university of groningen / faculty of science and engineering

Forest Fire Control with Connectionist Reinforcement Learning

Bachelor's Thesis

Travis Hammond, dashdeckers@gmail.com
Dirk Jelle Schaap, d.j.schaap@student.rug.nl
Supervised by dr. Marco A. Wiering, m.a.wiering@rug.nl

Abstract: With global temperatures on the rise, forest fires are becoming more frequent and forest fires, in turn, contribute to global warming by releasing large amounts of CO_2 into the atmosphere and eliminating the trees that would be able to process the CO_2. Finding a way to effectively control and contain these fires is therefore becoming more and more of a priority. In this paper, we propose a connectionist reinforcement learning system that can learn to contain the spread of a simulated fire. It can do this by cutting fire lines around the fire, removing the fuel needed for it to spread further. We show the performance of a connectionist Q-Learning algorithm with a target network and experience replay, and compare it to three other algorithms. One that uses the on-policy algorithm SARSA, one with the addition of a dueling network architecture and one with both modifications. To accelerate learning, we use a state representation in which the system receives as input three versions of the full map size each showing only a single feature such as agent location and fire locations. We also provide the algorithm with different amounts of demonstration data. The results show the ability of each proposed system to successfully contain the fire within a reasonable number of training episodes. Both modifications have their advantages and disadvantages with regard to reliance on demonstration data and learning stability. Dueling SARSA, combining both modifications, shows the best performance.

1 Introduction

1.1 Forest Fires

The ever increasing temperature around the globe due to global warming brings many consequences. One of which is the increased risk of forest fires. Warmer climates are plagued by forest fires not only more frequent, but also more intense. In most cases, beginning wildfires are extinguished before they get out of hand. Sadly, some wildfires escalate into nearly uncontrollable infernos.

Fighting these forest fires is a challenging task. To extinguish a fire one or more of the three required elements has to be eliminated: fuel, heat or oxygen. The ordinary tactic is to remove the heat and oxygen by spraying water or foam from hoses, but large forest fires require more effort to be contained. Possible options include dropping water bombs via aircraft, burning down specific areas in a controlled fashion, or using a bulldozer to cut fire lines. The use of these techniques need to be carefully planned by the fire-fighters when constructing a plan. To create the perfect plan is a near impossible job and it is not uncommon for plans to fail and cause the loss of more forest.

Not only can forest fires result in the tragic loss of lives and houses, the ecological effect has to be taken into account as well. Trees and plants are a key factor in the carbon cycle (Kasischke, Christensen Jr, and Stocks, 1995). Using photosynthesis massive amounts of CO_2 are filtered from the atmosphere and stored. When fires destroy large forests, all this stored CO_2

is released back into the atmosphere, which is inconsistent with the carbon cycle. Since this CO_2 is considered a greenhouse gas (Houghton, Jenkins, and Ephraums, 1991) which boosts the already increasing global warming, this will increases the likelihood and risk of forest fires. The just described relationship has the potential to result in a dangerous cycle with grave consequences for the ecosystem as well as for the habitability of the planet for humans.

In light of the seriousness of the problem there is still not much research being done in the field of artificial intelligence to optimize the coordination of fire fighting efforts. Previous research mostly focussed on the detection and prediction of forest fires, but exceptions include investigations into how to construct a simulator for forest fires and how reinforcement learning algorithms could optimise policies by interacting with such a simulation (Wiering and Doringo, 1998), research exploring how enforced sub-populations (ESP) could be used to evolve neural network controllers capable of solving the forest fire problem (Wiering, Mignogna, and Maassen, 2005), and a model of multi-agent coordination in fire fighting scenarios (Moura and Oliveira, 2007).

In this paper we explore how connectionist reinforcement learning (RL) can be used to allow an agent to learn how to contain forest fires in a simulated environment by using a bulldozer to cut fire lines. We make use of existing algorithms: Q-Learning (Watkins, 1989), SARSA (Rummery and Niranjan, 1994) and Dueling Q-Networks (Wang, de Freitas, and Lanctot, 2015). We show that using a simple baseline algorithm to generate demonstration data to be used in experience replay can greatly increase the algorithm's performance. We show that these algorithms are able to complete this task successfully in small simulations. We also introduce a new RL algorithm, Dueling SARSA, which combines the latter two and outperforms all, especially in simulations of a larger size where others fail.

Our research question is: Which connectionist reinforcement learning algorithm, Q-learning, SARSA, Dueling Q-learning or Dueling SARSA, performs best for containing the spread of simulated forest fires by cutting fire lines?

1.2 Reinforcement Learning

Reinforcement learning (Sutton and Barto, 2018) is a machine learning paradigm typically consisting of two elements. The first is the agent, which represents the reinforcement learning algorithm, and the second is the environment, which represents what the algorithm is trying to solve. This is typically a game or in this case, a simulated forest fire that should be contained.

At each discrete time step $t \in \{1, 2, 3..., T\}$, the environment provides the agent with an observation $s_t \in \mathcal{S}$. Then, the agent interacts with the environment by choosing an action a_t from a limited set of possible actions $\mathcal{A} = \{1, ..., K\}$, and observes the result of that action in state s_{t+1} and the obtained reward r_t. This interaction can be modelled by a Markov Decision Process, or MDP, as long as the Markov property holds: The probability of state s_{t+1} only relies on the previous state s_t and the performed action a_t. This property indeed holds, as the simulation only requires the current state and agent action to produce the next state.

The goal of the agent is to select actions in a way that maximizes the cumulative future reward from the current time step t, which is defined as:

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}, \tag{1.1}$$

where T is the time-step at which the game terminates and $\gamma \in [0, 1]$ is a discount factor that determines the trade-off between the importance of immediate and delayed rewards.

