Hierarchical reinforcement learning: decision-making in real-time strategy games

Bachelor's Project Thesis

Remi Niel, s2385481, r.f.niel@student.rug.nl,
Jasper Krebbers, s2585529, j.krebbers.1@student.rug.nl
Supervisor: Dr M.A. Wiering

Abstract: Real-Time Strategy (RTS) games can be abstracted to resource allocation, which is a topic applicable in many fields and industries. Reinforcement learning can be applied to many different aspects within RTS games and thus the resource allocation topic. In this thesis Q-Learning and Monte Carlo learning algorithms are tested with individual and shared application of the reward function against a random and a pre-programmed opponent. A simplified custom RTS focused on mid-level combat was developed and the reinforcement learning algorithms were combined with a multi-layer perceptron (MLP) that receives higher-order inputs to increase speed and performance. The combination of Q-Learning and individual rewards yielded the highest win-rate against both opponents. Against the random opponent it obtained a win-rate of 28.6% and a tie-rate of 61.5% for all games played resulting in a win-loss ratio of 3 to 1. Against the pre-programmed opponent it obtained a win-rate of 19.5% and a tie-rate of 75.0% for all games played resulting in a win-loss ratio of 3.5 to 1.

1 Introduction

In this thesis we will focus on the real-time strategy (RTS) genre, which as the name suggests is played in real-time where both players make moves simultaneously. Moves in RTS games can generally be seen as actions such as move to a certain position, attack a specific unit, construct this building etc. These actions can be performed by units which are semi-autonomous agents. These agents usually come in different types with their own attributes and actions they can perform. The player can control all agents that are on his/her team via mouse and keyboard. The game environment is often seen from above with an angle to show depth and teams are indicated by color, in Figure 1.1 there is a blue team attacking a red team base. The AI opponents in today's games work mostly via finite state machines (FSMs) which cannot develop new strategies and are thus predictable. A higher difficulty is usually modeled by increasing gather-, attack- and hit-point modifiers for the AI (Buro et al., 2007). The FSM behaviour is solely based on state transition tables and while dynamic scripting can optimize performance and therefore the challenge (Spronck et al., 2006), it is however still dependent on a pre-programmed rule-base. The AI will behave more dynamically, but in the end it will still be script-based.

Figure 1.1: Command & Conquer: Generals, a popular RTS game

Games are a thriving area for reinforcement learning (RL) which have a long and mutually beneficial relationship (Szita, 2012). While the RTS genre in particular seems to be a hard nut to crack for RL, there are success stories. Evolution Chamber for example uses an evolutionary algorithm to find build-orders in the game of Starcraft 2. Temporal-difference learning, Monte Carlo learning and evolutionary RL (Wiering and Van Otterlo, 2012) are among the most popular techniques within the RL approach to games (Szita, 2012). An important framework on which a lot of RL research is based is the Markov decision process (MDP), which is a sequential decision making problem for fully observed worlds where the Markov property is assumed (Markov, 1960). The Markov property assumes that future states depend only on the current state and not on the events that occurred before it. Many RL techniques use MDPs as learning problems due to their stochastic nature, but they work under the assumption that the environment is stationary which is not the case in multi-agent systems (Littman, 1994).

Hierarchical reinforcement learning (HRL) is a framework that allows RL to scale up to more complex problems (Barto and Mahadevan, 2003), which playing an RTS undoubtedly is. In RTS games the game-play consists of many different game-play components like resource gathering, unit building, scouting, planning and combat, which have to be handled in parallel in order to win (Marthi et al., 2005). Hierarchical learning even allows these components to have their own MDPs or be copies of the same component to allow for a divide and conquer strategy (van Seijen et al., 2017). However now we face the problem that the MDPs towards the top of the hierarchy consist of macro actions that take more than a single time step. HRL therefore depends on the Semi-Markov decision processes (SMDP) theory, which allows for actions that last multiple time steps (Puterman, 1994). In this thesis we will focus on the sub-process of mid-level combat strategy. Neural network implementations of low-level combat behaviour have already shown to be possible (Patel, 2009; Buro and Churchill, 2012). In previous research agents in the game of Counter Strike were given a single task and a neural network was used to optimize performance accomplishing this task.

