UNIVERSITY OF AMSTERDAM

MSc Artificial Intelligence

Master Thesis

Recurrent Neural Networks for Reinforcement Learning: an Investigation of Relevant Design Choices

by

Jaimy van Dijk

11039698

July 14, 2017

36 EC
January 2017 - July 2017

Supervisor:
Dr. Frans Oliehoek

Assessor:
Dr. Maarten van Someren

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Abstract

When solving real-world problems like traffic control with reinforcement learning, the observation of the environment is often noisy or simply incomplete. Traditional forms of Q-learning are ill-equipped to handle this kind of problems called Partially Observable Markov Decision Problems (POMDPs). When aiming to solve a POMDP with function approximation a lot of research has made use of recurrent neural networks. With the recent successes of *Deep Q-Networks* it was inevitable that Deep *Recurrent* Q-Networks would be introduced. Even though the results of this model are very promising, most researchers make arbitrary design choices when it comes to the recurrent networks. This thesis aims to better investigate such design choices in the context of a traffic control problem. The second contribution of this thesis is a new architecture called *Action Motivation DRQN* (AMDRQN). This model appends the probability of the previously chosen action to the observation input. This enables an agent to make decisions based on sequences of observation-action pairs but is also creates a new path of gradients over which the parameters can be updated using information about the action probabilities.

Acknowledgements

I would like to thank my supervisor, Frans Oliehoek, for being my mentor during this project. I appreciate all the inspiring and insightful meetings we have had and I really enjoyed working together. Additionally, I would like to thank Maarten van Someren for agreeing to be my assessor. I am very grateful to Elise van der Pol for letting me use her code base and explaining the functionality of SUMO to me. I would like to express my thanks to Shimon Whiteson and Jakob Foerster for brainstorming about AMDRQN. Your insights helped me get a better understanding of the inner workings of the model which eventually led to the derivation of the new gradient estimation. Finally, I would like to thank Rian van Son, Hans van Dijk, Wiggert Altenburg and Robin van Dijk for their endless support and encouragement.

Contents

1 Introduction

In the last couple of years, Artificial Intelligence has made a lot of progress. One of the developments that made this progress possible is *deep learning*, a relatively new area in machine learning that uses increasingly complex models to approximate functions. The application of these models to the field of *reinforcement learning* has resulted in important milestones like defeating Lee Sedol, considered to be the greatest player of the game Go of the past decade. But the deep learning models proved to be able to learn much more tasks [22, 17]. This resulted in a lot of research on *deep reinforcement learning*.

The success of deep learning in game environments, inspired research to explore more difficult problems: applying these models to real world problems. Traffic control is a problem that every city in every country has to deal with. It is a problem where, given a situation, the right action might not be known but the effect of a good action is. The correct moment to change a traffic light from green to red might not be obvious but it is clear that if the throughput of the crossing is high, then the behaviour of the traffic lights is correct. This makes it well suited for reinforcement learning and has been the subject of deep reinforcement learning research [25, 26].

This thesis continues on earlier work [35] in which an *agent* is in control of the traffic light configuration of an intersection. Based on its observations of the environment it was tasked to optimize the traffic throughput. This problem was defined and solved as being a *Markov Decision Process*. This framework assumes that the given input, the *state*, contains all the information about the environment. An observation of a real world environment is, however, often noisy and incomplete making the problem a *Partially Observable Markov Decision Problem*. The model used to solve the traffic control problem, the *deep Q-network* [22], has been known to perform worse when used on incomplete state information. To overcome this shortcoming, the input of the model contains information about two consecutive time steps. This transforms the partially observable problem, to a fully observable problem. Another solution would be to use a *deep recurrent Q-network* that is capable of handling partial observability [7].

The aim of this thesis is to extend earlier work on solving the traffic light control problem with deep reinforcement learning and to get a better understanding of using *recurrent neural networks* in a reinforcement learning setting.

1.1 Contributions

Based on earlier work [35], the agent observes the traffic environment as a top-down image. This is used as the input of the deep learning algorithm. This thesis will answer the following research question:

Q_1. *Can a deep reinforcement learning agent, using a recurrent neural network, learn to optimize the flow of traffic based only on one top-down image per time step of the traffic situation?*

When using a recurrent neural network as function approximation, a hidden state is passed down through time that contains information about the past. How many time steps it tries to remember is a parameter that can be changed (see Section 3 for full explanation). The considered research question is:

Q_2. *How does the number of time steps the agent learns to remember affect the performance?*

The agent makes its decision based on a history of observations but does not get any information about how and why these sequences of observations came to be. Giving the agent information about the actions that have been taken answers the question of *how* the observation sequences came to be, but not *why*. In this thesis, a new architecture is proposed called *Action Motivation DRQN*. To test this new architecture the following question is answered:

Q_3. *What is the effect of the addition of action probabilities to the input of the model on the performance?*

Using the computation graph of a deep recurrent Q-network, an unbiased gradient estimation can be derived and a surrogate loss function that can be used to approximate this gradient (see Section 9 for explanation). This thesis answers the following question about this subject:

Q_4. *Does using an unbiased gradient estimation stabilize and/or improve deep recurrent Q-learning?*

In summary, this thesis' contributions are: research on applying deep reinforcement learning to the traffic control problem using a recurrent neural network. It explored the impact of different parameter settings on the performance of the trained models. In addition it uses a surrogate loss function to approximate the unbiased gradient of a partially observable Markov decision problem. Lastly, it introduces a new architecture Action Motivation DRQN that uses the probability of an action as input for the recurrent neural network.

1.2 Outline

The background information about reinforcement learning, deep learning and deep reinforcement learning are explained in Section 2. The experimental setup is explained in Section 4 and the results of the various experiments are presented in Sections 5, 6 and 7. The new architecture called Action Motivation DRQN is presented in Section 8 as well as its performance. Section 9 explores the theoretical side of using RNNs and the findings are explored in the traffic control scenario. Section 10 compares this thesis to related work on deep recurrent Q-learning. finally, Section 11 discusses the found results and Section 12 concludes the thesis with suggestions about future work.

2 Background

This section will explain the background information needed to understand the concepts of deep reinforcement learning.

2.1 Deep Learning

Deep learning is an area of machine learning that focuses on neural networks with many layers. Recent successes with these deep neural networks have resulted in a lot of research to make these models faster and more reliable.

2.1.1 Neural networks

Neural networks are a family of parametric, non-linear and hierarchical learning functions. Given a dataset D they need to find the optimal parameters θ^* that minimize some *loss function*. These models are called networks because they are a collection of functions that can be represented as an acyclic graph [6]. This graph is divided into layers and each layer represents a computation of the form:

$$h_1 = f_1(W_1 \cdot x + b_1) \tag{1}$$

Here x is the multidimensional input of the model that is mapped to the *hidden unit* h_1 using weights $W_1 \in \theta$ and biases $b_1 \in \theta$. The function $f_1(\cdot)$ is called an *activation function*. The output of one layer can be used as input for another layer.

$$h_2 = f_2(W_2 \cdot f_1(W_1 \cdot x + b_1) + b_2) \tag{2}$$

Hence the *hierarchical* aspect of neural networks. The field of *deep learning* focuses on neural networks with a large number of these layers because they are capable of approximating more complex functions. However, as the number of layers and parameters grows the model becomes more difficult to train. A lot of research is dedicated to developing methods to better train these deep neural networks.

2.1.2 Recurrent Neural Networks

Recurrent neural networks (RNN) are part of the family of neural networks and are able to process sequential data. To understand the information that is incorporated in a sequence, an RNN needs *memory* to know the *context* of the data. Information about the past is passed through the network using *hidden states*. The RNN can be represented as a function with parameters θ, that computes its output given input x_t and h_{t-1}:

$$h_t = f(x_t, h_{t-1}|\theta) \tag{3}$$

Where h_{t-1} is the hidden state of the network at the previous time step [6]. Another important feature of a recurrent network is *parameter sharing*. The parameters θ used in Equation 3 are shared over all time steps. This means that every step in a sequence is processed the same way and therefore promotes generalization to unseen sequence lengths.

Backpropagation Through Time A recurrent neural network is updated using *backpropagation through time* (BPTT). Because the result at time t depends on all previous time steps, the error calculated from the result needs to be propagated back through all these steps. For a finite sequence, this could just be the length of the given data. It is, however, very expensive to go very far or even infinitely back in time. To keep BPTT computationally tractable the number of time steps is truncated to a finite number τ.

Long Short-Term Memory Networks Recurrent neural network is the name of a family of network that all have their own definition of Equation 3. One of these networks is the Long Short-Term Memory (LSTM) network. More basic recurrent networks struggle with long-term dependencies in a sequence. LSTM networks were designed to be able to learn these long-term relations without overlooking the short-term dependencies as well [8]. These networks have been very successful on various problems and are therefore very popular. The main idea behind LSTM is its *cell state* C_t. This state is sparsely updated and can therefore be seen as the "long-term memory". As the name suggests LSTM also has a short-term memory h_t. Given input x_t and hidden state h_{t-1}, the cell state C_{t-1} is updated to C_t. The new cell state is then combined with x_t and h_{t-1} to compute the output h_t.

2.1.3 Optimization Algorithms

To learn with neural networks the parameters are optimized with *gradient descent* [6].

$$\theta^{t+1} = \theta^t + \alpha \nabla_\theta \mathcal{L} \tag{4}$$

In words, the parameters are updated with the gradients of the loss function $\mathcal{L}$ with respect to the parameters discounted by the *learning rate*. The computation of the gradients through the neural network is called *back-propagation*.

Stochastic Gradient Descent. Computing the gradients over the entire dataset D can be very expensive. Certainly when the dataset is large and the network is deep. Instead, *mini-batches* are sampled stochastically from the data and the parameters are updated like this:

$$\begin{aligned} B &\subseteq D \\ \theta^{t+1} &= \theta^t + \frac{\alpha}{|B|} \sum_{i \in B} \nabla_\theta \mathcal{L}_i \end{aligned} \tag{5}$$

The size of the mini-batches determine the variance of the gradients. If the size is too small the variance might guide the parameters in the wrong direction, but a very large batch size might prevent the model of getting out of a local minimum.

ADAM The optimization function is an important part of learning with neural networks. Because of that, there has been a lot of research into making functions that are more reliable than standard stochastic gradient descent. One of these optimization functions is ADAM (adaptive moment estimation) [11].

$$\begin{aligned} m_{t+1} &= \beta_1 \cdot m_t + (1 - \beta_1) \cdot \nabla_\theta \mathcal{L} \\ v_{t+1} &= \beta_2 \cdot v_t + (1 - \beta_2) \cdot \nabla_\theta \mathcal{L}^2 \end{aligned} \tag{6}$$

Where $\nabla_\theta \mathcal{L}^2$ is the elementwise square of the gradient vector. These values estimate the mean (m_t) and the variance (v_t) of the gradients. Since these vectors are initialized with zeros, they are biased towards zero. This bias is counteracted by bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$.

$$\begin{aligned} \hat{m}_{t+1} &= m_{t+1} / (1 - \beta_1^{t+1}) \\ \hat{v}_{t+1} &= v_{t+1} / (1 - \beta_2^{t+1}) \\ \theta^{t+1} &= \theta^t - \alpha \cdot \hat{m}_{t+1} / \left(\sqrt{\hat{v}_{t+1}} + \epsilon \right) \end{aligned} \tag{7}$$

2.2 Reinforcement Learning

This section introduces terms and concepts used in the field of reinforcement learning.