A policy π is a mapping of states to actions (or distribution over actions). To determine the optimal policy π^*, that leads to the highest reward as defined in Equation (1.1), we define the optimal action-value function (also known as Q^*) to be:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t | s_t = s, a_t = a, \pi] \tag{1.2}$$

We can compute this Q-function using dynamic programming methods through iterative

updates to the Bellman equation:

$$Q_{i+1}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_i(s',a')], \tag{1.3}$$

where $P(s'|s,a)$ is the probability of observing state s' after executing action a in state s, and $R(s,a,s')$ is the reward obtained after executing action a in state s and ending up in state s'. Such a value iteration algorithm will eventually converge to the optimal Q-function Q^* as $i \to \infty$. From this function, the optimal policy can easily be derived by simple taking the highest-valued action in each state. In practice, the transition function is not known and there can be a huge number of states, and therefore dynamic programming cannot be used. In this case, connectionist reinforcement learning can be used. In connectionist reinforcement learning, it is common to approximate this function using a neural network:

$$Q(s,a;\theta) \approx Q(s,a), \tag{1.4}$$

where θ are the parameters, or weights, of the Q-Network.

2 Methods

2.1 Environment

The forest fire is simulated using a grid of cells, each of which has a number of attributes. The main ones are heat potential, temperature, ignition threshold and fuel. Heat potential is the amount of heat, once ignited, the cell radiates to its neighbours to increase their temperature. As soon as the temperature of a cell reaches the ignition threshold, the cell ignites and burns. A burning cell consumes one fuel per iteration, and stops burning and radiating heat when the fuel is empty, becoming a dead cell. The heat from a burning cell reaches the cells directly north, south, east and west of it. If that cell is flammable its temperature is increased by the heat potential of the burning cell, otherwise nothing happens.

For a visual representation please see Figure 2.1. The shape of the grid is always square and has a size of either 10-by-10 or 14-by-14 cells.

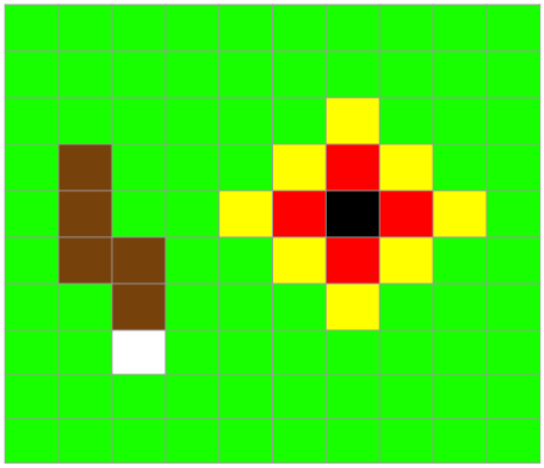

Figure 2.1: A visual representation of the environment. The agent is shown in white, leaving behind a trail of inflammable dug cells (shown in brown). The trees (green) can ignite to become burning cells (red), which heat up the neighbouring cells (yellow). When a burning cell runs out of fuel it dies (black)

The green cells represent trees that can be ignited. The agent (or bulldozer) is represented by the white tile. Wherever it moves it destroys the trees and an empty, inflammable (brown) cell is formed. A line of these dug cells forms a fire line over which the fire cannot spread. Dead cells are represented in black. The agent has to move each time step, it is not allowed to idle on the same cell, and it can only die if it moves into an actively burning (red) cell. The only possible actions for the agent to take are the 4 valid movements (north, south, east, west), and the agent is always digging as it moves. The environment reaches a terminal state and restarts if the agent dies, or if there are no more burning cells.

2.1.1 The Reward Function

Any reinforcement learning algorithm will rely on the two things that constitute the input to the agent. The quality of the state representation, or in other words how much of and how well the agent can perceive the environment, and the quality of the reward function,

which defines how well the agent's notion of success corresponds with the desired behaviour. Because of this, the performance of the agent may depend heavily on the design of the reward function.

The reward function also determines the speed at which the agent will be able to learn the optimal policy. To take the gradient descent analogy of a problem landscape, if the reward function produces a smooth gradient to the optimal solution, the agent will be able to find a path to that solution more easily than if the reward resembles a flat landscape with sparse spikes in which the value jumps from almost always 0 or negative to a positive reward (Sutton and Barto, 2018). In other words, the agent should be provided gradual feedback instead of sparse and delayed rewards in order to facilitate fast and efficient learning.

We defined the reward function as

$$R_t = \left\{ \begin{array}{ll} 1000, & \text{Fire is contained} \\ 1000 * (p), & \text{Fire burns out} \\ -1000, & \text{Agent dies} \\ -1, & \text{Otherwise} \end{array} \right\}, \tag{2.1}$$

where p is the percent of the map undamaged by either fire or digging. Note that this reward function does not produce a smooth gradient, and the performance of the agent will likely suffer as a result.

Containment of the fire is defined as

$$\sum_{f \in \mathcal{F}} \sum_{b \in \mathcal{B}} astar(f, b) = 0, \tag{2.2}$$

where f is a burning cell from the set of currently burning cells $\mathcal{F}$ and b is a cell on the border of the map from the fixed set of border cells $\mathcal{B}$. The function $astar$ is defined as

$$astar(f, b) = \left\{ \begin{array}{ll} 1, & \text{if A* path exists} \\ 0, & \text{Otherwise} \end{array} \right\}, \tag{2.3}$$

where a path is a sequence of directly connecting cells starting with cell f and ending with cell b, determined using the A* path-finding algorithm and not allowing diagonal steps. The intuition is that if there exists a path between any burning cell and any cell on the border of the map, then there exists a way for the fire to spread beyond control and so it is not contained.

