem becomes intractable as the number of units increases. For example, when ordering 10 units simultaneously, we have 9 765 625 possible actions, providing a combinatorial explosion. To avoid dealing with such huge branching factor,

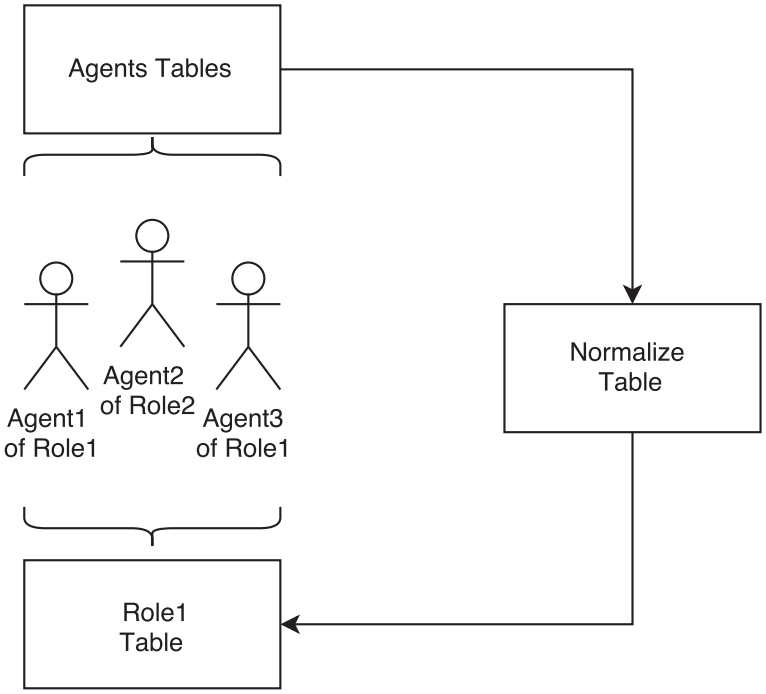

Figure 3 Unit Q-learning diagram.

we apply Q-learning to each unit individually, executing parallel Q-learning updates at 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, similar to Jaidee and Munoz-Avila (2012). This process is illustrated in Figure 3, where the units are the Agents, the Agent table is a dedicated Q-table to each unit, the normalize table is the process of normalizing all tables of each specific role, and the role table is the normalized table of such role.

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

$$Q(s,a) = \frac{\sum_{i=0}^{agents} Q_i(s,a) * frequency(Q_i(s,a))}{frequency(Q(s,a))} \tag{6}$$

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 for state s and action a. The frequency function returns the number of times a state-action pair was visited. The idea is that agents who visited a Q-value pair more often are better able to determine the value of such Q-value, so we value their experience more than agents who visited the same state-action-pair less often.

In Algorithm 2, the *mergeTables* method implements the update described by Equation (6). 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 loop assigns to each unit a Q-table based on it's role, and selects an action for such unit by using this table. The second loop merges the tables of units with the same type at the end of a simulation (at a terminal state). Finally, the algorithm returns the composed action.

4.2 State encoding

To mitigate the state space of the MicroRTS we develop an encoded representation of the game state space. To create such representation of a MicroRTS game state, we follow three steps. First, we design a binary representation for the state space we would traditionally store in the Q-table. We define this as the *raw encoding*, which is responsible for faithfully representing most aspects of a MicroRTS state. 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, applying the Q-learning algorithm in a encoded version of the state space.