We constructed a similar approach, but with task selection. Instead of giving the neural network a single task for which it has to optimize, our neural network will need to optimize task selection for each unit. This simulates giving an order, like defend the base or attack that unit. The implementation of the order will then be executed via an FSM. An analogy can be found in the example of Hengst's four-room task (Hengst, 2012). Our neural network will take the place of the parent-task (which rooms to move through) and low-level combat behaviour can be seen as the child-task (how to exit a room). This allows the parent-task to choose from abstract actions, which are then accomplished by the child-tasks. Abstract actions reduce the state space and the number of time steps before rewards are received, this is advantageous in RTS games due to the many options and real-time nature which requires fast decision-making (Hengst, 2012).

For learning to play RTS games we propose to use HRL with a multi-layer perceptron (MLP). The combination of RL and MLP has been already successfully applied to game-playing agents (Ghory, 2004; Bom et al., 2013). The MLP receives higher-order inputs, an approach where only a subset of (processed) inputs is used that has been successfully applied to improve speed and efficiency in the game Ms. Pac-man (Bom et al., 2013), we suspect this would also apply to RTS games. Two RL methods, Q-learning and Monte Carlo learning (Wiering and Van Otterlo, 2012), will be used to find optimal performance against a pre-programmed AI and a random AI. Since winning in an RTS game is a team effort we are curious if sharing the rewards between the whole multi-agent system is beneficial for performance compared to rewarding units on an individual basis.

A simple custom RTS was developed in such a way that every aspect can be controlled in order to reduce unwanted influences or effects. The game contains two bases, one for each team. A base spawns one of three types of units until it is destroyed, the goal of these units is to defend their own base and and destroy the enemy base. The different types of units have a rock, paper, scissor type of strengths and weaknesses. All decision-making components are handled by FSMs except for the component that assigns behaviours to units, which is the subject of our research.

Contributions There are a number of contributions to RL in this thesis. First we show the possibility of playing RTS games using higher-order inputs. Secondly we determine that the combination of RL and higher-order inputs learns to play the game surprisingly fast. Finally we show that different applications of the reward function impacts risk taking in neural networks.

This thesis will attempt to answer four research questions:
(1) Is hierarchical reinforcement learning with neural networks a viable approach to complex planning problems (SMDPs) in the form of RTS games?
(2) Which reinforcement learning approach between Q-Learning and Monte Carlo learning performs best?
(3) Is there any performance difference when using shared versus individual application of the reward function?
(4) Can RTS games be played well with the use of higher-order inputs?

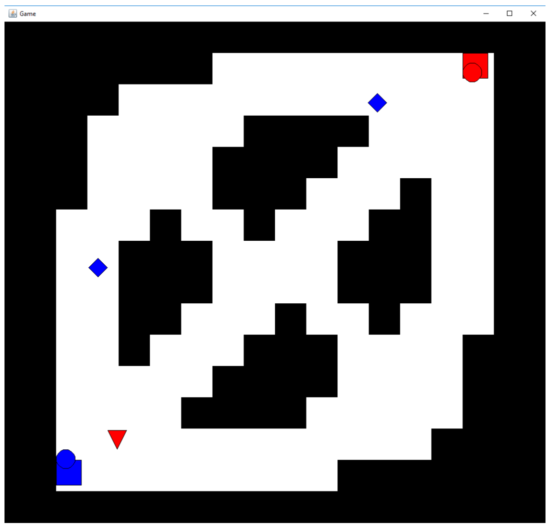

Figure 2.1: Visual representation of the custom RTS game

2 Methods

2.1 RTS Game

The game is a simple custom RTS game programmed in the object-oriented programming language Java. Since we focus on the mid-level combat behaviour a lot of RTS game-play features such as building construction and resource gathering are omitted, while other aspects are controlled by FSMs and algorithms to reduce unwanted influences and effects. An example is the A* search algorithm which is used for path finding, while unit building is done by an FSM that builds the unit that counters the most enemies for which there is not a counter already present. A visual representation of the game can be found in Figure 2.1.

The game itself consists of a $n \times n$ tiles, black tiles are walls and can't be moved through while white tiles are open space. Since units can only move in 4 directions we use the Manhattan distance to determine the distance between 2 points, although units do not steps as large as a tile, our path finding algorithms find a path from tile to tile to make them faster. When a unit is within a tile of the target it simply moves directly towards it.

The goal of the game is to destroy the opponent's base and defend your own base (indicated by large blue and red squares in Figure 2.1), which is similar to most RTS games. The game is finished when the hit-points of a base reach zero, which can be achieved by units attacking it. Depending on the unit type a base has to be attacked at least 4 times before it is destroyed. The base is also the spawning point for new units of a team, the spawning time depends on the cool-down time of the previously produced unit.