2.2.1 Markov Decision Processes

A Markov Decision Process (MDP) is a mathematical framework that describes decision making processes. It formalizes such a process by defining an environment as a set of *states* $\mathcal{S}$, a set of *actions* $\mathcal{A}$ that can be executed by an *agent*, a *reward function* $\mathcal{R}(s, a)$ that specifies reward r for taking action a in state s and a *transition function* $\mathcal{T}(s'|s, a)$ that describes the probability of ending up in state s' given that action a was taken in state s [32]. A key assumption of the MDP framework is that the *Markov Property* is satisfied, saying that the transition function only depends on the current state and the action that was taken. Formally, this is described as:

$$P(s_{t+1}|s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, ..., s_0, a_0, r_0) = P(s_{t+1}|s_t, a_t) \tag{8}$$

The agent needs to solve the specified problem by maximizing the received reward over time. This target is specified as the *expected return*:

$$R_t = r_{t+1} + r_{t+2} + ... + r_{t+T} \tag{9}$$

Where T is the final time step of the process. In the context of traffic light control the process has no end (e.g. $T = \infty$). In that case the expected return we are trying to maximize could become infinite. That is why the reward is *discounted* over time, giving rewards further in time less value than if we would get them immediately. The new objective of the agent is to maximize the *discounted return*:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{10}$$

where γ is called the *discount rate* with a value $0 \leq \gamma \leq 1$. The discount rate determines how far into the future the agents learns to plan. To optimize its solution, the agents needs to find a *policy* π that describes a strategy of selecting an action that maximizes the discounted reward given a state. A policy can be *deterministic*, meaning that π is a function that maps a state to an action. If π is *stochastic* $\pi(s)$ is a probability distribution over the action set $\mathcal{A}$.

2.2.2 Value Functions

Most reinforcement learning algorithms are based on the estimation of the so called *value function*. These functions estimate "how good" a certain state is. Given a policy π, the value of a state is the expected discounted return when starting in state s and following policy π.

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}\left\{R_t | s_t = s\right\} \\ &= \mathbb{E}_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\} \end{aligned} \tag{11}$$

$V^{\pi}$is called the *state-value function for policy* π [32]. Similar to Equation 11 is the function that describes the value of taking action a in state s under a policy π called the *action-value function for policy* π (Q-function):

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{\pi}\left\{R_t | s_t = s, a_t = a\right\} \\ &= \mathbb{E}_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\} \end{aligned} \tag{12}$$

An important feature of Equation 12 is its *recursive property*. For any policy, state and action the following condition holds between the value of (s,a) and its possible successor states:

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\} \\ &= \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a)\left[\mathcal{R}(s,a) + \gamma V^{\pi}(s')\right] \end{aligned} \tag{13}$$

This equation defines the *Bellman Equation* of $Q^{\pi}(s,a)$ and can be used to solve the MDP.

2.2.3 Q-learning

The goal of the agent is to find the *optimal policy* π^*. There may be more than one of these policies but they all share the same optimal action-value function $Q^*(s,a)$ defined according to the *Bellman optimality equation*:

$$Q^*(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a)\left[\mathcal{R}(s,a) + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a')\right] \tag{14}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Any policy that is *greedy* with respect to $Q^*(s,a)$ is an optimal policy. One way to solve a MDP is to approximate the Q-value with temporal difference (TD) learning [34] and use a greedy policy to select actions. TD learning is a *model free*, learning method. This means that it does not need a model of the environment's dynamics. Just like Monte Carlo methods [27] it learns from experience. The agent uses these experiences to update a table containing the Q-values for all possible s,a-pairs. These values are updated using the following equation:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right] \tag{15}$$

The Q-value of action a_t in state s_t is updated with the difference between the target value and the estimated value. The target of TD learning is based on the existing estimate of $Q(s_{t+1},a)$ and is therefore called a *bootstrapping* method [32]. Q-learning [38] is an *off policy* TD algorithm. Unlike its on policy counterpart SARSA [28], which estimates $Q^{\pi}(s,a)$ for the current policy π, Q-learning directly approximates $Q^*(s,a)$ regardless of the policy that is being followed. Tabular Q-learning has been shown to converge given an infinite amount of time.

2.2.4 Policy Search

Value function approximation like Q-leaning [38] and SARSA [28] are not the only methods that can be used to find the optimal policy. Instead of approximating the value function, the policy can also be estimated directly. The methods that do this are called *policy search* methods [33]. A number of examples of this method exist and one of them is REINFORCE [39]. This algorithm tries to train the policy to maximize the expected reward. REINFORCE finds an unbiased estimate of the gradient without using a value function. Its gradient estimators do, however, suffer from a large variance. There are other methods that combine policy search and value function approximation: the *actor-critic* method [12]. The policy is called the *actor* and the value function is the *critic*. The critic is trained using an on-policy TD-learning algorithm and the actor is updated using a function that is similar to the one used in REINFORCE. These policy gradient methods have successfully been applied to deep reinforcement learning as well. Mnih et al. applied asynchronous advantage actor-critic (A3C) to the Atari Learning Environment with great success [21] while Lilicrap et al. used it to train an agent in a continuous environment [17]. Actor-critic methods have also been used in a traffic control setting [25].

2.2.5 Partial Observability

Up until now it was assumed that the state of the environment was fully observable for the agent. In real-world problems, however, this is usually not the case. Instead of a state, the agent receives only an estimate of the environment state. This *observation* of the environment is o with probability $\mathcal{O}(o|s)$. These problems are called *Partially Observable Markov Decision Processes* (POMDP). The observations no longer satisfy the Markov Property but the underlying states still do [31]. This means that given only observations, the agent needs knowledge about its past to solve the problem. For example, a robot cannot catch a ball by looking at it for one instant, it needs to know the ball's trajectory. To deduce this information, the agent needs to know where the ball was at an earlier moment. Normal Q-Learning does not have the mechanisms to solve a POMDP but a lot of research has focussed on estimating the underlying states from the observations [24, 3] or sequences of observations [19, 20]. Other research focused on using methods that are not as affected by the non-Markovian nature of the observations [9].

2.3 Deep Reinforcement Learning

This section explains how deep learning methods are applied in the field of reinforcement learning.

2.3.1 Deep Q-networks

In real-world problems both the state- and action-space could become very large, too large to enumerate over s, a-pairs like in tabular Q-learning. A solution to this problem is to let go of the tabular representation and to represent them implicitly using a parametrized function that is trained to approximate the Q-function. Gradient descent method have been applied on various MDPs and have produced good results even though there is no guarantee that they converge [34, 1]. These methods usually perform best in domains where input features can be hand-crafted. One of the benefits of deep neural networks is that they can learn from raw input data. When applied to an MDP, a deep neural network was able to solve it using only the pixels as input [23]. This model is called the *Deep Q-network* (DQN) where a deep neural network is used as function approximation for the Q-function. The parameters θ can be trained using gradient descent methods by minimizing the *Mean Squared Error (MSE)* [16] (see Section 2.1). In the case of Q-learning, the error is the TD-error.

$$MSE(\theta) = \mathbb{E}_{(s,a,r,s')}\left[\left(r + \gamma \max_{a'} Q(s', a'|\theta) - Q(s, a|\theta)\right)^2\right] \tag{16}$$

Unfortunately, the use of deep neural networks comes at a cost of stability. In [23] two solutions for these stability issues were proposed: experience replay and a target network that is only periodically updated. The algorithm for DQN with these solutions is described in Algorithm 1.

Experience Replay In sequential decision making problems the subsequent observations are highly correlated meaning that (s_t, a_t) influences the probability of (s_{t+1}, a_{t+1}). This breaks the assumption that the data points the deep neural network is trained on are independent and identically distributed. This assumption underlies many machine learning techniques. Experience replay [18] tries to solve this problem by storing transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ in a replay memory D. By uniformly sampling transitions from D into a mini-batch and using that to train the network, the data points are once again distributed independently. How experience replay is used for training can be seen in Algorithm 1 on lines 17 to 22.

Freezing Target Network Using a neural network as a function approximation means that the s, a-pairs are no longer stored in a table but implicitly represented in the model parameters θ. These parameters are updated globally, meaning that an update based on one s, a-pair can affect other Q-values as well. During training, the parameters are updated at every time step using those same parameters to calculate the target value. As a result of changing θ, the value of $Q(s', a'|\theta)$ changes as well. An analogy for this situation is a dog chasing its own tail: every time it makes a move to catch it, the tail moves as a result of its own action. For a neural network this means that the learning process can oscillate or learning could even be obstructed entirely. In order to prevent this from happening the parameters used to calculate the target value are frozen for a set number of iterations [23]. After a so called *freeze interval* the parameters θ^- set to be equal to the updated parameters (see lines 14 to 16 in Algorithm 1).

Algorithm 1 Deep Q-Learning algorithm with experience replay

Initialize replay memory D with capacity N
Initialize action-value function parameters θ_0 with random weights
Set target action-value function parameters $\theta^- \leftarrow \theta_0$
$i \leftarrow 0$
while $i <$ max iterations **do**
    $s \leftarrow s_0$
    while $s \neq$ terminal **do**
        With probability ϵ select random action a
        Otherwise $a = argmax_{a'} Q(s, a'|\theta_i)$
        Take action a
        Observe next state and reward (s', r)
        Store transition (s, a, r, s') in D
        $s \leftarrow s'$
        if i % freeze interval == 0 **then**
            $\theta^- \leftarrow \theta_i$
        end if
        $B = \{(s_j, a_j, r_j, s'_j)\}_{j=1}^{\text{batch size}} \subseteq D$
        for each $(s_j, a_j, r_j, s'_j) \in B$ **do**
            $y_j = r_j + \gamma \; max_{a'} Q(s'_j, a'|\theta^-)$
            $\mathcal{L}_j = \big(y_i - Q(s_i, a_i|\theta_i)\big)^2$
        end for
        $\theta_{i+1} = \theta_i + \frac{\alpha}{|B|} \sum_{j=1}^{\text{batch size}} \nabla_{\theta_i} \mathcal{L}_j$
        $i = i + 1$
    end while
end while

3 Deep Q-learning in P.O. Environments using RNNs

As discussed in Section 2.2.5, normal Q-learning methods do not have the means to deal with partially observable data and the same is true for DQN [7]. Also discussed in Section 2.2.5 was that some research tried to estimate the underlying state of a POMDP from sequences of observations and recurrent neural network are a natural fit to this kind of data [2, 19]. After the success of applying deep neural networks as function approximations in Q-learning, the recurrent version of the model was introduced: *Deep Recurrent Q-learning* (DRQN) [7]. Even though the results were very promising, some arbitrary design choices were made. This thesis aims to better investigate such design choices in order to make better use of RNNs in deep Q-learning.