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
```

Our raw encoding consists of a 156-bit binary representation. These 156 bits are based on the formal description we provided in Section 3.3 We use the 64 first bits to represent the player units position, each bit represents a position in the grid. This considers only a 8×8 scenario. Thus, if the scenario is bigger we must train another network. Using Figure 2 as example, the following bit representation would represent the blue units:

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

We use the next 64 bits to represent the enemy units, using the same concept as described before. Furthermore, we must represent the health points of the units. Since the auto-encoder has a fixed input size, we want to avoid a variable-sized matrix for all possible combinations of units health points, we decided to represent only the health points of the bases. We assign four bits to represent the health points of the player base, and four 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 five bits, where if the resources are higher than 25 in value they all use one for every bit. Five more bits must be used for the resources of the enemy player. To

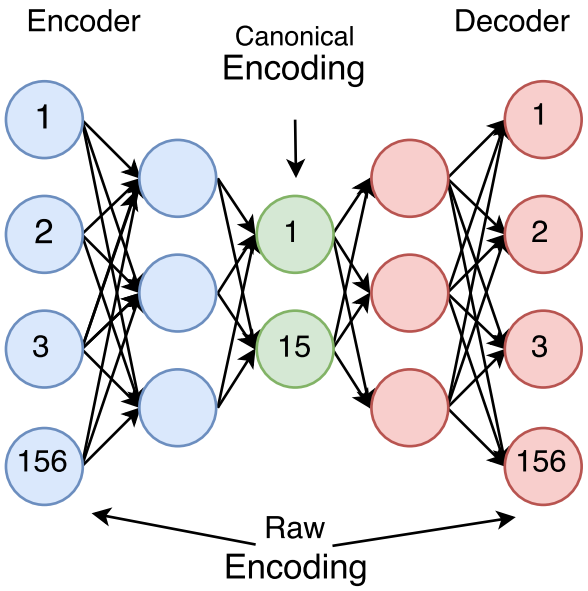

Figure 4 Auto-encoder architecture canonical encoding.

represent the unit carrying a resource, we design one bit if any unit is carrying a resource, and one 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 one bit to represent if the enemy has a barracks. The last six bits represent the action executed in that state, where the three first bits represent the type of the action, and the three other bits the direction.

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. 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 receives a binary state-representation corresponding to our raw encoding. Traditional Q-learning that relies on unfactored state representations tend to converge only when the state space stored in the Q-table is relatively small Tesauro (1995). Since we are working with binary representations, it would be ideal to compress to a number of bits that represent less than 10 000 states. The number of bits to create a representation with less than 10 000 numbers is 13, as 2^{13} results in 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. The auto-encoder has the following 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 15 100 states using the raw encoding. These states were generated from matches from two random strategies. We used random strategies matches because they are able to provide a much greater variety of states, given us more states to train the auto-encoder, providing better generalization. To ensure a binary representation in the auto-encoder, we use binary transformations after each RBM. If we used competitive strategies, we would get a much smaller number of visited states, since these strategies tend to follow one single strategy.

In Figure 4, 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 for simplicity.

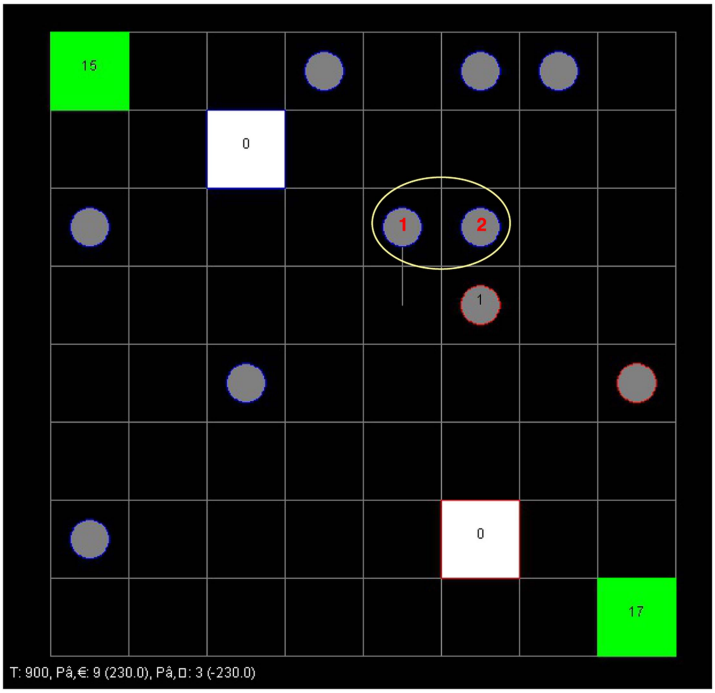

Figure 5 MicroRTS reward example. RTS = real-time strategy.