There are three different types of units: archer, cavalry and spearman. Each unit has different statistics (stats) for attack, attack cool-down, hit-point, range, speed and spawning time. Spearmen are the default units with average stats, archers have a ranged attack but move and attack speed is lowered, cavalry units are fast and have high attack power but take longer to build. All units also have a multiplier that doubles their damage against one specific type, the archer has a multiplier against the spearman, the cavalry has a multiplier against the archer and the spearman has a multiplier against the cavalry. This results in a rock, paper, scissors approach (countering mechanism) which is commonly applied in the strategy genre.

The most basic action a unit can perform is moving. Every frame it can move up, down, left, right or stand still. If it is within attacking range of an enemy building or enemy unit after moving, it will deal damage to all the enemies that are in range. The damage dealt is determined by the unit's attack power and the unit-type multiplier. When a unit is damaged its movement speed is halved for

25 frames (0.5s in real-time), which prevents units rushing the enemy base while enemy units cannot stop them in time. To make sure units do not die immediately they also have an attack cooldown after each attack, which make it so they cannot attack for a few turns after attacking.

2.2 Behaviours

Players do not directly control their units in our game, instead they give the units orders in the form of behaviours (goals). Four such behaviours are available: evasive invade, defensive invade, hunt and defend base. Units that are currently using a specific behaviour will follow rules that correspond to that behaviour to determine their moves. All behaviours make use of an A* algorithm to either find the optimal paths to other assets/locations, or to find the distance between locations. There is also an "idle" behaviour which means the unit does nothing. This behaviour is used when the unit is awaiting an order, unless a player is unable to give orders fast enough the unit should receive the next order the following frame.

2.2.1 Defend Base

When starting this behaviour the unit will select a random location within 3 tiles of its base as its guard location. This makes sure not all guards stay at the same spot. If an enemy comes close to the base (within 3 tiles) it will move towards and attack that enemy. If no enemies have come close to the base for 100 frames (2s in real time) the unit will stop this behaviour and go to the idle state awaiting a new order. The repeating part of the behaviour can be found in Algorithm 2.1.

Algorithm 2.1 Defend Base

```
if not within 3 tiles of the base then
  move back to its random guard location
else
  minDistance = 3;
  target = NULL
  for enemy in list of enemy units do
    distance=distance(base,enemy)
    if distance < minDistance then
      minDistance = distance
      target = enemy
    end if
  end for
  if target != NULL then
    Move towards and attack target
  else
    move back to its random guard location
    if frame count > 100 then
      state = "idle"
    end if
  end if
end if
```

2.2.2 Evasive Invade

The unit will take a path to the enemy base that is at most a map length longer than the shortest path. It then chooses the path with the least enemy resistance of all the possible paths. Note that while moving along the path it will also attack everything in range including the enemy base. The idea is to find a weakness in the enemy defence and to exploit it. If the unit is damaged while in this behaviour the unit returns to 'idle' until a new order is received. This is because if the unit was damaged it clearly failed to attack the enemy base while evading enemy unit. This behaviour can be seen in the pseudo-code in algorithm 2.2.

Algorithm 2.2 Evasive Invade

```
if Damaged then
  state="idle"
  return
end if
lowest resistance= $\infty$
for path in find path to enemy base do
  if resistance < lowest resistance then
    lowest resistance = path resistance
    best path = path
  end if
end for
walk best path
```

2.2.3 Defensive Invade

The unit will take a path to the enemy base that is at a the map length longer than the shortest path. It then chooses the path with the most enemy resistance of all the possible paths, to perform a counter attack on the strongest enemy front. Note that after every step the unit attempts to attack everything around it. The idea here is to either destroy invading enemy units or at least slow them down, while still putting pressure on the enemy base defences. The behaviour does not default to the "idle" state since the termination condition is either the destruction of the enemy base or the death of the unit. The pseudo-code is provided in algorithm 2.3 and is similar to the evasive invade behaviour.

Algorithm 2.3 Defensive Invade

```
highest resistance=$-\infty$
**for** path in find path to enemy base **do**
   **if** resistance $>$ highest resistance **then**
      highest resistance = path resistance
      best path = path
   **end if**
**end for**
walk best path
```

2.2.4 Hunt

The unit will move towards and attack the closest enemy asset (enemy unit or base) it can find, it will pursue the enemy asset until either it or the enemy asset is dead. If the enemy asset dies it will default back to an "idle" state, which is represented in algorithm 2.4 by the target having 0 or less health. It will be in this state until the player assigns a new order to the unit.