2.1.2 The State Representation

The state of the environment, as it is visible to the agent, consists of 3 matrices of size N^2 with a boolean domain resulting in, after flattening, an array of $3 * N^2$ boolean inputs where N is the size of the square map. The environment is thus represented three times: One layer contains only the agent position, translating to a matrix of zeros except for a single one representing the position of the agent. The second layer consists of the positions of the fire. Cells that are on fire are represented by a 1, the rest are set to 0. The third layer represents the fire lines cut by the agent in a similar boolean fashion, resulting in a total of 300 inputs to the agent when $N = 10$. This is shown graphically in Figure 2.2.

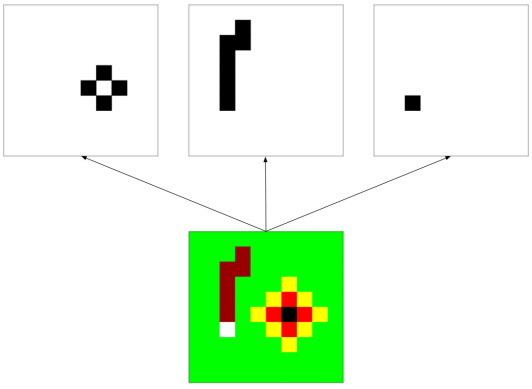

Figure 2.2: An example of the state representation. Each layer shows an important aspect of the world: The location of the fire, inflammable cells and the agent itself.

This state representation can speed up the learning process as well as increase performance (Knegt, Drugan, and Wiering, 2018). Indeed it had a noticeable effect on the performance and learning speed of our implementation compared to a single matrix representing the grayscaled map as input, likely because the agent can more easily differentiate between important attributes due to the boolean domain, and

because only the relevant information is presented. Furthermore, the agent can now see whether the cell it is occupying is already dug or not.

2.2 RL Algorithm

2.2.1 Known Problems with Connectionist RL

The combination of function approximation (the neural network), bootstrapping (TD methods that update Q-values using estimated return values) and off-policy training (the Q-Learning algorithm) make up the deadly triad (Sutton and Barto, 2018), which is known to cause instabilities and even divergence in the learning process. This is at least partly due to correlations in the sequence of consecutive observations, the fact that even small updates to Q can change the policy and therefore the data distribution, and the correlations between the Q-values $Q(s_t, a_t; \theta_t)$ and the target values $Y_t = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t)$ (Mnih, Kavukcuoglu, Silver, Rusu, Veness, G. Bellemare, Graves, Riedmiller, K. Fidjeland, Ostrovski, Petersen, Beattie, Sadik, Antonoglou, King, Kumaran, Wierstra, Legg, and Hassabis, 2015).

As will be clear by the end of this section, all three of these elements of the deadly triad are present in some of the algorithms. There are, however, two well known strategies to reduce this effect and stabilise learning: Using experience replay and a target network.

2.2.2 Network Architecture

First we define the neural network architecture* that will approximate Q, or the action-value function. We use only a shallow network with one hidden layer of 50 units using a sigmoidal activation function, and a simple output layer with a linear activation. The network takes as an input a state s, consisting of an array of size $3 * N^2$, and outputs an array of size 4 corresponding to the Q-values of taking any of the 4 possible actions a in that state. We used the Adam optimizer to train this network.

*The source code is available at https://github.com/dashdeckers/Wildfire-Control-Python and can be accessed and used for non-commercial uses only.

2.2.3 Experience Replay

Figure 2.3: Schematic structure of the experience replay process

Instead of training the network on incoming experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time step t directly, the experiences can be stored in a memory buffer $B_t = \{e_1, ...e_t\}$ and the network can be trained on a mini-batch of memories randomly sampled from this buffer $(s, a, r, s') \sim U(B)$ (see Figure 2.3). The buffer B allows for M experiences to be stored until it is full, at which point the oldest memories will be discarded to make room for new ones.

This helps because neural networks assume that each training sample is identically and independently distributed from the population, which is in this case the set of Q-values that the network is set up to approximate, and training directly on incoming experiences violates this assumption in two regards.

For one, consecutive incoming experiences are obviously highly correlated because the environment does not radically change after any action (unless it is a terminal state). This means that any experience differs from the previous experience only to the extent to which the environment can change in one update step from a single action from the agent.

Secondly, the distribution of incoming experiences is also dependent on the agent's current policy. If the current policy determines that the agent should head east, then the next experiences recorded by the agent will involve the agent headed east. Apart from the obvious correlation, this can also lead to feedback loops and the network parameters could get stuck in local optima or even diverge (Tsitsiklis and Van Roy, 1997; Mnih et al., 2015).

In addition to stabilising the learning process, this approach also increases sample efficiency by allowing memories to be re-used multiple times until they are discarded when they reach the end of the queue. We can also pre-initialize this buffer with memories before learning to provide the agent with demonstration data, as will be explained in Section 2.2.8.

2.2.4 Target Network

Another source of instability arises when we use the predictions of the network to generate the target values which directly update the weights of that same network, as in the standard Q-learning update

$$\theta_{t+1} = \theta_t + \alpha(Y_t - Q(s_t, a_t; \theta_t))\nabla_{\theta_t} Q(s_t, a_t; \theta_t), \tag{2.4}$$

where α is the learning rate and the target Y_t is defined as

$$Y_t \equiv r_t + \gamma \max_a Q(s_{t+1}, a; \theta_t). \tag{2.5}$$

The update rule in Equation (2.4) resembles stochastic gradient descent, updating the value $Q(s_t, a_t; \theta_t)$ towards the target value Y_t. Using the same network can lead to unwanted feedback loops. A delay to the loop can be added to reduce these effects by using a periodically updated, frozen copy of the network with weights θ^- to generate the target values instead (as shown schematically in Figure 2.4)

$$Y_t^{Target} \equiv r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t^-), \tag{2.6}$$

where every C time steps, the weights of the network are copied into the target network $\theta_t^- = \theta_t$. This modification was also used by Mnih et al. (2015) alongside experience replay to make the algorithm more stable and dramatically increase its performance.