5 Implementation and experiments

In this section we detail how we implemented our approach and analyze the results of our approach against multiple approaches.

5.1 Implementation details of the approach

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. Both are rewarded for winning the game and killing an enemy unit, but the harvester also receives a positive reward for gathering a resource and for delivering to the base. Furthermore, harvesters are rewarded for constructing new structures.

To apply RL to any scenario, we must design a reward function (Mataric, 1994). 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 5, the units assigned by the red numbers 1 and 2 are attackers. 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

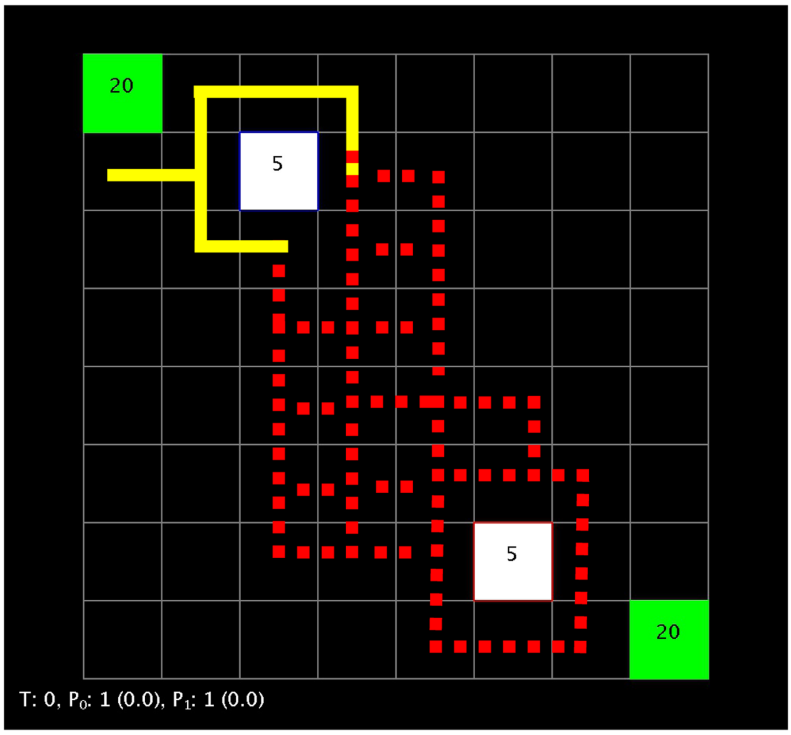

Figure 6 MicroRTS movement trace. RTS = real-time strategy.

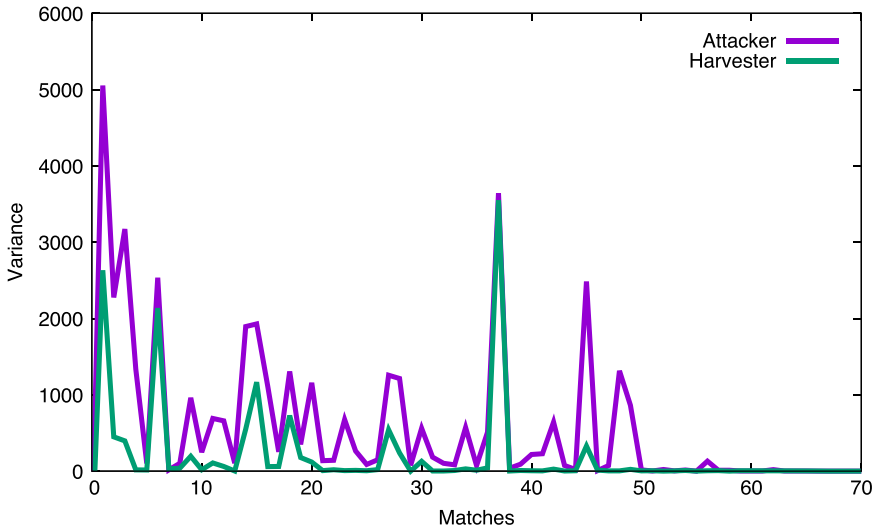

Figure 7 Convergence of attacker and harvester.