2.3 Reinforcement Learning

We use reinforcement learning to teach the neural network how to play our RTS. A reinforcement learning system consists of 5 parts, a model, an agent, actions, a reward function and a value function (Sutton and Barto, 1998). In our case the model is the game itself, our neural network is the agent, the policy determines how states are mapped to actions using the value function, the reward function defines rewards for specific states and finally the value function reflects the expected sum of future rewards for state-actions pairs. This value takes in account both short term rewards but also future rewards. The goal of the agent is to reach states with high values.

Algorithm 2.4 Hunt

```
**if** target=NULL **then**
   minDistance=$\infty$
   **for** enemy in list of enemy assets **do**
      **if** enemy distance $<$ distance **then**
         minDistance = enemy distance
         target = enemy
      **end if**
   **end for**
**else**
   **if** target health $> 0$ **then**
      find path to target
      walk path
   **else**
      target = NULL
      state = "idle"
   **end if**
**end if**
```

It should be noted that in reinforcement learning systems it is assumed that future rewards can be predicted using only the information in the current state: past actions / history are not needed to make decisions. This is called the Markov property (Markov, 1960).

Our reward function is fixed and based on the zero-sum principle, points are distributed according to what would be prime objectives in RTS games: killing enemy units and destroying the enemy base which results in winning the game. The rewards are received the moment a unit destroys the enemy base or kills an enemy unit. Dying or losing the game is punished, dying isn't punished harsher than the reward for killing because units are expandable given that they at least take out 1 enemy unit before dying. The reward function for our RTS game can be found in Table 2.1. If multiple rewards are given while a specific behaviour is active they are simply summed and the total reward is taken as the reward for taking the chosen behaviour.

We have two different ways of distributing the rewards, individually and shared. For individual rewards the units get only the reward they caused themselves and so the only shared reward is the "Lose" reward. It is assumed that all units are responsible for losing. With shared rewards the moment a unit achieves a rewarding event all units from the same team get the reward. The exception here is that the step-reward is still only applied once per time-step to prevent extreme time-based punishment.

Table 2.1: List of events and their corresponding rewards

Event	Reward	Description
Enemy killed	100	Unit has killed enemy unit
Died	-100	Unit died
Win	1000	Unit has destroyed the enemy base
Lose	-1000	The unit's base has been destroyed
Step	-1	Time step

We use the ϵ-greedy exploration strategy, this means that we choose the action with the highest state-action value all but ϵ of the time where $0 \leq \epsilon \leq 1$. In the cases it does not act greedily it will select a random action. We start with an ϵ of 0.2 and lower it over time to 0.02. We do this because intuitively the system knows very little in the beginning so it should explore, while over time the system should have more knowledge and therefore act more greedily.

2.4 Learning Methods

There are various learning algorithms that can be used to learn the value-function. We implemented both Q-learning and Monte Carlo methods (Sutton and Barto, 1998) and compared them to each other.

From here on a state at time t is referred to as s_t and an action at time t as a_t. The total reward received after action a_t and before s_{t+1} is noted as r_t. The time that r_t spans can be arbitrarily long.

Monte Carlo methods implement a complete policy evaluation, this means that for every state we sum the rewards from that point onward, with a discount factor for future rewards, and use the total sum of discounted rewards to update the expected reward of that state-action pair. The general Monte-Carlo learning rule is:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot (\textstyle\sum_{i=0}^{\infty}(\lambda^i \cdot r_{t+i}) - Q(s_t, a_t))$$

Where α is the learning rate and λ the discount factor. The learning rate determines how strongly the value function is altered, while the discount factor determines how strongly future rewards are weakened compared to immediate rewards.

As opposed to Monte-Carlo learning, Q-learning uses step by step evaluation. This means it uses the reward it gets after an action (can take arbitrary amount of time) and adds the current maximal expected future reward to determine how to update the action-value function. To get the expected future rewards the current value-function is used to evaluate the possible state-actions pairs. The general Q-learning rule is:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot (r_t + \lambda \cdot \max_a Q(s_{t+1}, a) - Q_t(s_t, a_t))$$

As before α is the learning rate and λ the discount factor.