2.2.5 Q-Learning versus SARSA

We implemented both Q-Learning (Watkins, 1989) and SARSA (Rummery and Niranjan, 1994) to investigate the difference in performance when using an off-policy versus on-policy algorithm.

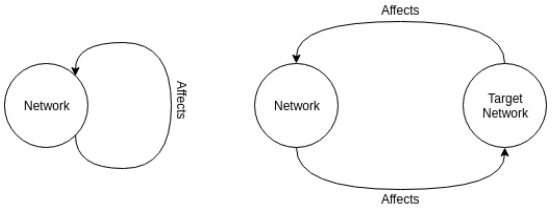

Figure 2.4: Using a target network to add a delay to the feedback loop. The main network affects (copies itself into) the target network only every C iterations.

There are two commonly used policies, the greedy and the ϵ-greedy policy, and both depend on Q. The greedy policy simply always chooses the best action to take in the current state at any given time step: $\text{argmax}_a\, Q(s_t, a)$.

The ϵ-greedy policy is similar, but it regulates the exploration/exploitation trade-off with the parameter $\epsilon \in [0, 1]$. With a probability of ϵ, this policy chooses a random action and otherwise chooses the best action. By annealing this probability towards 0 over time, we can ensure a high exploration rate in the beginning and then slowly decrease it to switch focus to the exploitation stage.

In Q-learning the agent follows the ϵ-greedy policy while optimizing the greedy policy, making it an off-policy algorithm. Putting together everything so far, we end up with the following loss function for the base Q-Network algorithm:

$$L_i(\theta_i) = \sum_{(s,a,r,s') \sim U(B)} [(Y^{Target} - Q(s, a, \theta_i))^2]. \tag{2.7}$$

Where Y^{Target} is the target value defined in Equation (2.6).

In SARSA however, as the acronym suggests, we also save the next action to the memory buffer with the tuple (s, a, r, s', a'). This way the agent can both follow and optimize for the same ϵ-greedy policy, making this an on-policy algorithm with the loss function:

$$L_i(\theta_i) = \sum_{(s,a,r,s',a') \sim U(B)} [(Y^{SARSA} - Q(s, a, \theta_i))^2]. \tag{2.8}$$

with

$$Y_t^{SARSA} \equiv r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}; \theta_t^-). \tag{2.9}$$

This should lead to an increase in learning stability due to one less element of the deadly triad present in the system. Because SARSA updates Q in a way that takes into account the random actions the agent sometimes takes it is expected to find a safer, but less optimal policy (Sutton and Barto, 2018) and report a higher average performance during learning.

2.2.6 Dueling Networks

One limitation of the Q-Network approach is that it is not able to estimate the value of a state and the action separately (Wang et al., 2015). This ability can be very useful, since often in states an action has no relevant consequence. The dueling network architecture achieves this by having two streams each predicting different things, instead of one stream predicting both. It is implemented by configuring two hidden layers in parallel, replacing the single hidden layer that is standard in the Q-Network. One of these two layers will estimate the state value V, while the other layer will estimate the action advantages A for all possible actions. The two are merged together according to Equation (2.10). This equation differs from the original paper, since here we do not use any convolutional layers.

$$Q(s,a;\alpha,\beta) = V(s;\beta) + \left(A(s,a;\alpha) - \frac{1}{c}\sum_{a'} A(s,a';\alpha)\right) \tag{2.10}$$

Here, α and β denote the weights of the two fully connected layers and c the number of actions. This equation automatically prevents the layer for the state value from estimating anything related to the action advantages, since the sum of the advantages is kept at zero. The equation results in the Q-values which can be used in the same way as the single stream Q-values. All learning techniques and code can be recycled, and the target network and experience replay is also used. Given the results from the original paper, this modification is expected to boost the maximum performance and especially the speed of learning.

2.2.7 Dueling SARSA

As well as combining dueling networks with Q-Learning, we also combine on-policy SARSA with the dueling network architecture to introduce a new RL algorithm, Dueling SARSA. We expect this algorithm to benefit from both modifications for a twice improved performance and both increased learning speed and learning stability. Just as the other three algorithms, Dueling SARSA also uses the target network and experience replay.

2.2.8 Demonstration Data

Because the reward function defined in Section 2.1.1 provides only sparse and delayed rewards, the agent might require additional guidance to learn to contain the fire in a reasonable training time. To point it in the right direction, we can fill the memory buffer with demonstration data before learning as mentioned in Section 2.2.3. The agent can then use these memories during the learning process to its advantage.

The demonstration data only needs to show the agent how it might be able to collect the containment reward, to this effect we developed a simple algorithm that moves the agent around the fire in a clockwise direction. It does this by choosing randomly from one of two possible actions, unless one of the actions would lead to the death of the agent in which case it chooses the safe action. The two possible actions depend on the position of the agent relative to the fire (This is shown schematically in Figure 2.5).

The environment is reset upon containment as defined in Equation (2.2). Only memories (transitions) leading to successful containments are stored. This results in an average of 35 memories per episode, or containments of the fire, on the 10x10 map, and around 48 memories per episode on the 14x14 map. The number of episodes of demonstration data required can be specified.