History Size In Section 2.1.2 it was explained that the learning process of a neural network is truncated to a finite number of time steps τ. It could be said that the more time steps an agent is trained to remember, the better its performance will be. More time steps means more data and more data could bring more noise into the system, possibly confusing the agent instead of giving it more information. Research question Q_2. calls for an investigation of the effect of the parameter τ and the related experiments are presented in Section 5.

Sequential Updates Just as DQN, DRQN is trained with a replay memory [18]. But instead of training on a mini-batch of transitions, DRQN is trained on a mini-batch of sequences of transitions. The number of transitions in one sequence should be the same as the number of time steps the BPTT process is truncated on (e.g. sequences of length τ). These sequences are sampled uniformly at random from the replay memory D with the condition that the transitions in one arrangement all belong to the same episode. At the start of an update the hidden state of the LSTM is set it to zero [7]. At every step of this sequence, the squared TD-error can be calculated. Whether it is better to use all, some or only the error at the final time step of a sequence will be explored in Section 6.

New DRQN Architecture This thesis introduces a new architecture for DRQN: *Action Motivation DRQN*. This architecture not only uses sequences of observations to approximate the Q-function but uses sequences of action-observation pairs. The performance of this architecture and thus answer to Q_3. will be discussed in Section 8.

Unbiased Gradient Estimation In Section 9 the process of deep recurrent Q-learning is explored theoretically. Working with time dependent data changes more than just the type of model that should be used.

These changes result in a new, unbiased gradient estimation that is applied in practice answering research question Q_4.

4 Experimental Setup

This chapter explains the setup of the single agent traffic control problem. This approach extends earlier work [35], in which the DQN algorithm was applied to different scenarios of traffic light control. The results of this work are used as a baseline for new experiments in which this problem is treated as a POMDP instead of a MDP.

4.1 Traffic Light Control

In traffic control the urban infrastructure is a network of roads and intersections. Each of these intersections is controlled by one agent that needs to optimize the throughput of the network while minimizing the number of collisions and traffic jams.

4.1.1 SUMO

The environment is simulated using the open source traffic simulator SUMO [13]. SUMO uses an extension of the stochastic car-following model developed by Stefan Krauß[14]. This model is assumed to be collision-free because the cars keep enough distance between each other to be able to respond to emergency brakes. In the case that a collision occurs anyways, SUMO teleports the involved vehicles to a new location in the network. The same method is applied to vehicles that have not moved for a long time. This makes it difficult to discriminate between collisions and traffic jams. Therefore, the action space of an agent is limited to legal actions that cannot cause intersecting roads to cross at the same time.

4.1.2 Yellow Time

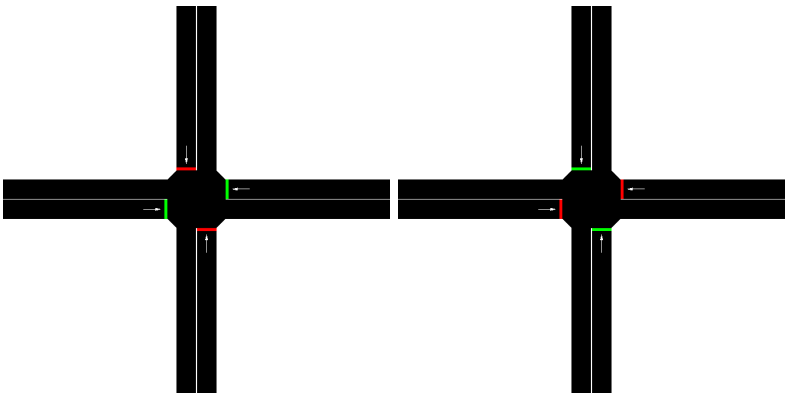
When a traffic light goes from green to red, there is a time in between that the light is yellow. The amount of time necessary to safely let the vehicles slow down before a red light can be calculated according to the traffic engineers handbook [40].

$$Y_t = t + \frac{V}{2a + 2Gg} \tag{17}$$

where Y_t is the yellow time in seconds, t is the reaction time of the drivers (set to 1s), V is the maximum allowed speed which is 70 km/h, a is the deceleration rate (set to $\sim$ 3m/s), G is the acceleration due to gravity (9.81m/s) and g is the grade of approach (set to 0). Given these values, the yellow time would be ≈ 4.2 seconds in this thesis. Thus, when an agent takes the action that requires the lights to go from green to red, those lights go to yellow for $Y_t = 4$ time steps.

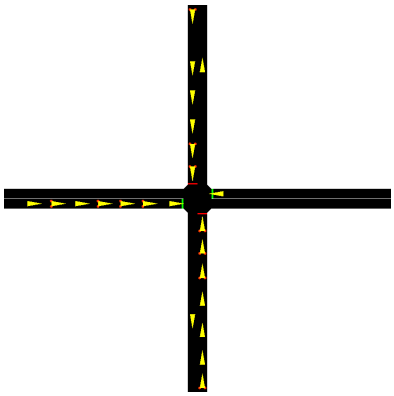
4.1.3 Single Agent Scenario

Because a single agent is trained, only one intersection needs to be simulated. The roads are all two-way streets of 500 meters long. In this scenario each compass point has a probability of 0.1 to add a vehicle to the respective lane every time step, 3600 time steps long. The vehicles are not allowed to turn on the intersection, making this the most simple scenario for the agent (see Figure 1). As discussed before, the action space of the agent is restricted to legal actions. In this scenario, that results in the following actions: *rgrg* (red, green, red, green) and *grgr* (green, red, green, red).



(a) (red, green, red, green) (b) (green, red, green, red)

Figure 1: The legal actions an agent can take in the described scenario.



(a) Snapshot of the simulation.

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

(b) Corresponding 8×8 position matrix.

Figure 2: Example of binary position matrix based on vehicle positions.

4.2 State Representation

The agent observes the environment just as a person would do looking at SUMO, as a top-view of the intersection. Every time step (one simulated second) the agent receives a snapshot of the current situation at the crossing. This input only shows the current positions of the cars and thus is an observation of the underlying state of the traffic situation. In previous work [35], the agent was given multiple, stacked snapshots as input in order to approximate the state of the environment and solve this problem as an MDP. This work aims to solve the traffic control problem as a POMDP, giving the agent only the current snapshot as input. In this thesis two types of snapshots are used.

4.2.1 Position Matrix

The top-view of the intersection is translated to an $n \times m$ binary matrix where the location of a vehicle is marked by a 1 in the matrix [26] (see Figure 2). The SUMO environment is continuous, meaning that the positions of the vehicles are approximated in the fixed-sized grid. As a result, some information can be lost: cars that are close together will be assigned to the same position in the matrix and will be interpreted as a single vehicle. It is also possible that if a car is close to the edge of the environment, it will not be represented in the matrix at all. The higher n and m are (e.g. the higher the resolution), the preciser the position matrix will become.

4.2.2 Position Matrix with Light Information

Knowing where the vehicles are is obviously relevant information for the agent, but information about the current configuration of the traffic lights might also be relevant. This information is added as floats to the position matrix at the locations of the traffic lights. The different colors are encoded as follows: green as 0.8, yellow as 0.5 and red as 0.2. These numbers were chosen to be between 0 and 1 with the same step size between them [35].

4.3 Reward Function

As stated before, the goal of an agent is to maximize the throughput of the network while minimizing the number of traffic jams and collisions. By restricting the action space of the agent to legal actions and the assumption that SUMO is a collision-free model, the number of collisions is already minimized to 0. This means that if a teleportation occurs in the simulation, it is caused by a traffic jam. In [36], the reward function looked like this:

$$r_t = -0.1c - 0.1 \sum_{i=1}^{N} j_i - 0.2 \sum_{i=1}^{N} e_i - 0.3 \sum_{i=1}^{N} d_i - 0.3 \sum_{i=1}^{N} w_i \tag{18}$$

Where N is the number of vehicles in the environment, c is a boolean indicating whether the traffic light configuration has changed since the last time step, j is the number of teleports, e is the number of emergency stops (deceleration of more than $4.5 m/s^2$), d is the delay of a vehicle defined as $1 - \frac{\text{vehicle speed}}{\text{allowed speed}}$ and w is the penalty for waiting one (-0.5) or more than one time step (-1.0).

After some preliminary runs it became clear that with a yellow time of 4 seconds, no emergency stops occurred. There were also no teleports during training because an ϵ-greedy policy is used for action selection and

apparently when $\epsilon = 0.1$ the vehicles keep moving enough to never trigger a teleportation. These elements of Equation 18 were dropped, together with c as that did not have a lot of impact on the value of the reward. This resulted in a new reward function that was used throughout this thesis.

$$r_t = -0.5 \sum_{i=1}^{N} d_i - 0.5 \sum_{i=1}^{N} w_i \tag{19}$$

4.4 Architecture

The architecture of the Deep Recurrent Q-Network is the same as described by Mnih et al. in a publication presented at NIPS [22], but with the fully connected layer of 256 nodes replaced by an LSTM layer of the same size (see Figure 3). The choice for the location of the recurrent layer was made based on results from [7] that indicate that LSTM performs better when using the information directly from the convolution layers.

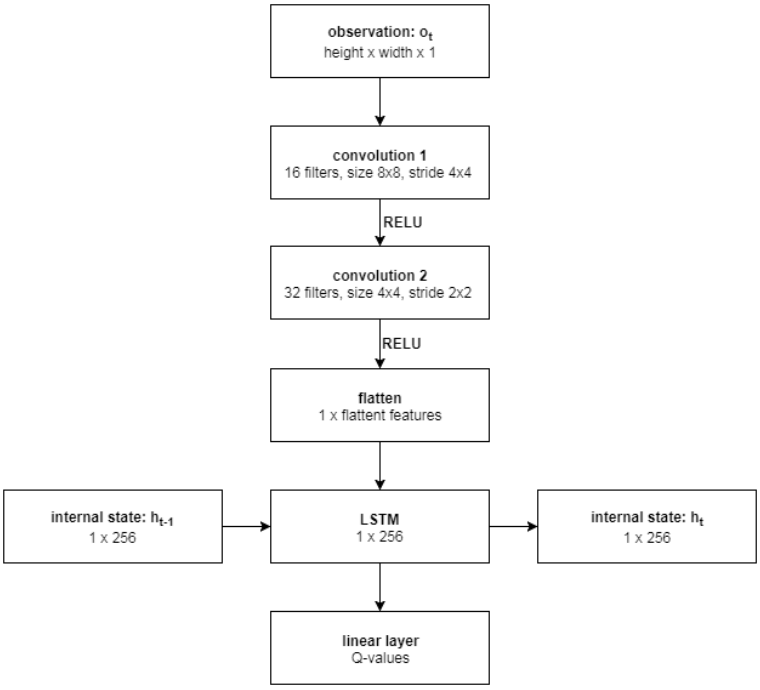


Figure 3: DRQN applies convolution two times on an input image with one channel. The activations are processed by an LSTM, which uses internal states to pass information along to itself over time. The LSTM outputs are translated to Q-values by a fully connected layer.

4.5 Training

The starting point of this thesis is based on earlier work [35] in which a DQN agent was trained to find the optimal policy for the scenario that is described above. In this research, various parameters have been tested and these results were used to decide upon the hyper parameter values that are described in Table 1.