The neural network itself is updated using an altered back-propagation algorithm where the target value is given by one of our learning algorithms. The back-propagation algorithm takes a target for a specific input-action pair and then updates the network such that given the same input the output is closer to the given target. When using reinforcement learning the target is given by a combination of the reward(s), discount factor and in case of Q-learning the value of the best next state-action pair. This gives us 3 different formulas to determine the target-value to train the feedforward neural network. The first function is used when this is the last behaviour of the unit, both learning methods share this formula.

$$T(s_t, a_t) = r_t$$

In other states Monte-Carlo learning uses the following formula to determine the target-value:

$$T(s_t, a_t) = \sum_{i=0}^{\infty}(\lambda^i \cdot r_{t+i})$$

While for Q-learning the following formula is used to determine the target-value:

$$T(s_t, a_t) = r_t + \lambda \cdot \underset{a}{max}\, Q(s_{t+1}, a)$$

2.5 State Representation

Our neural network does not directly perceive the game, and receives as input numeric variables that represent the state of the model. These variables have to be chosen carefully because they should contain enough information to make optimal decisions. Including more information generally needs

Table 2.2: Inputs used to represent a state

Unit specific inputs
Amount of hit-points left
Boolean (0 or 1) value "is spearman"
Boolean (0 or 1) value "is archer"
Boolean (0 or 1) value "is cavelry"
Minimal travel distance to enemy base
Minimal travel distance to own base
Resistance around the unit
Game specific inputs
Amount of defenders
Amount of attackers
Amount of hunters
Amount of enemy spearmen
Amount of enemy archers
Amount of enemy cavelry
Minimal travel distance between base and enemy assets

a larger network to make use of the information, which makes the method slower and takes longer to train. In our case the network gets 14 inputs, see Table 2.2. Half of the inputs are about the unit for which the behaviour has to be decided while the other half contains information about the current state of the game.

Unit specific inputs contain first of all basic information: the amount of hit-points the unit has left and which type it is (in the form of 3 boolean values). It also contains 2 inputs which give distance values namely the minimal travel distances to the enemy base and its own base. The final unit specific information contained in the inputs is the 'enemy resistance' around the unit, this counts all enemy units in a 5×5 square around the unit where the unit type it is strong against is counted as a half unit. Then the amount of friendly units in the same square is subtracted from this number. The result gives an indication how dangerous the current location is for the unit.

Game-wide inputs provide information about the owner of the units: the amount of defenders, attackers and hunters the owner already has. Note that for attackers the aggressive and evading invaders are summed. This information could be used to prevent creating too many defenders. The inputs about the enemy contain the composition of the enemy army, so the amount of archers, cavalry and spearmen. This could be used to prevent for example hunting behaviour if the enemy has a lot of archers while the unit in question is a spearman. Given that a spearman cannot perform ranged attack and is not very fast it would be taken out before achieving anything. The last input gives the distance between the owner's base and the enemy unit closest to it.

3 Experiments and Results

3.1 Testing Setup

To test our methods all configurations (shared vs individual rewards and Q-learning vs Monte Carlo methods) have been tested against two preprogrammed opponents, a random AI which simply chooses a random behaviour whenever it needs to make a decision and a classic AI which we programmed ourselves to follow a set of rules we thought to be logical. We also tested these two against each other and found that they never tied (all games ended within 4500 frames) and the classic AI wins about 46.5% of the games. Making a deterministic AI that plays well against the random AI as well as other opponents is quite difficult since the random AI is hard to predict, and countering the random AI specifically could result in the AI equivalent of over fitting where it wins from the random AI but loses from other opponents. Besides the predictability issue the limited amount of options in commands makes it hard to issue the right order for every specific situation.

Each configuration has been ran for 100 trials where each trial is 26 epochs (games) long. Each initial neural network was stored on disk and after each game the network was again stored on disk. All of those stored networks were then tested against the same AI it was trained against for 40 games, during these 40 games training and exploration is disabled to determine the network's performance. We then stored the win, lose and tie percentages. A tie is a game that is not finished after 4500 frames (90 seconds real-time).

The networks consist of 14 inputs and 4 outputs. After several parameter-sweeps of all configurations we found that optimal performance was achieved with the following parameter settings. We used 2 hidden layers with layer sizes of 100 and 50 and a learning-rate which started at 0.005 and that is multiplied with 0.7 after each game degrading to a minimum of 10^{-6}. The exploration rate also degrades from a start exploration rate of 20% to a final exploration rate of 2%. The discount factor is 0.9. Since most units have a relatively small amount of behaviours before dying we discount future rewards relatively harshly. Finally we added momentum to the neural network, this means that the previous change of the network is used to adjust how the network should change now. In our case 40% of the previous change is added to the current change of a weight in the network. This reduces the fluctuations in weight changes when around an optimal value and it also speeds up training the network if weights have to be updated in the same direction repeatedly.