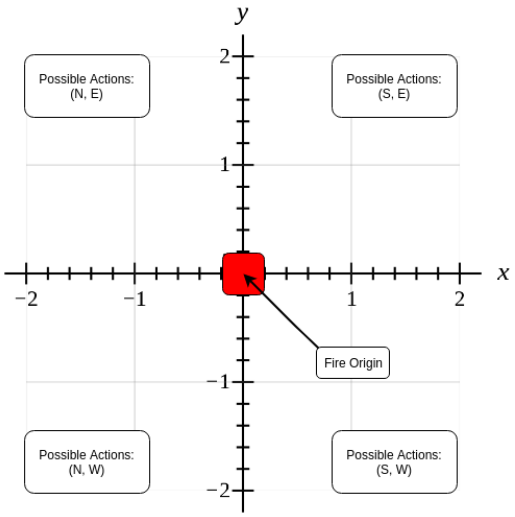

Figure 2.5: The possible actions for the agent to take, based on the position (quadrant) of the agent relative to the origin of the fire

2.3 Experimental Setup

2.3.1 Baseline Algorithm

To be able to reliably compare the performance of the different algorithms, we need a stable baseline. We build on the ideas discussed in Section 2.2.8 to define the algorithm shown in Algorithm 2.1. This algorithm is identical to the demonstration data generation except that it continues until the fire has burnt out, so it does not stop as soon as the fire is contained.

Algorithm 2.1 Baseline algorithm to contain the fire

```
procedure RunBaseline
    totalreward = 0
    while not done do
        action = random(possible actions)
        if action is dangerous then
            action = other possible action
        end if
        reward, done = execute(action)
        totalreward = totalreward + reward
    end while
    return totalreward
end procedure
```

2.3.2 Collection of Results

We investigate the performance of both Q-Learning and SARSA, both with and without the dueling network architecture modification, for a total of 4 different algorithms. The hyper-parameters we used throughout all experiments are shown in Table 2.1.

Each algorithm and parameter combination was run 10 times for 10,000 episodes per run. We compared the baseline performance to 2 map sizes ($N = 10$ and $N = 14$), 3 different amounts of demonstration data (0, 100 and 1000 episodes) and 4 algorithms for a total of 240 runs (260 including the baselines). At the start of each run, the environment is initialized with trees and a single cell at the center of the map is ignited. The agent starts at a random location on a circle centered around the fire with a radius of either 1, 2 or 3 (also randomly selected). Since each run took approximately 4.5 hours on average, they were calculated on the Peregrine computing cluster provided by the University of Groningen.

Memory size	20000
Batch size	32
Target update (C)	20
Gamma (γ)	0.999
Alpha (α)	0.005
Epsilon decay (ϵ)	0.01
Epsilon maximum	1
Epsilon minimum	0.005

Table 2.1: All relevant hyper-parameters used in the training process. These values were selected by performing an informal search using the Q-Learning algorithm without dueling networks. The target is updated every C episodes. The epsilon value is decayed after every episode.

3 Results

It should be noted that all four algorithms are color-coded consistently throughout the plots. The lines represent averages of 10 training runs. The error bars are based on the standard error at that point in time. All tables are based on

the final 2500 episodes of these 10-run averages.

3.1 10-by-10 simulations

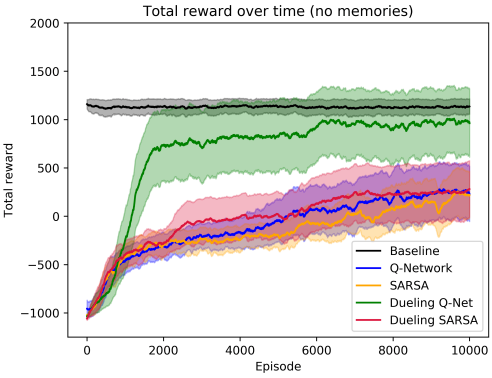

Figure 3.1: 10-run averages given 0 episodes of demonstration data.

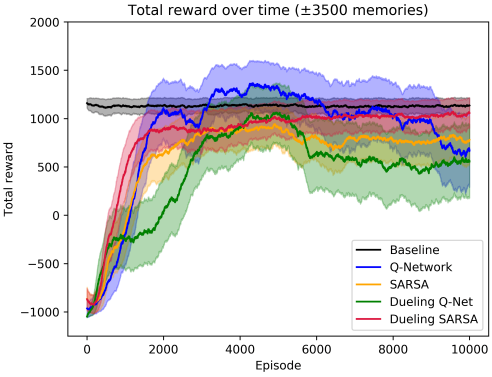

Figure 3.2: 10-run averages given 100 episodes of demonstration data.

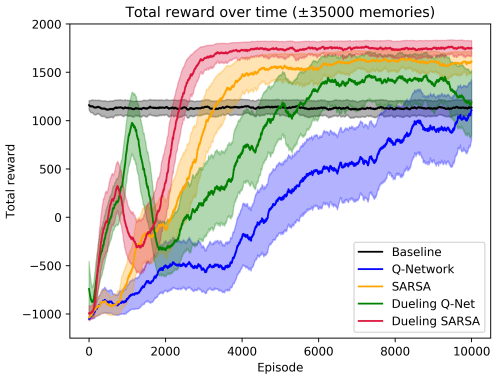

Figure 3.3: 10-run averages given 1000 episodes of demonstration data.

Algorithm	**Average Reward**	**Std. Error**	**Best Reward**
Baseline	**1129**	80	**1387**
Q-Network	221	283	715
SARSA	132	240	563
Dueling *Q*-Network	956	352	1335
Dueling SARSA	241	296	582

Table 3.1: Averages of the last 2500 episodes given 0 episodes of demonstration data. The numbers in bold indicate the highest average and best rewards.

Algorithm	**Average Reward**	**Std. Error**	**Best Reward**
Baseline	**1129**	80	1387
Q-Network	878	357	**1758**
SARSA	776	237	1292
Dueling *Q*-Network	521	378	1535
Dueling SARSA	1031	162	1312

Table 3.2: Averages of the last 2500 episodes given 100 episodes of demonstration data.