Parameter	**Value**
Optimizer	ADAM
Replay memory size	30,000
Batch size	32
Learning rate (α)	0.00025
Exploration rate (ϵ)	0.1
Discount factor (γ)	0.99
Freeze Interval	30,000
State matrix size	84×84
State matrix type	Binary + light information

Table 1: The parameters used to train the Deep (Recurrent) Q-networks.

The DQN agent is trained using Algorithm 1 and an ϵ-greedy policy. During training $\epsilon = 1$ until the replay memory is filled, after which ϵ will be as specified in Table 1. The same parameters were used for training the DRQN models but the algorithm that is used to train this agent is slightly different (see Algorithm 2).

Algorithm 2 Deep Recurrent Q-Learning algorithm with experience replay

Initialize replay memory D with capacity N
Initialize action-value function parameters θ_0 with random weights
Set target action-value function parameters $\theta^- \leftarrow \theta_0$
$i \leftarrow 0$
while $i <$ max iterations **do**
    $t \leftarrow 1$
    $o_t \leftarrow \mathcal{O}(s_t)$ an observation is only part of a state
    $h_0 \leftarrow \mathbf{0}$
    while $o_t \neq$ terminal **do**
        With probability ϵ select random action a_t
        Otherwise $a_t, h_t = argmax_{a'} Q(o_t, h_{t-1}, a'|\theta_i)$
        Take action a_t
        Retrieve next observation and reward (o_{t+1}, r_t)
        Store transition (o_t, a_t, r_t, o_{t+1}) in D
        $t = t + 1$

        if i % freeze interval == 0 **then**
            $\theta^- \leftarrow \theta$
        end if
        $B = \{(s_j, a_j, r_j, s'_j)...(s_{j+\tau}, a_{j+\tau}, r_{j+\tau}, s'_{j+\tau})\}_{j=1}^{\text{batch size}} \subseteq D$
        for each sequence $(s_j, a_j, r_j, s'_j)...(s_{j+\tau}, a_{j+\tau}, r_{j+\tau}, s'_{j+\tau}) \in B$ **do**
            $h_{j-1} \leftarrow \mathbf{0}$
            for $k = j$ **to** $k = j + \tau$ **do**
                update the hidden state $h_k = Q(s_k, h_{k-1}|\theta_i)$
            end for
            $y_j = r_{j+\tau} + \gamma \; max_{a'} Q(s'_{j+\tau}, a', h_{j+\tau}|\theta^-)$
            $\mathcal{L}_j = \big(y_{j+\tau} - Q(s_{j+\tau}, a_{j+\tau}, h_{j+\tau-1}|\theta_i)\big)^2$
        end for
        $\theta_{i+1} = \theta_i + \frac{\alpha}{|B|} \sum_{j=1}^{\text{batch size}} \nabla_{\theta_i} \mathcal{L}_j$
        $i = i + 1$
    end while
end while

4.6 Evaluation

Each model is evaluated during the training period to approximate the learning curve. Evaluation is done by running 15 SUMO simulations using a purely greedy policy. Each figure plots the performance of the networks in terms of average rewards and average travel time per vehicle each with their standard error.

5 History Size

This section presents the experiments performed in order to answer research question Q_1. and Q_2.

5.1 Motivation

An important question when using recurrent neural networks is how much the backpropagation through time algorithm will be truncated. It is easy to claim that if the agent remembers more time steps it will perform better as well. If that is the case, then setting the horizon for the BPTT algorithm as high as computationally possible would always give the best results. However, it is not so obvious that this is really the case. Remembering more time steps means that the agent is exposed to more data and that could possibly confuse the agent instead of giving it more information. Additionally, it could be that what happened many time steps ago is not even relevant for the current decision. In that case computational resources are wasted on time steps the agent does not even need.

5.2 Experimental Setup

The performance of DRQN will be compared to its non-recurrent counterpart. The DQN model takes 2 stacked input matrices as input [36] and will be compared to DRQN with the same size in history ($\tau = 2$), a history size as long as the yellow time ($\tau = 4$), the same history size as in [7] ($\tau = 10$) and an even bigger history size ($\tau = 20$). All models are trained for 2,000,000 iterations using the hyper parameter values from Table 1.

5.3 Results

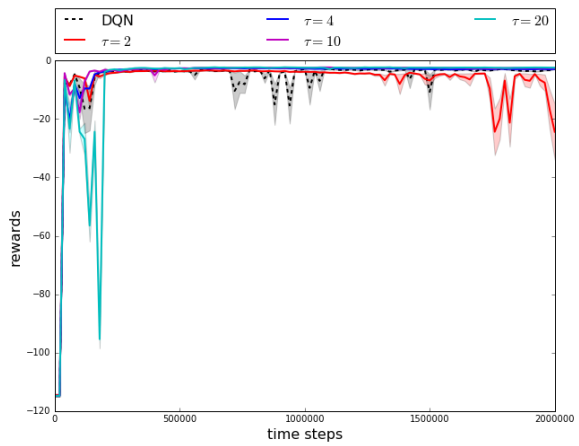
In Figure 4 the results are presented. The DRQN model with $\tau = 2$ seems to be able to find a policy that performs just as well as the DQN agent, but becomes unstable as the training continues. When using a history size of 4, DRQN performs just as well as the DQN model and seems more stable in preserving the found policy. Still, the performance decreases a bit at the end of training. With $\tau = 10$, DRQN outperforms the DQN model and preserves its performance better than with a history size of 4. Finally, DRQN with $\tau = 20$ seems to perform a little bit better than the model with a history size of 10, but it has a big drop in performance at the beginning of training. It is clear from the plots that all models experience fluctuation in their performance before finding a good policy, but since a longer time dependency is harder to train it may explain why the model with the largest history size struggles the most at the beginning.

Table 2 shows the training times of the different models. Between the recurrent models the training time seems to increase linearly with τ. If the sequences are longer, the number of computations per update increases by just as much. The difference between DQN and DRQN with the same τ is caused by the fact that DQN with two stacked images as input has more parameters than DRQN. When an input image has more channels (an RGB image has 3 channels) the filters that convolve that images have the same number of channels and thus more weights to be updated.

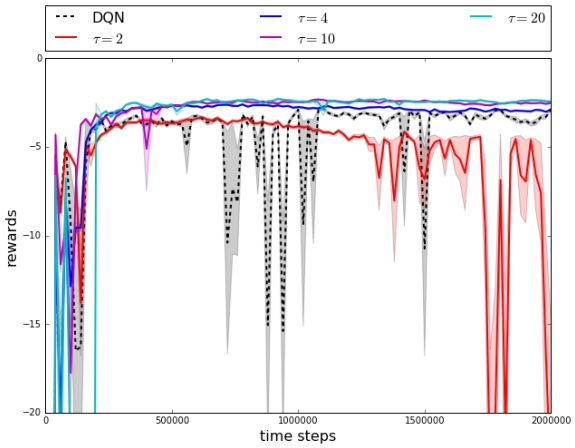
The best performance found in [35] was a DQN model trained with an input with twice the resolution as the input used to train the models in these experiments. For the sake of comparison the best performing DRQN model, DRQN with $\tau = 10$, was compared to DQN that uses two stacked position matrices with light information of the size 168x168 as input (see Figure 5). Even when using a single frame that is half the resolution, DRQN is able to perform just as well as DQN. For future experiments the history size is set to 10.

Model	τ	**Train time (minutes)**
DQN	2	1206.73
DRQN	2	939.41
DRQN	4	1328.60
DRQN	10	2480.07
DRQN	20	4417.98

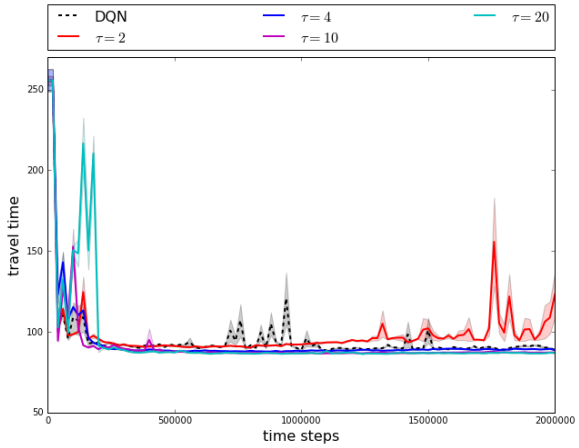
Table 2: The time in minutes it took for the models to train for 2,000,000 iterations. For the DQN agent τ represents the number of stacked frames that is given as input.



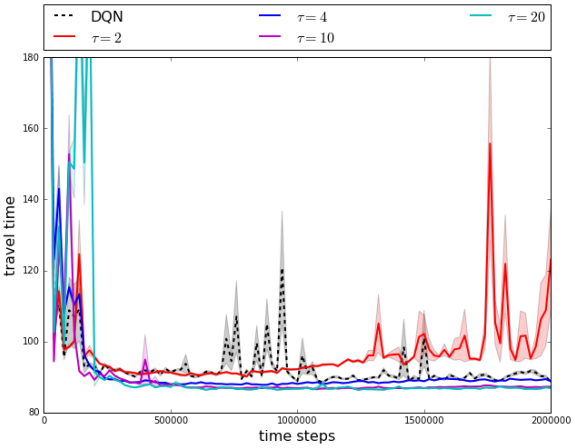
(a) Average reward and standard error.



(b) Average reward between 0 and -20.

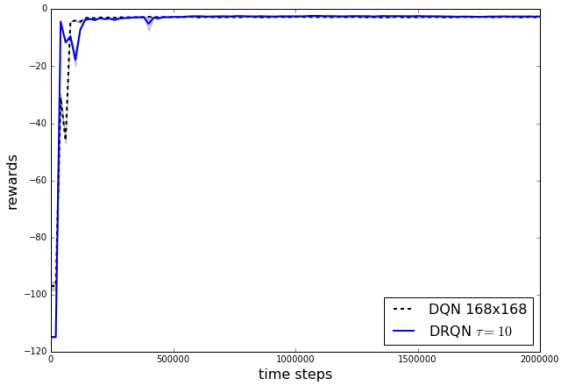


(c) Average travel time and standard error.

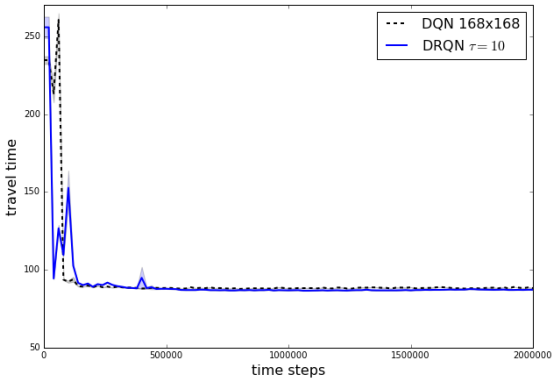


(d) Average travel time between 80 and 180.

Figure 4: The effect of different history sizes on the average reward and travel time of the greedy policy of DRQN compared to the DQN baseline.



(a) Average reward and standard error.



(b) Average travel time and standard error.