3.2 Results

The results that were gathered are plotted in Figures 3.1 - 3.4. Figure 3.1 and Figure 3.2 contain the mean ratio between wins and losses after X amount of epochs (games) for different combinations of learning algorithms and reward applications. Figure 3.1 shows the ratios of every configuration playing against the random AI, while Figure 3.2 shows the ratios for every configuration playing against the classic (pre-programmed) AI. The win-loss ratio shows how well the neural network performs in comparison to the opponent, a value of 1 represents equal performance. A value higher than 1 such as the Q-learning individual rewards result in Figure 3.1 represents better performance than the opponent, while a value lower than 1 represents worse performance than the opponent.

The results shown in Figure 3.3 and Figure 3.4 contain the weighted sum of the mean win- and tie-rates for all different combinations of learning algorithms and reward applications. The win-rate has a weight of 1, loss-rate a weight of 0 and the tie-rate has a weight of 0.5. The lines indicate the mean weighted sum while the gray area indicates the standard deviation.

In all figures it can be observed that all lines increase over time, this indicates that all configurations at least improve their performance during training. In the figures you can clearly see that the combination of Q-learning with individual rewards outperforms all other configurations significantly. After training it achieves a final win:loss ratio which is approximately 7:2 against the classic AI and 3:1 against the random AI, roughly 3 times higher than the second best configuration. The weighted sum of its win- and tie-rates are also significantly higher than all other combinations. It is noticeable that Q-learning outperforms Monte-Carlo learning in both performance measures given that the other factors are equal and individual rewards outperforms shared rewards given the other factors are equal.

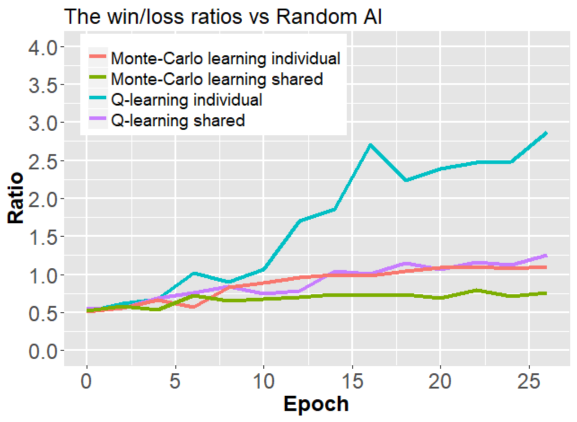

Figure 3.1: Graph that shows the ratio between wins and losses for all configurations against the random AI

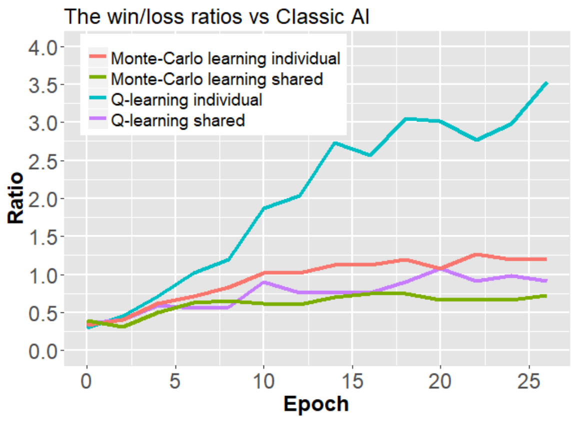

Figure 3.2: Graph that shows the ratio between wins and losses for all configurations against the classic AI

All the results measured show considerable tie-rates, against the classic AI the tie-rates are mostly in the region of 65-75% and against the random AI they are mostly between 45-65% as shown in Table 3.1. The exceptions for both opponents are

Table 3.1: Mean performance after 26 epochs (games)

Opponent	Method	Reward Application	Win-rate	Tie-rate	Loss-rate	Win:loss
Classic	Q-learning	Individual	19.5%	75.0%	5.5%	7:2
Classic	Q-learning	Shared	13.2%	72.3%	14.4%	9:10
Classic	Monte-Carlo methods	Individual	18.0%	67.0%	15.0%	6:5
Classic	Monte-Carlo methods	Shared	21.9%	48.0%	30.0%	7:10
Random	Q-learning	Individual	28.6%	61.5%	10.0%	3:1
Random	Q-learning	Shared	23.5%	57.8%	18.8%	5:4
Random	Monte-Carlo methods	Individual	28.8%	45.1%	26.2%	11:10
Random	Monte-Carlo methods	Shared	32.0%	26.2%	41.9%	3:4