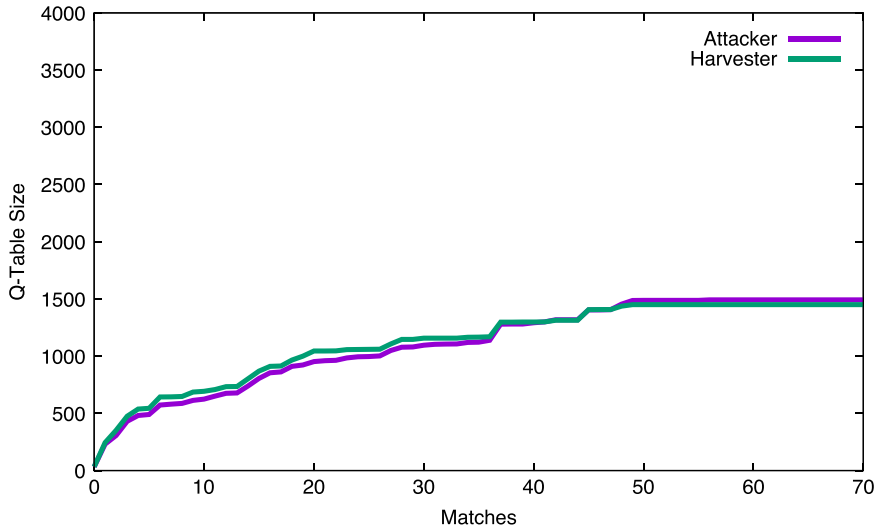

Figure 8 Q-table size of attacker and harvester.

actual state. This reward function individualizes the agent's contribution to an outcome. 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.

Using a reward function of the $R(s,a',s')$ from, we develop three distinct reward functions, one for each type of unit. One reward function for the *Base* unit, responsible for producing other units with a similar reward function for barracks. A reward function for the *harvester workers* and a reward function for *attackers*, which include workers and every other unit. Ranged units are not included in this work. The attacker is rewarded by attacking an enemy and the harvester for delivering a resource. The objective of the base is to build harvesters and attackers. The base is rewarded by producing any unit, but is also rewarded when an enemy unit is killed and when an a resource is delivered to it.

In Figure 6, 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. As we can see, the harvester unit will stay only close to the base, harvesting and delivering resources to he base. The attacker unit will consistently move towards the enemy base, circling in it, trying to find an angle to attack it.

5.2 Experiments

Using the implementation described in Section 4, 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. Figure 7 illustrates the convergence rate of both unit types in an 8 by 8 grid, the default used in our experiments. The values represent the variation of the sum of all values of the Q-table over training steps. 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. At the end, both the Q-table growth and the variation stops, converging to a policy. Figure 8 shows the Q-table size as the number of matches increases, indicating that both units have a very close Q-table size once it converges.

We test our approach against seven different approaches available in MicroRTS, namely:

- 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.

Table 1 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

AI = artificial intelligence; NaïveMCTS = Monte carlo tree search.

Table 2 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

AI = artificial intelligence; NaïveMCTS = Monte carlo tree search.

Table 3 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

AI = artificial intelligence; NaïveMCTS = Monte carlo tree search.

- 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).

Table 4 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

AI = artificial intelligence; NaïveMCTS = Monte carlo tree search.

Table 5 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

AI = artificial intelligence; NaïveMCTS = Monte carlo tree search.

Table 6 Response time for each approach.