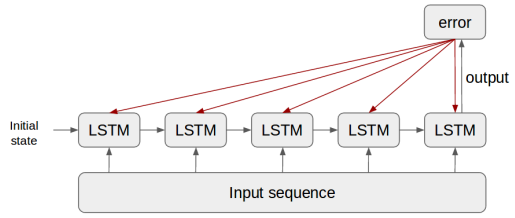
Figure 5: The average reward and travel time of DRQN with $\tau = 10$ compared to DQN that uses input matrices with a size of 168x168.

6 Update Methods

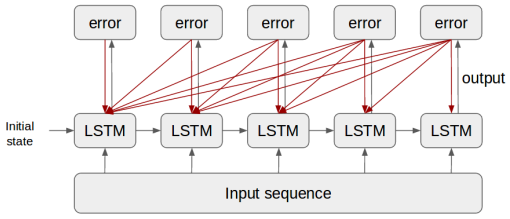
This section explores the effect of using different parts of a sequence to calculate the loss.

6.1 Motivation

A recurrent neural network is trained using sequences of data. But what is the error signal of a sequence? In Algorithm 2 the loss function is described as the mean squared TD-error of the last time step (Figure 6(a)). This type of update can be used to train a recurrent model to summarize a sequence in the hidden state [6]. It is, however, also possible to calculate the squared TD-error at *every* time step (Figure 6(b)). The loss of the entire sequence is therefore the sum of the errors over all the time steps. By calculating the loss at every time step, the model learns to map a sequence of inputs to a sequence of outputs [6].



(a) The time steps leading up to the output serve to encode the hidden state with the information needed for the final output.



(b) Every time step in de sequence produces an output and therefore an error signal that is used for backpropagation.

Figure 6: Visualization of two different types of error propagation.

As described in Section 3, the internal state of the LSTM is initialized to be zero at the start of each mini-batch of sequences. This means that the internal state at the beginning of the sequence contains less information, possibly insufficient to make a good prediction. The updates resulting from these predictions could be flawed as well, interfering with the learning process. As a solution to this issue, some time steps in the sequence are only used to update the internal state of the recurrent network [15]. The time steps that are not used to calculate the errors are called the "burn-in period". The update method visualized in Figure 6(a) could also be described as using a burn-in period of $\tau - 1$ where τ is the length of the sequence.

6.2 Experimental Setup

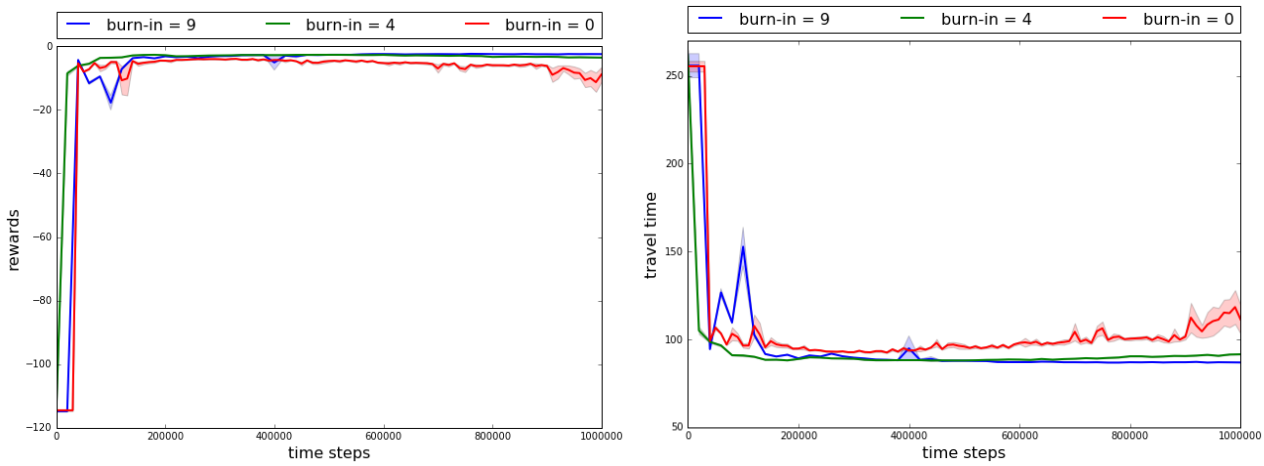
Two DRQN methods are trained: one using the sum of the squared TD-errors of every time step as the error signal of a sequence (burn-in = 0) and the other using only part of the sequence to sum the errors over. For both models $\tau = 10$ and the burn-in period is set to 4 because the results of DRQN with $\tau = 4$ were good enough to assume that after those time steps the hidden state contains enough information to make good predictions. The results of these models are compared to DRQN with $\tau = 10$ that only uses the error of the final time step (burn-in = 9). All models are trained for 1,000,000 time steps with the hyper parameter values of Table 1.

As an extra experiment, different learning rates were tested on the full sequence update method. A summation of errors means that there is also a summation of gradients, thus the updates of the weights have become bigger and might cause the model to overshoot. Adjusting the learning rate to a smaller value could solve this problem.

6.3 Results

The results of this experiment are shown in Figure 7. The update method that uses the errors at every time step performs the worst. Even though the burn-in period of 4 time steps is more stable at the beginning of training, eventually the performance starts to decline as it does with a burn-in period of 0 steps. This behaviour shown by the DRQN models with a smaller τ value. This could mean that using a burn-in period of 9 is simply the best setting for the traffic control problem.

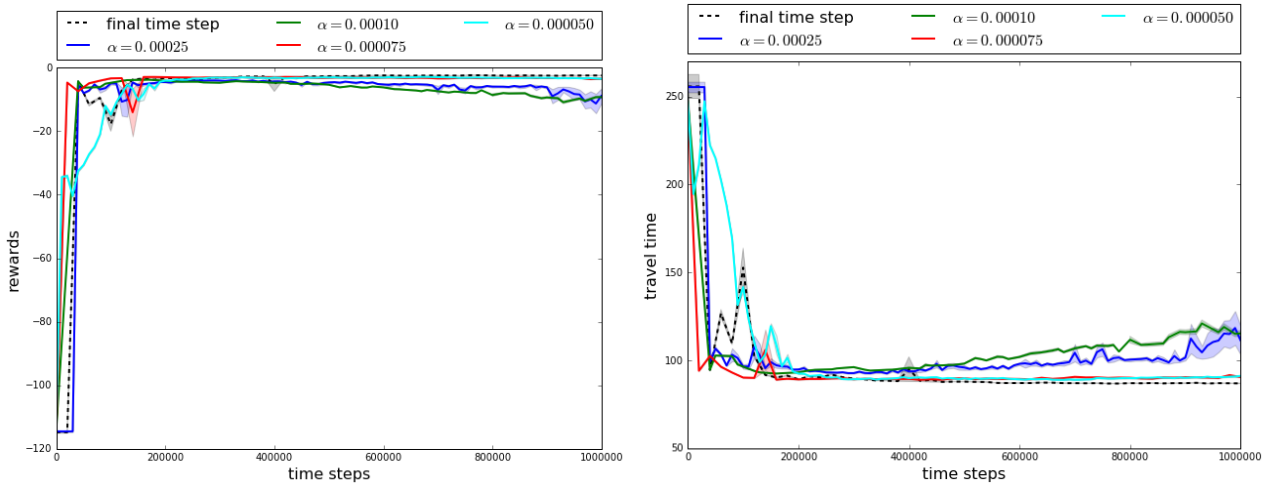
Using a smaller learning rate when using more error signals from one sequence helps to dampen the decline in performance that happens when using a learning rate of 0.00025 (see Figure 8). However, it also shows that using the squared TD-error from more time steps in a sequence does not result in a better policy than using the squared TD-error of the final time step.



(a) Average reward and standard error.

(b) Average travel time and standard error.

Figure 7: The average reward and travel time of the different number of time steps that are used to update DRQN.



(a) Average reward and standard error.

(b) Average travel time and standard error.

Figure 8: The average reward and travel time of different learning rates when using the entire sequence to calculate the loss. The dotted line is the DRQN model trained by only using the squared TD-error of the final time step.

7 Freeze Interval

This section explores the behaviour of the trained models in order to better understand DRQN.

7.1 Motivation

The DRQN model is trained to predict the optimal Q-value for a given observation. The Q-value is the discounted return starting from the given observation and action (see Equation 14). If the mean reward per time step ($\bar{r}$) is known, the expected return can be calculated like this:

$$Q(o_t, a_t, h_{t-1}|\theta) = \sum_{k=0}^{\infty} \gamma^k \bar{r} = \frac{\bar{r}}{1-\gamma} \tag{20}$$

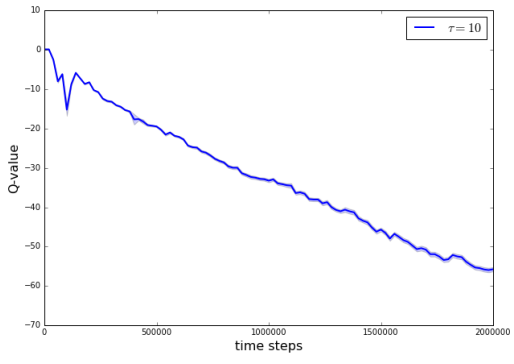
The mean reward of an episode *is* know as well as the discount factor, the expected Q-value can therefore be calculated. In the case of the DRQN model with $\tau = 10$ at the end of training the model received an average reward of $r \approx -2.53$. Given that $\gamma = 0.99$, the mean value for the best action should be -252.85. According to Figure 9(a) however, the average predicted Q-value of the best action over an episode is -55.77. If the predicted values are this inaccurate, then the TD-error must be big. However, the TD-errors shown in Figure 9(b) are < 10. This suggests that the target value used during training is incorrect as well. Something is restricting the model in learning the correct Q-value for the actions. After initialization, the output for the target network is approximately zero for every state. This means that the loss function looks like this:

$$\begin{aligned} \mathcal{L}_t &= \big(Q(o_t, a_t, h_{t-1}|\theta) - y_t\big)^2 \\ y_t &= r_t + \gamma[max_{a'} Q^-(o_{t+1}, a', h_t|\theta^-)] \\ &= r_t + \gamma 0 \end{aligned} \tag{21}$$

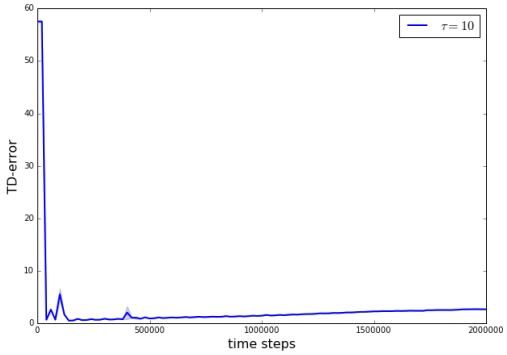
Until the parameters of the target network are updated (see line 18 of Algorithm 2), the action network is trained to predict the immediate reward (e.g. $Q(o_t, a_t|\theta) \rightarrow r_t$). Once the target network parameters are updated a new target value is defined:

$$\begin{aligned} \mathcal{L}_t &= \big(Q(o_t, a_t, h_{t-1}|\theta) - y_t\big)^2 \\ y_t &= r_t + \gamma[max_{a'} Q^-(o_{t+1}, a', h_t|\theta^-)] \\ &= r_t + \gamma r_{t+1} \end{aligned} \tag{22}$$

The new target of the action network becomes the immediate reward plus the discounted, immediate reward it predicts for the next state according to its old parameters (e.g. $Q(o_t, a_t, h_{t-1}|\theta) \rightarrow r_t + \gamma r_{t+1}$). It seems that the number of times the parameters are synced influences the number of steps the model learns to look ahead. When using the hyper parameters values shown in Table 1, the model should learn to look approximately 67 time steps ahead. A smaller freeze interval would suggest that the model learns a more accurate Q-value.