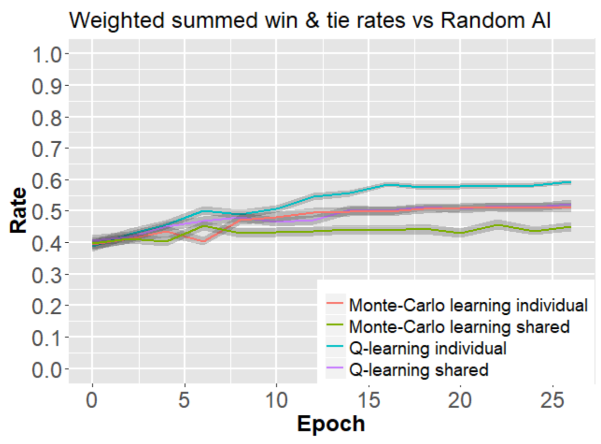

Figure 3.3: Graph that shows the summed ratios of wins and ties for all configurations against the random AI

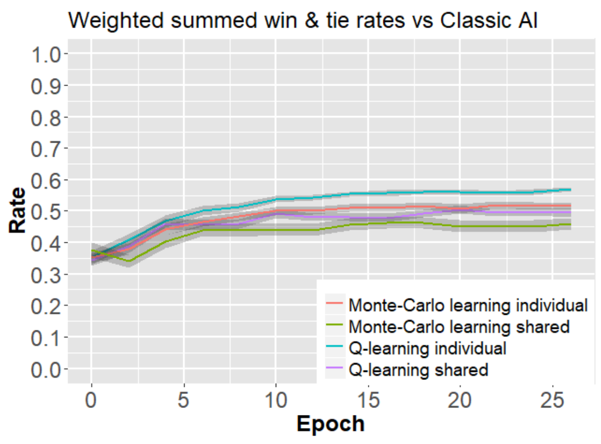

Figure 3.4: Graph that shows the summed ratios of wins and ties for all configurations against the classic AI

the results of Monte Carlo methods using a shared reward function, there the tie-rate converges to 50% against the classic AI while it converges to 25% against the Random AI. This might seem favourable, but the decrease of the tie-rate has an almost one to one inverse relation with the loss-rate and is thus an overall worse result.

3.3 Discussion

The results show that individual rewards out perform shared rewards, we suspect the reason for this is that shared rewards incentivizes taking less risks. The unit that takes the risk to hunt enemy units or attack the enemy base gets the same reward as a unit that defends the base while it is not under attack. So the system then learns that defending the base from nothing in this case was as good as attacking the enemy. While in the case of individual rewards only using the behaviour at the right time is rewarded, such as killing an enemy while defending or destroying the base while attacking.

We also encountered relatively high tie-rates, there are several possible causes for this relatively high tie-rate in the measured results. A round is deemed a tie when a time-limit of 90 seconds is reached, this feature is implemented to reduce stagnating behaviour and allow for faster data collection. Increasing the time limit should lower the amount of ties. The game's simple nature also represents the so called "early-game" in most RTS games, this stage does not usually yield a winner unless major mistakes are made or huge risks are taken.

4 Conclusions

With the obtained results there is clear data which shows that Q-learning performs better than Monte Carlo methods in our RTS game, thereby answering our second research question. The results also show that individual rewards perform better than shared rewards, this answers our third research question. The performance of Q-learning with individual rewards as shown by the win-loss ratio of 3.5:1 against the pre-programmed AI and 3:1 against the random AI supports our theory that hierarchical reinforcement learning with higher-order inputs is viable approach for learning to optimize behaviour selection in our RTS game, thereby answering both our first and fourth research question.

4.1 Future Work

Even though our assumption that it is possible to play RTS games with higher-order inputs is confirmed, the performance difference in comparison with using much more input information is not known. This might change performance significantly, but it most likely will increase computation time significantly as well. Without research this will be impossible to tell with any certainty however.

There are several facets which deserve a closer look following our experiences during this research. Especially the use of different modules and the multiplicative effects they can have. The unit builder is a great example, we found that having a more intelligent unit builder significantly improves performance. Even though the unit builder in this case was handled by an FSM it implies that adding a module to the AI that can learn which units to build would likely increase the performance as a whole significantly. The result of a neural network using a smart unit builder (FSM) against our classic AI with a random unit builder can be observed in Figure 4.1. One can clearly see that the win-rate approaches 90%. We suspect that the combination of the hunt behaviour with the right units has a significantly positive impact on performance.

More performance gain could possibly be obtained by training (more) behaviours in a similar fashion to Patel's bot training (Patel, 2009). If this is done in a bottom-up style where low-level behaviours are trained first then higher level behaviours have access to optimized sub-behaviours.