Algorithm	Average Reward	Std. Error	Best Reward
Baseline	1129	80	1387
Q-Network	907	343	1696
SARSA	1607*	108	1748
Dueling Q-Network	1369*	276	1826
Dueling SARSA	**1745***	83	**1860**

Table 3.3: Averages of the last 2500 episodes given 1000 episodes of demonstration data. The asterisk (*) indicates the average reward is greater than the average reward of the baseline.

In Figures 3.1, 3.2 and 3.3 we see the three cases where the simulation consists of a grid of 10-by-10 cells and the algorithms are given 0, 100 or 1000 episodes of demonstration data respectively.

Firstly, in Figure 3.1 we see that all algorithms struggle to beat the baseline algorithm. Only the Dueling Q-Networks is able to come near. It offers a greatly increased learning speed and is able to sustain a higher maximum performance level. The other three algorithms perform similarly to each other, but noticeable worse than Dueling Q-Networks.

Secondly, in Figure 3.2 we see the performance of all algorithms come together. The Dueling Q-Networks which performed better than the others before, has now lost its edge and is now the worst performer. The performance of Q-Networks greatly increases when given $\pm$3500 memories compared to none. It surpasses the baseline algorithm temporarily. SARSA and Dueling SARSA do not stand out, but also increased their performances given more memories than before.

Lastly, in Figure 3.3 we see each algorithm performing differently. The Q-Network loses performance compared to the previous configuration, but still performs better than when no memories were given. The Dueling Q-Network did not improve its performance with $\pm$35000 memories and shows a peculiar peak near the 1000th episode. SARSA has increased its performance once again and now beats the baseline algorithm. The same holds for Dueling SARSA which outperforms SARSA. Dueling SARSA shows the same, yet smaller, peak like Dueling Q-Networks. The two algorithms incorporating SARSA offer a noticeably lower standard error.

The results show that all algorithms perform at least reasonably well and respond very differently and sometimes uniquely to the demonstration data given at the start of the training process. Q-Networks performed best with 100 episodes of memories while the performance of Dueling Q-Networks was less dependent on the memories. The latter also showed to have a large spike in performance near the 1000th episode. This is probably due to the large amount of memories it was given. In this way, the algorithm has very few chances to make mistakes and learn from those transitions during its exploration phase. It focuses too much on the demonstrated behaviour and is hindered in trying out actions to explore consequences. This explanation can also hold for why the Q-Network performs worse with 1000 episodes of memories.

SARSA showed to be one of the most consistent and trustworthy algorithms in terms of its response to the memories. In essence, more memories meant a higher sustained performance level. However, the speed at which it learned was decreased. This may also be due to the same reason mentioned; large amounts of memories hinder the algorithm's exploration abilities. More memories also showed to have another positive effect on SARSA, namely its standard error. Higher performance levels came with lower standard errors. The general stability of SARSA compared to Q-Learning can be explained, as mentioned in Section 2.2.5, by the fact that, because it is an on-policy algorithm, we effectively remove one of the three elements of the deadly triad.

We found that combining Dueling Q-Networks and SARSA into Dueling SARSA resulted in the overall best performer, especially in high memory scenarios. It inherits the best of both worlds; the learning speed from the former, and the performance and consistency from the latter. The algorithm also inherited the performance peak from the Dueling Q-Networks,

albeit not as high and slightly earlier. We estimate the maximum possible reward for an algorithm to achieve to be ±1850, since it varies due to the random starting position of the agent. In Table 3.3, we see the algorithm has no problem achieving that maximum reward and having a average reward close to it. Moreover, its standard error there is similar to the baseline algorithm, which is impressive since the baseline is a hard-coded solution.

Tables 3.1 through 3.3 show us that out of the 12 scenarios, only three times an algorithm was able to achieve an average reward higher than the baseline. All three occurred in the setting where 1000 episodes of memories were given. The Q-Network stands out as being the only one with an average reward lower than the baseline.

3.2 14-by-14 simulations

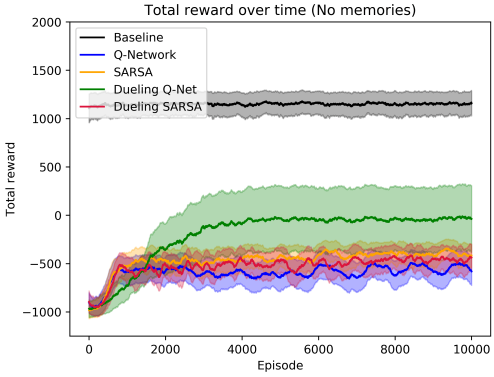

Figure 3.4: 10-run averages given 0 episodes of demonstration data.

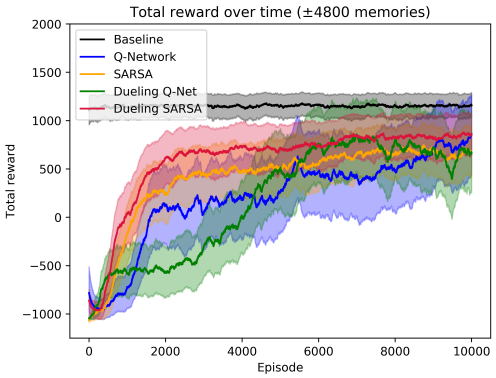

Figure 3.5: 10-run averages given 100 episodes of demonstration data.

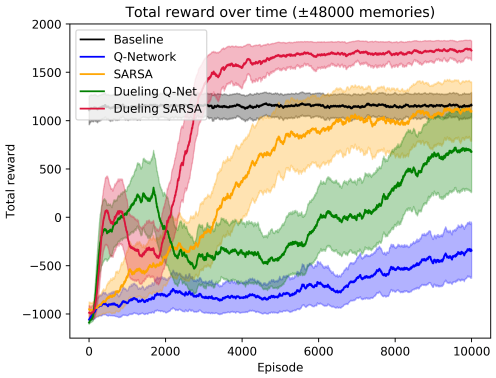

Figure 3.6: 10-run averages given 1000 episodes of demonstration data.