(a) The average Q-values and standard error of the chosen actions following the greedy policy.



(b) The average TD-error and standard error.

Figure 9: Average Q-values and TD-errors of a DRQN with $\tau = 10$.

7.2 Experimental Setup

In this experiment, all models are DRQNs with $\tau = 10$ and trained as described in Section 4.5. The only variable that is changed is the freeze interval that is set to 1, 10, 100, 1000, 10000 and 30000 (value from Table 1). The models are each trained for 1,000,000 steps.

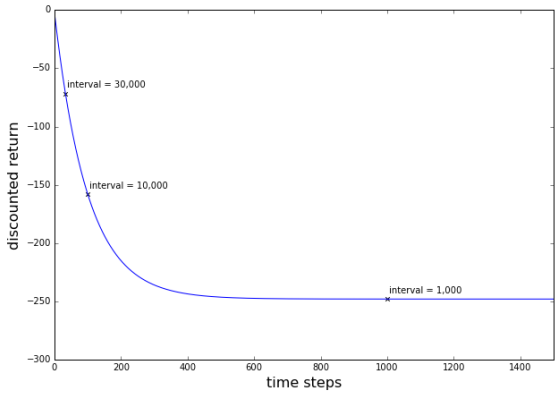


Figure 10: The plot of $\sum_{t=0}^{T} 0.99^t \times -2.53$ with T being the number of time steps the discounted return is calculated for (x-axis).

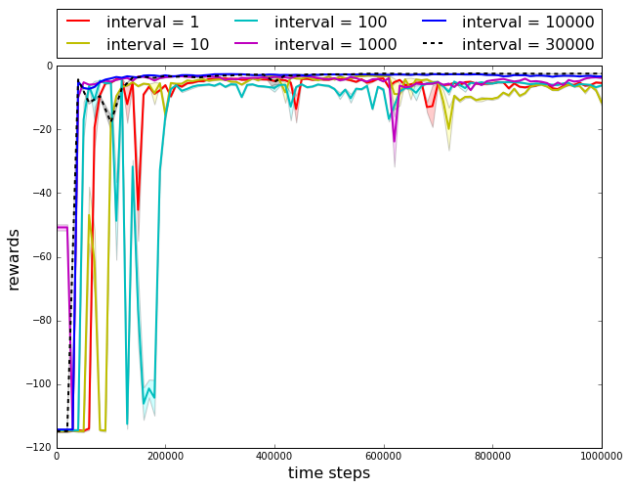
7.3 Results

Figure 11 shows the performance of DRQN trained using different intervals to synchronize the target and the action network. None of the models seem to predict the correct Q-value at the end of training and they all perform worse that the model that uses an interval of 30,000 steps. A smaller freeze interval destabilizes the learning curve of the model and prevents the network from learning a proper policy. However, the worst performing model is not the one using the smallest freeze interval. Compared to the intervals of 10 and 100, the 1 time step interval seems to be more stable even though it is unable to find a good policy. A small freeze interval comes at the risk of not giving the model enough time to minimize the loss function, giving each prediction a high inaccuracy. This could be solved by increasing the learning rate, enabling the model to apply bigger updates on the parameters.

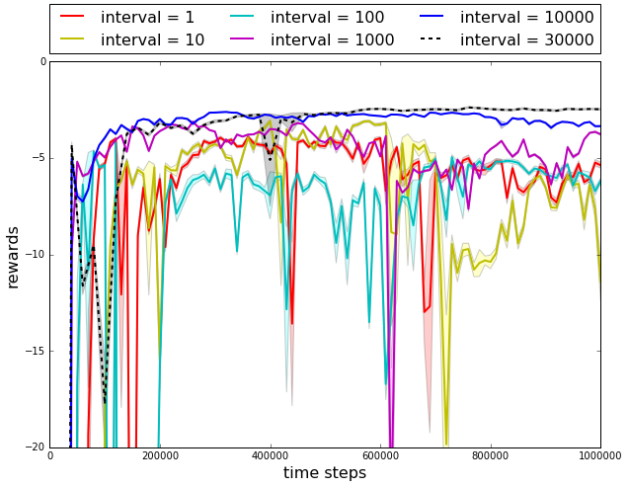
The mean reward, predicted Q-value, expected Q-value and squared TD-error of the 1,000,000th iteration are shown in Table 3. Also shown in this table is the RMSE between the predicted Q-value and the expected Q-value calculated using Equation 20. The RMSE could be interpreted as the expected squared TD-error, assuming that the expected Q-value is the optimal value. Figure 10 shows the discounted return as a function of the number of time steps it is calculated for. Also marked in this plot are the number of time steps some freeze intervals reach to predict after 1,000,000 training steps. The image shows that both an interval of 30,000 and 10,000 prevent the model from learning the expected Q-value because the target network parameters have not been updated often enough. All the intervals smaller than 10,000 should have been able to learn the correct Q-values. In order to learn the optimal Q-value as described above, the model must minimize the squared TD-error within the number of time steps the freeze interval provides. Looking at the squared TD-errors in Table 3, only with a freeze interval of 30,000 the loss was able to do that. The models using a smaller interval have not been able to minimize the TD-error and therefore their predictions are inaccurate which might be an explanation for their poor performance. Because of these inaccurate predictions, the target values also become inaccurate. This explains why the squared TD-error of these models is so much bigger than the RMSE: the target value is not predicting the expected return. From the models whose interval allows to learn the correct Q-value, the one with the interval of 1,000 seems to have a squared TD-error similar to the RMSE in Table 3. Nevertheless, its performance is worse than the model with an interval of 30,000 and the learning curve is very unstable.

Freeze interval	**Mean reward**	**Predicted Q-value**	**Expected Q-value**	**RMSE**	**TD-error^2**
1	-5.38	-236.79	-538	304.80	532.71
10	-11.42	-355.38	-1142	830.04	3213.95
100	-6.19	-201.82	-619	418.61	2519.96
1,000	-3.78	-123.63	-378	254.67	179.93
10,000	-3.35	-77.11	-335	258.49	30.54
30,000	-2.48	-33.23	-248	215.30	1.44

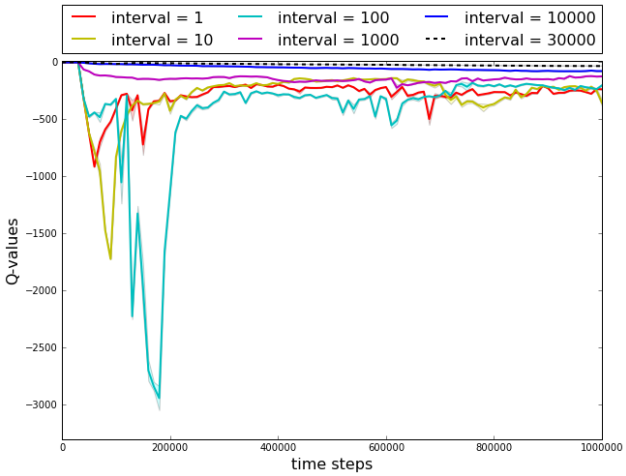
Table 3: The model performance at the end of training. The RMSE column stands for Root Mean Squared Error between the predicted Q-value and the expected Q-value.



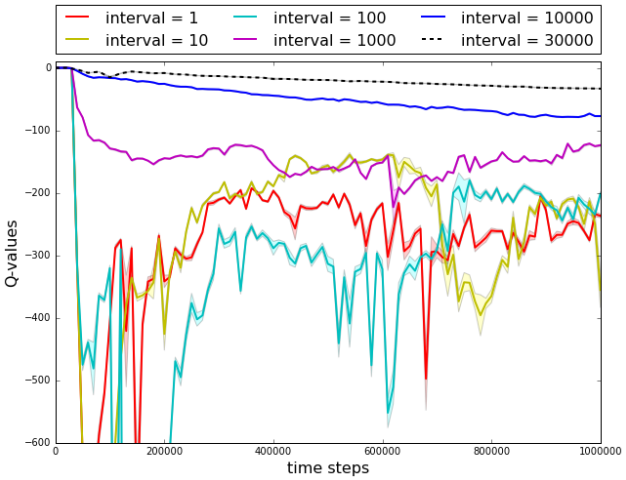
(a) Average reward and standard error.



(b) Average reward between 0 and -20.



(c) Average Q-value and standard error.



(d) Average Q-value between 0 and -600.

Figure 11: The average rewards and Q-values of DRQN models trained with different freeze intervals.

8 Action Motivation DRQN

In this section, a new architecture for deep recurrent Q-learning is presented. It gives the model information about the action taken in the previous step but in a new way that, to our knowledge, has not been done before. This new model is applied to the traffic control problem in order to evaluate its performance.

8.1 Motivation

In a POMDP the agent is given an observation of the environment which only contains a part of the total information there is. Using a recurrent neural network, DRQN learns to make decisions based on a sequence of observations which should reveal more about the total information the environment has to offer. Something else that could give the agent more information is remembering what actions have been taken so far. Remembering an action that has been taken a while ago (e.g. whether or not the agent has picked up a key to open a door) or deducing the effect of an action over multiple time steps can only be done if the agent receives the taken action as input as well. Foerster et al. [4] extended DRQN with this information in a multi-agent setting. Instead of appending the action to the input directly, [41] introduced Action DRQN (ADRQN) in which the action is embedded into a high-dimensional vector and then appended to the input.

When using experience replay for training, the model is presented a sequence of inputs $\{x_0, ..., x_\tau\}$ with $x_t = [a_{t-1}, o_t]$. It is important to note that during training the actions represented in this sequence are selected by a different policy than the one that is used at the time of the update. It is entirely possible that the action that the current policy proposes given x_t is different than the action that is given in x_{t+1}. Even though the replay memory provides the model with the "correct" action, it does not provide any feedback as to *why* that action was taken or *how different* the current policy is with its action selection. This "action mismatch" does, however, propagate through the entire sequence via the hidden state of the recurrent neural network. In this section a new architecture is proposed to give the recurrent neural network information about the *current* policy at every time step. Action Motivation DRQN (AMDRQN) appends the probability of the chosen action to the input. Not only will the model receive information about what action was taken but it will know how certain or how much better that action was compared to the other actions.