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

AI = artificial intelligence; NaïveMCTS = Monte carlo tree search.

- 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 (Ontañón, 2013).
- NaïveMCTS (Monte carlo tree search): The approach built by Ontañón (2013). We use the default values set in MicroRTS.

We evaluate our approach under 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 similarly 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 8×8 grid, the same used in all images of MicroRTS in this work. Tables 1–4 show the results when training with only one approach. Table 1 we show the results when training with only the passive strategy. As expected, the approach performs poorly against the most complex strategies (Worker Rush and NaïveMCTS), but win consistently against the others. In Table 2, we train only against the Random approach. Since the random approach generate many different states with enemy units (especially workers) the results were better against the most competitive approaches. In Table 3, we train only against the light rush strategy. The light rush strategy builds a barrack as soon as possible, the same as Monte Carlo approach. This yields good results against the Monte Carlo approach, but performs poorly against approaches that use workers to attack. In Table 4 we train only against the worker rush approach, the most competitive of the rushes approach. The approach performs well overall, only losing to the NaïveMCTS. Finally, Table 5 shows the results of our approach when trained with each approach and played against the same approach.

Tables 1–4 indicate that our approach performs best when trained against the worker rush strategy. As we can see in Table 5, 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. 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 larger.

In Table 6, we show the response time of each strategy. As we can see, our approach was very fast and never surpassed the 1 s of response time. Most approaches compute an action nearly instantaneous, 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.

6 Related work

In this section, we discuss and compare related work. First, we discuss work done on distributed RL. Second, we introduce work done MARL. Third, we summarize transfer learning. Finally, we discuss the reduction of state-space using deep learning.

6.1 NaïveMCTS

NaïveMCTS is the algorithm develop with MicroRTS in Ontañón (2013) to play RTS games. RTS games are domains where two players can simultaneously issue actions, affecting the state. Traditional adversarial algorithms, such as *minimax*, might underestimate or overestimate the value of a state because it acts using the assumption that each player acts on different turns. However, in Ontañón (2013), 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.

6.2 Distributed RL

Q-learning and State action reward state action (SARSA) algorithms ensure that they will eventually compute an optimal policy (Watkins & Dayan, 1992). However, in some environments, this computation can take too long. Mnih *et al*. (2013) develop a new RL architecture, the deep Q network (DQN). Using a DNN 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, Nair *et al*. (2015) describe a distributed architecture that enables to scale up DQN by exploiting massive computational resources. This architecture, called *Gorila* (General RL 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.

Unlike parallel actors, the parallel learners read the stored replay memory, and update the policy, according to a given offline RL algorithm. In Nair *et al*. (2015), 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.

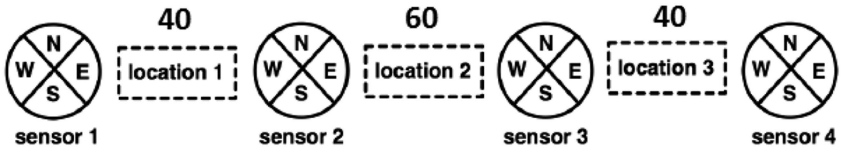

Figure 9 Tracking problem.

To prove the ability of learning faster, *Gorila* was compared to the DQN used in Mnih *et al*. (2013), 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 Mnih *et al*. (2015), Mnih further extends this work, discussing the similarities of this architecture with ability of learning of the human brain.

Regarding distributed RL, Mnih *et al*. (2015) and Nair *et al*. (2015) works share many similarities with ours, such as the usage of DNNs to encode the state space. Unlike their work, we use a pure binary representation of the internal game state as well as a reward function we designed instead of the raw game screen. 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 RL in, than emulating the difficulties of playing with the same information a human player would have.

Finally, Vinyals *et al*. (2017) develop a framework for training RL agents in the Starcraft II game. Starcraft II is a full-fledged RTS game containing multiple units, three distinct factions and complex scenarios. The early approach using RL is unable to compete with standard built-in agents of the game. However, the approach is able to learn distinct tasks for each unit, such as using harvesting resources.