Algorithm	Average Reward	Std. Error	Best Reward
Baseline	**1152**	125	**1513**
Q-Network	-550	144	-139
SARSA	-398	116	-92
Dueling Q-Network	-40	335	349
Dueling SARSA	-455	134	-44

Table 3.4: Averages of the last 2500 episodes given 0 episodes of demonstation data.

Algorithm	Average Reward	Std. Error	Best Reward
Baseline	**1152**	125	1513
Q-Network	652	418	169
SARSA	670	257	1275
Dueling Q-Network	667	404	**1748**
Dueling SARSA	836	224	1249

Table 3.5: Averages of the last 2500 episodes given 100 episodes of demonstation data.

Algorithm	Average Reward	Std. Error	Best Reward
Baseline	1152	125	1513
Q-Network	-459	253	411
SARSA	1057	316	1626
Dueling Q-Network	522	406	1534
Dueling SARSA	**1713***	108	**1846**

Table 3.6: Averages of the last 2500 episodes given 1000 episodes of demonstation data.

In Figure 3.4, 3.5 and 3.6 we see the three cases where the simulation has a grid size of 14-by-14 and the algorithms are given 0, 100 or 1000 episodes of demonstation data.

Firstly, in Figure 3.4 we see results comparable to Figure 3.1, however the performance difference of Dueling Q-Networks compared to the rest has decreased. The best performing algorithm here does not come close to the performance of the baseline.

Secondly, in Figure 3.5 we see results similar to Figure 3.2, but no algorithm is able to beat the baseline like Q-Networks was able to before.

Lastly, in Figure 3.6 we once again see results comparable to Figure 3.3, however only Dueling SARSA is now able to beat the baseline algorithm. SARSA is able to perform at the same level as the baseline in the end. The (Dueling) Q-Networks do not offer good performance, but Dueling Q-Networks does outperform Q-Networks.

When switching to a simulation consisting of a square grid of size 14 instead of 10, we see the algorithms struggle a lot more in general. Since we use a layered state representation, this nearly doubles the amount of inputs from 300 to 588. The relatively small hidden layer used seems to struggle to properly learn and process all information. The number of episodes that the algorithm was allowed to learn for was also kept constant which could explain the low overall performances.

When the algorithms are not given any memories the Dueling Q-Networks is still able to perform better than the rest, but it achieves similar scores to the worst performers in the 10-by-10 simulation. Q-Networks, SARSA and Dueling SARSA seem to barely improve at all. When given 100 or 1000 episodes of memories the Dueling Q-Network improves in both cases, but does not achieve the same levels of performance. The Q-Network shows the same behaviour as before as it performs best with the medium amount of memories given and drops that advantage when given a high amount. This algorithm is also not able to solve the problem. Both Q-Network and Dueling Q-Networks fail to solve the problem and beat the baseline in all cases.

SARSA is still very receptive to memories with its best performance being the scenario where ± 48000 memories were given. Here it is the only algorithm so far that is able to match the baseline algorithm. The only algorithm that is able to consistently beat it is Dueling SARSA in the high memory scenario. It shows performance very similar to the 10-by-10 simulation. The only difference is that it tops out later and suffers from a slightly higher standard error.

One other interesting thing to mention is that in the 14-by-14 simulation both Dueling Q-Networks and Dueling SARSA show a peak in performance at the start of the learning process as well. While the peak is now not as high, both algorithms have a peak of roughly the same size. The peaks are more spread out and Dueling SARSA's peak still happens earlier. In the 10-by-10 simulation both algorithms managed to quickly regain performance and continue learning after this peak, while in the 14-by-14 simulation only Dueling SARSA is able

to continue learning properly.

To conclude this section, we can look at the figures with an asterisk (*) in Tables 3.1 through 3.6. These are the cases where the average score of the last 2500 episodes of an algorithm was greater than the same average of the baseline. In the 24 algorithm/memory/size combinations, this happened only four times. Two of which were obtained by the Dueling SARSA algorithm. With the highest amount of memories it was the only algorithm to solve the task successfully on both the 10-by-10 and 14-by-14 simulations. When looking at just the scenario where the algorithms were given the maximum amount of memories on a 10-by-10 simulation, two other algorithms were able to succeed: Dueling Q-Networks and SARSA. This rhymes nicely with the expectation that combining these two algorithms into Dueling SARSA makes it inherit the best of both worlds.

4 Conclusion

The simulation environment was very simplified. We have the functionality to determine the wind speed and direction but did not have the time to explore those options. There is also support for more cell types such as grass versus trees or rivers, each with different properties for a more complex and realistic environment. We believe this should be investigated in subsequent research, because the system should ultimately prove itself reliable in more complex environments (that is real life).

The reward function is, in its current form, very hard for an agent to learn with. It provides very sparse and delayed rewards, this might be improved to provide a more smooth gradient and allowing for faster and more stable learning, and less reliance on demonstration data. There are also methods of introducing additional rewards while keeping the optimal policy identical (and in the case of multi-agent control, also keeping the Nash equilibria equal) (Ng, Harada, and Russell, 1999). A more general framework that can learn efficient reward shaping without the need for expert knowledge has also been proposed (Zou, Ren, Yan, Su, and Zhu, 2019). Hindsight experience replay (Andrychowicz, Wolski, Ray, Schneider, Fong, Welinder, McGrew, Tobin, Abbeel, and Zaremba, 2017) might be an interesting way to deal with the sparse rewards, it allows the agent to learn from unsuccessful episodes (which there are a lot of) by imagining the negative outcome, in hindsight, to have been the goal all along.