8.1.1 Architecture

Either from the environment or from the replay memory, the model is presented with an input o_t and a one-hot encoded vector representing the action that was taken in the previous time step a_{t-1} (see Figure 12). The observation is processed as usual while a_{t-1} is multiplied with the policy calculated from the Q-values of the previous time step which are passed along just as the hidden state of the LSTM is. This results in a one-hot encoded vector that contains the probability of choosing the action a_{t-1} under the current policy. This vector

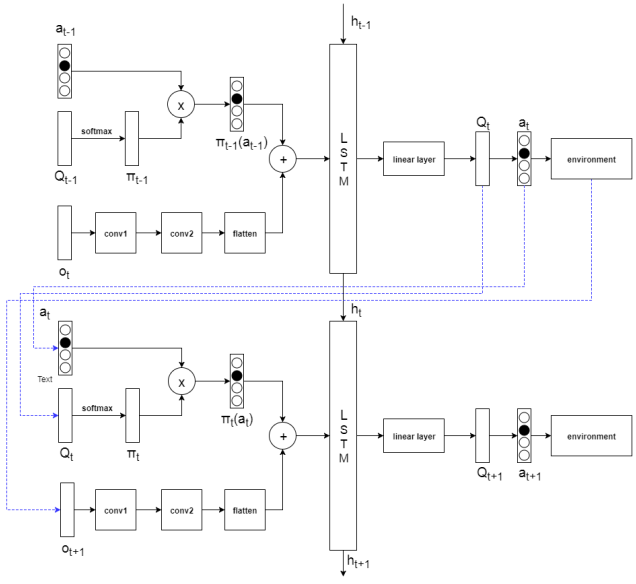


Figure 12: The architecture of the Contextual Action DRQN is the same as the architecture described in Section 4.4 except one thing: before the LSTM, the output of the flatten layer and the one-hot encoded action probability are concatenated.

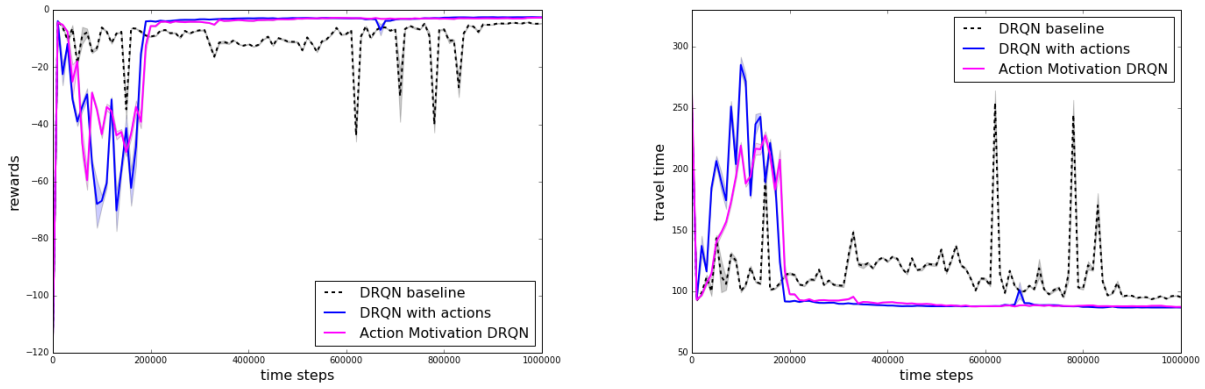
is concatenated to the flattened output of the convolution layers and passed through the LSTM. The Q-values are then calculated by means of a fully connected layer and used to select a new action. The action selection mechanism and the environment are beyond the model's domain, thus o_t and a_{t-1} are treated as input. On the contrary, the Q-values are passed along over time and are treated the same way as the internal state of the LSTM. This creates a new path of gradients that update the parameters through the policy and the Q-values at every time step. The error that is propagated backwards through the network is the squared TD-error.

8.2 Experimental Setup

Three models are trained in SUMO: DRQN, DRQN with one-hot encoded actions appended to the input and AMDRQN. The hyper parameters are set to the values in Table 1 with $\tau = 10$ but instead of using the binary position matrix with traffic light information only the position matrix was given as input. All models are trained for 1,000,000 steps.

8.3 Results

The information about the current configuration of the traffic lights is removed from the input but can be deduced from the action input. The results in Figure 13 show that without the information about the state of the traffic lights, DRQN struggles to find and maintain a good policy. Both DRQN with action information and AMDRQN are able to find a policy that performs just as well as DRQN that uses the position matrix with light information as input. This means that the models can deduce as much information from their past actions as from the state of the traffic lights. The information about the probability of an action does not provide the model with enough extra information to outperform its one-hot encoded counterpart. However, the fluctuations at the beginning of training seem to be a bit smaller in magnitude for AMDRQN and the found policy seems more stable in the long run. Even though the differences are small, this could mean that the new gradient path dampens the some fluctuation. This oscillation at the beginning of training seems to be caused by the action information as it does not occur in the DRQN baseline model, nor in the DRQN models trained *with* light information as input. In this scenario of traffic light control, the number of actions is restricted to two. If the instability is indeed connected to the action input, more actions could lead to more fluctuation. In those scenarios AMDRQN could provide stability and maybe even speed up learning.



(a) Average reward and standard error.

(b) Average travel time and standard error.

Figure 13: The average reward and travel time of DRQN, DRQN with action information and Action Motivation DRQN.

9 Value and Policy Search

Using RNNs to solve a POMDP is not as straight forward as it may seem. This is shown to be true in practice in the previous sections. This section explores the theoretical implications of using recurrent neural networks to solve POMDPs.

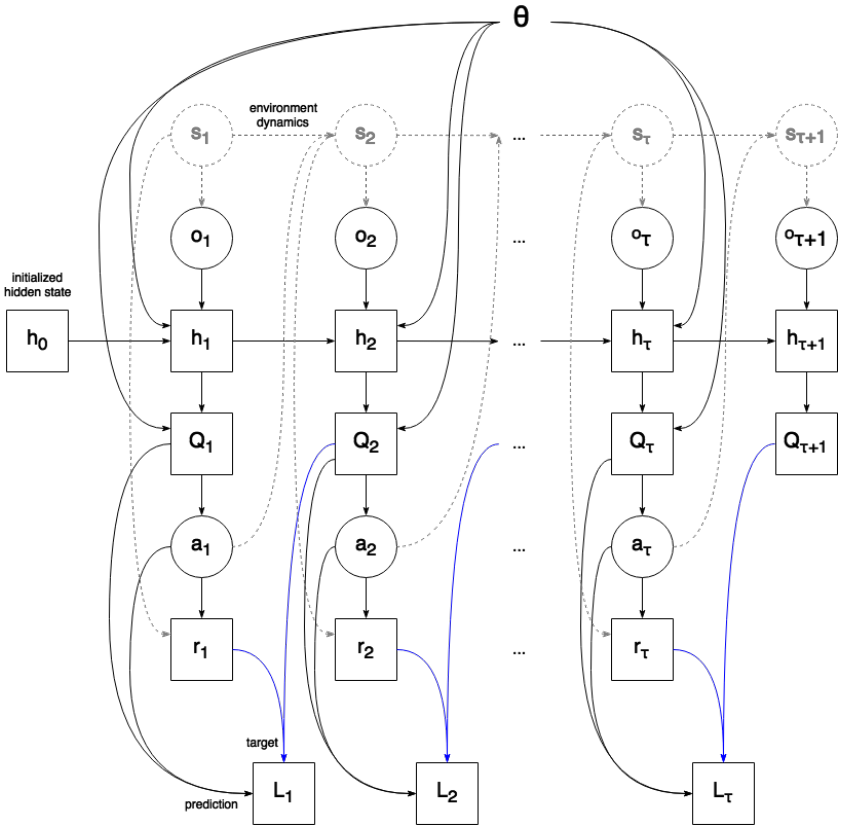


Figure 14: The computation graph of a Deep Recurrent Q-network unrolled over τ time steps.

9.1 Motivation

Using sequences instead of single observations fundamentally changes the computation graph of a process. Figure 14 describes a DRQN process unrolled over τ time steps. This graph was constructed using a formalism introduced by Schulman et al. [30]: round nodes represent stochastic variables and square nodes deterministic ones. When arrows or nodes have dotted lines it means that these relations and variables are unknown. In words, it represents a recurrent neural network that uses an observation o_t, the previous internal state h_{t-1} and parameters θ to compute the current internal state of the network h_t. Using other parameters in θ, h_t is transformed into Q_t, a vector containing the Q-value for each action in $\mathcal{A}$. These values are used to select the action a_t, thus the probability of choosing an action depends in the Q-values $\pi(a_t|Q_t)$. The reward is calculated using the unknown state and action $\mathcal{R}(s_t, a_t)$. All the calculations that are performed by the recurrent neural network are deterministic functions of their parent nodes. The states and observations are controlled by unknown environment dynamics and are therefore classified as stochastic variables. Even though the optimal policy is a deterministic function of the optimal Q-value, during training this is certainly not the case and therefore the policy in Figure 14 is stochastic.

When train on single transitions, the loss node is an expectation over (s, a, r, s') of the squared TD-error (see Equation 16). However, the graph in Figure 14 shows that the loss at any time step depends on all prior stochastic variables in the given sequence. $H_\tau = (s_1, o_1, a_1, ..., s_\tau, o_\tau, a_\tau, s_{\tau+1}, o_{\tau+1})$ is the sequence of stochastic variables up to time τ including the state and observation of step $\tau + 1$ which are needed to calculate the TD-error of the final time step.

$$
\begin{aligned}
L_\tau(H_\tau, \theta) &= \left(Q_\theta(h_\tau, a_\tau) - \left[r_\tau + \gamma \max_{a'} Q_\theta(h_{\tau+1}, a')\right]\right)^2 \\
\mathcal{L}_\tau(\theta) &= \mathbb{E}_{H_\tau \sim p_\theta}\left[L_\tau(H_\tau, \theta)\right]
\end{aligned}
\tag{23}
$$

The probability of H_τ is a combination of environment dynamics and the policy. As explained in Section 2.2.1 the probability distribution over the next states is given by the transition function $\mathcal{T}(s,a)$ and Section 2.2.5 introduced that the probability of seeing a certain observation depends on the underlying state $\mathcal{O}(s)$. The probability distribution p_θ is written using only stochastic terms as it describes the probability of a sequence of stochastic variables.

$$p_\theta = p(s_1)\mathcal{O}(o_1|s_1)\prod_{t=1}^{\tau}\pi_\theta(a_t|o_1...o_t)\mathcal{T}(s_{t+1}|s_t,a_t)\mathcal{O}(o_{t+1}|s_{t+1}) \tag{24}$$

In order to use backpropagation to train the DRQN, the gradient of the expected loss must be calculated. Schulman et al. [30] describe how to calculate an unbiased gradient estimation from a *stochastic computation graph* using the *score function* estimation [5] and the *path wise derivative*.

$$\frac{\partial}{\partial\theta}\mathcal{L}_\tau(\theta) = \frac{\partial}{\partial\theta}\mathbb{E}_{H_\tau\sim p_\theta}\left[L_\tau(H_\tau,\theta)\right] = \mathbb{E}_{H_\tau\sim p_\theta}\left[L_\tau(H_\tau,\theta)\left(\frac{\partial}{\partial\theta}\log p_\theta\right) + \frac{\partial}{\partial\theta}L_\tau(H_\tau,\theta)\right] \tag{25}$$

For the full derivation, see Appendix A. The partial derivative of the probability distribution can be simplified further: the only distribution in Equation 24 that is affected by the parameters is the policy. This removes the complication of having to propagate the error through the unknown environment functions.