6.3 Multi-agent RL

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 RL 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 RL in multi-agent systems is the need to compute a global policy for all agents (Guestrin *et al*., 2002). 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 Zhang and Lesser (2013), 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 (Kok & Vlassis, 2006). 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 9 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 9. 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 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 RL techniques, by compressing the Q-table.

6.4 Transfer learning

RL 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 (Pan & Yang, 2010): 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

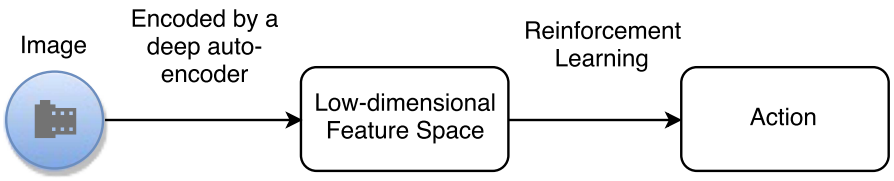

Figure 10 Image encoding framework

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/D_t or T_s/T_t. Transfer learning algorithms can be summarized by two main characteristics (Pan & Yang, 2010):

1. Which information is transferred. In the case of RL, 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 Bianchi *et al*. (2015), 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 (Hebb, 1949), 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 fundamentally different from our work, there are some similarities to our approach. Transfer learning aims to simplify computing a policy by using an already computed similar policy on a similar domain, or using existing policies to accelerate training in different tasks within the same domain. Conversely we use multiple sources of training information and merge them, accelerating the training of an agent. We could use transfer learning to learn different tasks within the domain of an RTS game, segregating similar tasks of the MicroRTS domain, much like Sharma *et al*. (2007).

6.5 Deep Learning state representation

To ensure that a RL algorithm converges to an optimal policy, the algorithm must explore the entire state space. Most RL algorithms are limited to solving tasks in which the state space has low dimensionality (Legenstein *et al*., 2010). Recent research tries to address this challenge using deep learning (Lange & Riedmiller, 2010) to find states that can be considered similar.

In Lange and Riedmiller (2010), 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 RL algorithm, that returns the best learned action for this representation. This process is shown in Figure 10.

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 Lange and Riedmiller (2010) and Mnih *et al*. (2015), we do not encode high dimensional images. We design our own binary state representation.

Finally, Barriga *et al*. (2017) use a deep convolution neural network to select the best action of an RTS game in a limited set of choices, and then utilize game tree search to improve low level tactics. The usage of CNN could benefit our work as a model to represent the state instead of our auto-encoder.

7 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 DNNs to improve the performance of RL algorithms. However, most such efforts (Mnih *et al*., 2013; Mnih *et al*., 2015), 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 RL 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 adapted for other complex scenarios where exploring the entire state is intractable. The auto-encoder can be used in any reinforcement scenario. However, the unit Q-learning would require a multi-agent scenario where there are roles defined for each agent.

For future work, we intend to improve our approach in at least two ways. First, we aim to evaluate more advanced auto-encoders and their impact in the quality of the learned policy; and second, we aim to obviate the need to model the reward function for each unity by learning it. Specifically, our auto-encoder is one of the simplest possible types available, and the quality of the compression could be improved by using a denoizing stacked auto-encoder. Furthermore, instead of using the game state as the internal representation, as it possibly provides more explicit information than a human player would have, we aim to compare the results of our current approach to one that uses graphics rendering of the state shown to the players and thus generalize the approach to any RTS game. This can be a very challenging task due to the high dimensionality of the MicroRTS game image in particular, and of commercial RTS games in general.

We currently need to design a complex reward function for each of the roles using domain knowledge. To avoid building a new reward function, we aim to use inverse RL techniques to learn the reward function from examples. Inverse RL allows an agent to learn the reward function of a domain by watching another agent perform in such domain. In the MicroRTS case, we have many computer controlled players already competitively playing the game. Our own RL 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.