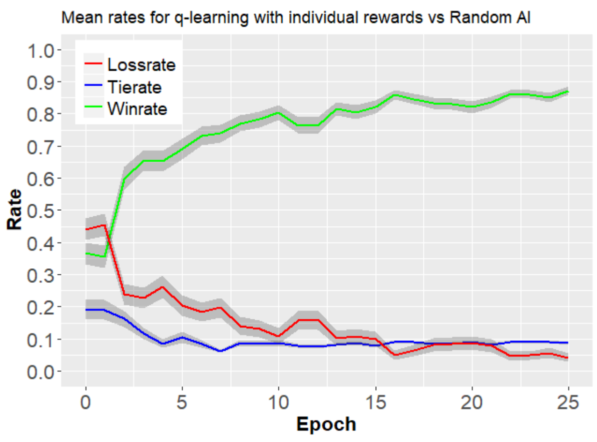

Figure 4.1: Graph of the performance for Q-learning with individual rewards where it has an improved unit building algorithm compared to the opponent's random choice

This will result in optimized behaviour in multiple levels of the hierarchical reinforcement learning structure. Other factors that may improve performance are learning techniques and structures such as HAMQ, MAXQ and HEXQ (Wiering and Van Otterlo, 2012) that promote integrated learning through multiple levels of the hierarchical system (Hengst, 2012). Especially since the network is trained for units individually, these learning techniques in combination with unit interaction options might be able to allow for squad formations and more coordinated behaviour between individual units on the same squad.

To increase the external validity of research like this, we strongly recommend the application of the tested techniques on different (or fully developed) games. This will most likely require the SMDPs to be extended to a new form op MDP similar to partially observable MDPs, since most RTS games use mechanics like the "fog of war" which hide parts of the map and this makes the environment only partially observable for each player.

References

A.G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13:341–379, 2003.

Luuk Bom, Ruud Henken, and Marco Wiering. Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs. In *Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2013 IEEE Symposium on*, pages 156–163, 2013.

Michael Buro and David Churchill. Real-time strategy game competitions. *AI Magazine*, 33(3):106, 2012.

Michael Buro, Marc Lanctot, and Sterling Orsten. The second annual real-time strategy game AI competition. *Proceedings of gameon NA*, 2007.

I. Ghory. Reinforcment learning in board games. *Department of Computer Science, University of Bristol, Tech. Rep*, 2004.

Bernhard Hengst. Hierarchical approaches. In *Reinforcement learning State-of-the-Art*, pages 293–323. Springer, 2012.

Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the eleventh international conference on machine learning*, volume 157, pages 157–163, 1994.

Andrei Andreyevich Markov. The theory of algorithms. *Am. Math. Soc. Transl.*, 15:1–14, 1960.

Bhaskara Marthi, Stuart J Russell, David Latham, and Carlos Guestrin. Concurrent hierarchical reinforcement learning. In *IJCAI*, pages 779–785, 2005.

P. Patel. Improving computer game bots behavior using Q-learning. Master's thesis, Southern Illinois University Carbondale, San Diego, 2009.

Martin L. Puterman. Markov decision processes. 1994. *John Wiley & Sons, New Jersey*, 1994.

Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. Adaptive game AI with dynamic scripting. *Machine Learning*, 63 (3):217–248, 2006.

Richard S Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

István Szita. Reinforcement learning in games. In *Reinforcement Learning State-of-the-Art*, pages 539–577. Springer, 2012.

Harm van Seijen, Mehdi Fatemi, Joshua Romoff, Romain Laroche, Tavian Barnes, and Jeffrey Tsang. Hybrid reward architecture for reinforcement learning. *arXiv preprint arXiv:1706.04208*, 2017.

Marco Wiering and Martijn Van Otterlo. *Reinforcement Learning State-of-the-Art*, volume 12. Springer, 2012.

A Appendix

A.1 Division of work

A.1.1 The game

Remi:

- Interface
- Game controller
- Graphical representation
- Neural network & learning methods
- Pathfinding & movement

Jasper:

- Unit statistics
- Attacking & collision
- Extension of reward function for shared rewards
- Classic AI opponent

Shared:

- Game design

A.1.2 Report

Remi:

- Results collection
- Results plotting
- Writing sections 3.2-3.5
- Reward table

Jasper:

- Writing sections 1, 2, 3.1, 5
- Providing pseudo-code in section 3.2

Shared:

- Writing section 4
- Results table

Parts that were written by one person were corrected and partly rewritten by the other.