Since all algorithms are based on connectionist reinforcement learning, they are likely to benefit from other improvements proven to be useful. Some we think would perform well include Deep Q-Networks (Mnih et al., 2015); Using a deep network with convolutional layers might extract some meaningful spatial information that it cannot otherwise, like shape of the circle and position of the agent relative to it for example. Prioritized experience replay (Schaul, Quan, Antonoglou, and Silver, 2015) might increase learning speed and performance due to a more efficient sampling of memories, and noisy nets (Fortunato, Azar, Piot, Menick, Osband, Graves, Mnih, Munos, Hassabis, Pietquin, Blundell, and Legg, 2017) have been shown to increase performance by replacing the e-greedy policy with parameterized noise in the network for a better exploration heuristic. For effective combinations of these improvements, see (Hessel, Modayil, van Hasselt, Schaul, Ostrovski, Dabney, Horgan, Piot, Azar, and Silver, 2017). Furthermore, Deep QV-Learning (Sabatelli, Louppe, Geurts, and Wiering, 2018) has shown an increased learning speed as well as performance on several Atari 2600 games compared to Deep Q-Networks, suggesting that keeping track of both the Q- and the V-functions might be beneficial in connectionist RL.

The results may have shown more interesting patterns for longer runs (more than 10,000 episodes). We might have seen the same patterns in performance for the 14-by-14 maps as for the 10-by-10 ones, just on a larger scale. It is also likely that some patterns are not visible even on the smaller map size with 10000 episodes, such as the drop in performance for Q-Learning without Dueling networks.

Acknowledgements

We would like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster. Furthermore, we want to thank dr. Marco Wiering for his guidance throughout the project.

References

M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL `http://arxiv.org/abs/1710.02298`.

J. Houghton, T. Jenkins, and G. Ephraums. *Climate change*. Cambridge University Press, Cambridge (UK), 3 edition, 1991.

E. Kasischke, N. Christensen Jr, and B. Stocks. Fire, global warming, and the carbon balance of boreal forests. *Ecological applications*, 5: 437–451, 1995.

S.J.L. Knegt, M.M. Drugan, and M.A. Wiering. Opponent modelling in the game of Tron using reinforcement learning. In *ICAART 2018 - Proceedings of the 10th International Conference on Agents and Artificial Intelligence*, volume 2, pages 29–40, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Daniel Moura and Eugénio Oliveira. Fighting fire with agents: an agent coordination model for simulated firefighting. In *Proceedings of the 2007 spring simulation multiconference-Volume 2*, pages 71–78. Society for Computer Simulation International, 2007.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning*, volume 99, pages 278–287, 1999.

G. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical report, University of Cambridge, Department of Engineering Cambridge, England, 1994.

M. Sabatelli, G. Louppe, P. Geurts, and M.A. Wiering. Deep Quality-Value (DQV) Learning. *arXiv preprint arXiv:1810.00368*, 2018.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

R. Sutton and A. Barto. *Reinforcement Learning: an Introduction*. The MIT Press, 2 edition, 2018.

J. Tsitsiklis and B. Van Roy. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081, 1997.

Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning.

CoRR, abs/1511.06581, 2015. URL `http://arxiv.org/abs/1511.06581`.

C. Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, 1989.

M.A. Wiering and M. Doringo. Learning to control forest fires. In H. Haasis and K. Ranze, editors, *Proceedings of the 12th international Symposium on 'Computer Science for Environmental Protection'*, pages 378–388, 1998.

M.A. Wiering, F. Mignogna, and B. Maassen. Evolving neural networks for forest fire control. In M. van Otterlo, M. Poel, and A. Nijholt, editors, *Benelearn '05: Proceedings of the 14th Belgian-Dutch Conference on Machine Learning*, pages 113–120, 2005.

Haosheng Zou, Tongzheng Ren, Dong Yan, Hang Su, and Jun Zhu. Reward shaping via meta-learning. *arXiv preprint arXiv:1901.09330*, 2019.

Appendix

Division of Work

In the first three months, when we built the simulation with the five of us, the work was actually pretty well divided and everybody did their part. At this time, Travis worked on the code for fire propagation while Dirk Jelle hunted bugs and tested the simulator. When the group split up and Travis and Dirk Jelle decided to work together, Travis had already ported the simulation from Java to Python. In the next few weeks, Travis did the main part of the work of exploring the OpenAI Gym environments and implementing a basic DQN algorithm. Dirk Jelle kept up with Travis' progression, but was focussed on his last exams at that time.

When the algorithm was able to learn, both of us explored what it could and could not do by running tests and tweaking parameters. We each implemented our own modification solo: Travis did SARSA and Dirk Jelle did Dueling Networks. Once we had an idea of what tests we wanted and needed to run, Dirk Jelle spent the most time learning how to use Peregrine. The work regarding the generating and processing of the results was mainly done by Dirk Jelle.

The work required for the presentation at the Bachelor's Symposium was divided equally. However, since Travis focussed more on the theory and the algorithms, he presented that part. Also Dirk Jelle presented what he was most familiar with, which were the simulation and the results. This also holds for the writing of the thesis. Travis mainly wrote the Methods and Conclusion, while Dirk Jelle mainly wrote the Introduction and Results. We both made sure that no section was written solely by one person.

An important note is that the neither of us excluded the other when doing some part of the project. We kept in touch at least every few days and always had the opportunity to ask questions to each other. While Travis wrote the bulk of the code for the learning algorithm, we both discussed how it should work and why. When Dirk Jelle processed the data to create results, we both spoke about what data goes into which type of plots/tables and why.

In conclusion, this project proved to be a lot of work. While we tried our best to do every task together, we did not always succeed. In the end, neither of us feel like the total workload was divided unfairly. Both of us worked hard, but not always at the same time. We hope this small summary is able to give some insight into who did what for the last six months.