Acknowledgement

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de NivelSuperior – Brasil (CAPES) – Finance Code 001.

References

Barriga, N. A., Stanescu, M. & Buro, M. 2017. Combining strategic learning with tactical search in real-time strategy games. In *AIIDE*, 9–15. AAAI Press.

Bianchi, R. A., Celiberto, L. A. Jr., Santos, P. E., Matsuura, J. P. & de Mantaras, R. L. 2015. Transferring knowledge as heuristics in reinforcement learning: a case-based approach. *Artificial Intelligence* **226**, 0 102-121.

Boyan, J. A. & Moore, A. W. 1995. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7*, Tesauro, G., Touretzky, D. S. & Leen, T. K. (eds). MIT Press, 369–376.

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

Hebb, D. O. 1949. *The Organization of Behavior: A Neuropsychological Theory*. Wiley.

Jaidee, U. & Munoz-Avila, H. 2012. Classq-l: a q-learning algorithm for adversarial real-time strategy games.

Kaelbling, L. P., Littman, M. L. & Moore, A. W. 1996. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research* **4**, 237–285.

Kok, J. R. & Vlassis, N. 2006. Collaborative multiagent reinforcement learning by payoff propagation. *Journal of Machine Learning Research* **7**, 1789–1828.

Lange, S. & Riedmiller, M. A. 2010. Deep auto-encoder neural networks in reinforcement learning. In *IJCNN*, 1–8. IEEE.

Legenstein, R., Wilbert, N. & Wiskott, L. 2010. Reinforcement Learning on Slow Features of High-Dimensional Input Streams. *PLoS Comput Biol* **6**, e1000894.

Mataric, M. J. 1994. Reward functions for accelerated learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, 181–189. Morgan Kaufmann.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

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. 2015. Human-level control through deep reinforcement learning. *Nature* **5180**, 529–533.

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. 2015. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296.

Ontañón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *AIIDE*, Sukthankar, G. & Horswill, I. (eds),. AAAI.

Pan, S. J. & Yang, Q. 2010. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering* **220**, 1345–1359.

Puterman, M. L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st edition. John Wiley & Sons, Inc.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. 1988. Neurocomputing: foundations of research. *Learning Representations by Back-propagating Errors*, 696–699. MIT Press.

Sharma, M., Holmes, M., Santamaria, J., Irani, A., Isbell, C. & Ram, A. 2007. Transfer learning in real-time strategy games using hybrid cbr/rl. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, 1041–1046. Morgan Kaufmann Publishers Inc.

R., S. & Barto, A. G. 1998. *Introduction to Reinforcement Learning*, 1st edition. MIT Press.

Synnaeve, G., Nardelli, N., Auvolat, A., Chintala, S., Lacroix, T., Lin, Z., Richoux, F. & Usunier, N. 2016. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*.

Tesauro, G. 1992. Practical issues in temporal difference learning. *Machine Learning* **8**, 257–277.

Tesauro, G. 1995. Temporal difference learning and td-gammon. *Communications of the ACM* **380**, 58–68.

Tokic, M. 2010. Adaptive e-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. Springer-Verlag.

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

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T. P., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J. & Tsing, R. 2017. Starcraft II: a new challenge for reinforcement learning. *CoRR* abs/1708.04782.

Watkins, J. C. H. & Dayan, P. 1992. Technical note: *Q*-learning. *Machine Learning* **8**, 279–292.

Wendel, V., Alef, J., Göbel, S. & Steinemtz, R. 2014. A method for simulating players in a collaborative multiplayer serious game. In *Proceedings of the 2014 ACM International Workshop on Serious Games*, SeriousGames '14, 15–20. ACM.

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

Zhang, J. & Zong, C. 2015. Deep neural networks in machine translation: an overview. *IEEE Intelligent Systems* **300**, 16–25.