$$\begin{aligned}\frac{\partial}{\partial\theta}\log p_\theta &= \frac{\partial}{\partial\theta}\left[\log\left(p(s_1)\mathcal{O}(o_1|s_1)\right) + \sum_{t=1}^{\tau}\log\pi_\theta(a_t|o_1...o_t) + \log\left(\mathcal{T}(s_{t+1}|s_t,a_t)\mathcal{O}(o_{t+1}|s_{t+1})\right)\right] \\ &= \sum_{t=1}^{\tau}\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|o_1...o_t)\end{aligned} \tag{26}$$

The resulting gradient is very similar to the policy-gradient method REINFORCE [39]. In order to compute these gradients the policy should be a non-zero, smooth function of the parameters. The ϵ-greedy policy that is used in Q-learning does not satisfy these requirements, instead the Boltzmann distribution [1] will be considered. The probability of action $a \in \mathcal{A}$ given the Q-values Q_t can be calculated using the following equation.

$$\pi_\theta(a|Q_t) = \frac{\exp\left(Q_\theta(h_t,a)/T\right)}{\sum_{a'\in\mathcal{A}}\exp\left(Q_\theta(h_t,a')/T\right)} \tag{27}$$

The lower the value of T gets, the more influence the Q-values have on the distribution. With a very low temperature ($T \to 0^+$) the probability of the highest Q-value will be approximately 1. To make use of automatic differentiation procedures, Equation 25 is translated into a *surrogate loss function* $\mathcal{L}(H_\tau)$ [30].

$$\begin{aligned}\frac{\partial}{\partial\theta}\mathcal{L}_\tau(\theta) &= \mathbb{E}_{H_\tau\sim p_\theta}\left[L_\tau(H_\tau,\theta)\left(\frac{\partial}{\partial\theta}\log p_\theta\right) + \frac{\partial}{\partial\theta}L_\tau(H_\tau,\theta)\right] \\ &= \mathbb{E}_{H_\tau\sim p_\theta}\left[L_\tau(H_\tau,\theta)\sum_{t=1}^{\tau}\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|o_1...o_t) + \frac{\partial}{\partial\theta}L_\tau(H_\tau,\theta)\right] \\ &= \mathbb{E}_{H_\tau\sim p_\theta}\left[\frac{\partial}{\partial\theta}\mathcal{L}(H_\tau,\theta)\right] \\ \mathcal{L}(H_\tau,\theta) &= \hat{L}_\tau(H_\tau,\theta)\sum_{t=1}^{\tau}\log\pi_\theta(a_t|o_1...o_t) + L_\tau(H_\tau,\theta)\end{aligned} \tag{28}$$

Where $\hat{L}_t(H_\tau,\theta)$ is the sampled value of L_t and is treated as a constant [30]. This is to ensure that no gradients are derived from the squared TD-errors that are multiplied with the log-likelihood of the sequence actions. DRQN is trained using gradient descent methods, meaning that this surrogate loss function will be minimized. The aim of deep Q-learning is to minimize the squared TD-error and the new loss function can be interpreted to do that in two different ways: (1) by minimizing the term $L_\tau(H_\tau,\theta)$ and (2) the gradients of the log-probability point towards the minimization of the likelihood of the given actions. These gradients are multiplied with $\hat{L}_\tau(H_\tau,\theta)$ which means that the probabilities of sequences with a high squared TD-error are minimized more than those with a small TD-error.

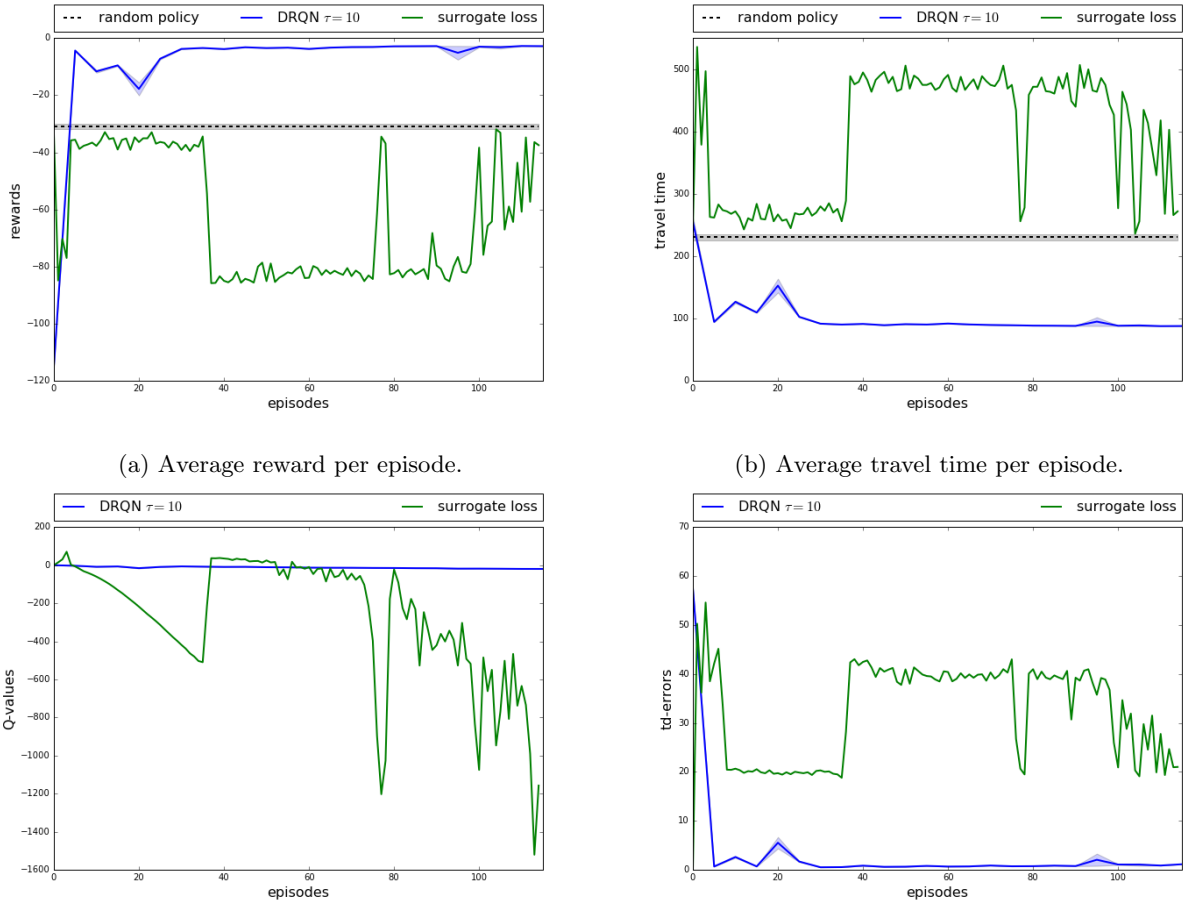
Independently, this can be seen as a new instance of the *Value and Policy Search* (VAPS) algorithm introduced by Leemon Baird in 1999 [1]. The gradient described in Equation 25 can be seen as the summation of two gradients: the gradient of the log-likelihood of the sequence and the gradient of the squared TD-error. The partial derivative of L_τ is the gradient that is used when doing Q-learning, *value-based search*. The partial derivative of the log-probability of the policy is the *policy-based search*.

9.2 Experimental Setup

A Deep Recurrent Q-Network was trained using the loss function described in Equation 28 and the same hyper parameter values as stated in Table 1 with $\tau = 10$. The gradients of the log-probabilities are clipped to [-2000,2000] because they could become infinitely big as the probabilities become smaller. This model was trained for 115 episodes and the results are from using an ϵ-greedy policy with $\epsilon = 0.1$.

9.3 Results

The performance of DRQN during training, using the surrogate loss function is shown in Figure 15. The best the model seems to reach is a performance that is almost as good as a random policy. Learning is very unstable but this does not necessarily mean that the loss function is flawed. As stated before, the partial derivative of a log-likelihood can become infinitely big if the probabilities are very small. These gradients are also multiplied with the squared TD-error which can be very large as well. Even if the gradients are clipped, the values can still be very large but are not compensated by a smaller learning rate. That being said, up until episode 40 the model seems fairly stable. The performance and TD-error are not improving but they are not getting worse either. In the same period, the Q-value goes down faster than when using the squared TD-error as loss function and very smoothly as well. Then something happens that increases the Q-value, and thus the TD-error. The performance of the model degrades as well which is to be expected if the Q-values are even more inaccurate than before. After a while the performance seems to slowly increase again, but it is hardly stable and it only reaches the performance it had before which is worse than randomly choosing actions. The performance of the random policy almost looks like a threshold the model is unable to cross. These results indicate that the implementation of the surrogate loss function described in Equation 28 needs more work before it can be used. Nevertheless, if this loss function can be stabilized and the behaviour we see at the beginning of Figure 15 is preserved, it could increase the speed at which the model learns the optimal Q-function.



(a) Average reward per episode.

(b) Average travel time per episode.

(c) Average value of the chosen action per episode.

(d) Average TD-error per episode.

Figure 15: The average reward and travel time of DRQN trained with the surrogate loss function described in Equation 28. The dotted like is the result of a policy that only chooses random actions.

10 Related Work

This section will discuss earlier research that has applied recurrent neural networks to Q-learning in POMDPs.

10.1 Deep Recurrent Reinforcement Learning

In previous work recurrent neural networks have been applied to multiple POMDP in order to handle the partial observable data. In most research, DRQN is simply used as a tool to solve a newly defined problem [41, 4, 21]. It is rarely the main focus of research but sometimes researchers report findings on their experiences with this model. Hausknecht et al. [7] introduced the DRQN model in the form that is now mostly used. They explored different positions of the LSTM layer in a deep network architecture and different sampling methods for the replay memory: sample random episodes and walk through them sequentially or sample random sequences from random episodes. But the choice of the number of unrolled time steps (called τ in this thesis) was not explained and neither was the choice of computing the squared TD-error at every time step of a sequence. They acknowledge that initializing the hidden state at the beginning of a randomly sampled sequence makes it harder for the model to learn dependencies that go beyond the length of the sequence. Nevertheless, Hausknecht et al. report no significant difference between sequential and randomly sampled updates but the results of the referred experiments are not shown either.

Other work used a different update method than presented in [7]. Lample et al. [15] trained a DRQN agent to play the game DOOM. Instead of using every observation in a sequence to update the network, they used the first n time steps to simply update the hidden state. Lample et al. claimed that because the first number of predictions are based on very little data, they would be inaccurate and harmful for the learning process. They do not, however, experiment with the value of n as was done in this paper.

10.2 DRQN with Actions

The idea of using the previous action as input for a deep recurrent Q-network is not new. Foerster et al. [4] introduced the *Deep Distributed Recurrent Q-Network* (DDRQN) in order to learn in a multi-agent environment. Their results indicate that the action information is important information when learning solutions to riddles, however, it was only tested in a multi agent setting. Inspired by this model, Zhu et al. [41] introduces *Action DRQN* (ADRQN). Their aim was to incorporate the action information in DRQN but simply appending a one-hot encoded vector of the previous action to the high-dimensional observation vector caused numerical instabilities. This was not observed in the experiments in Section 8 even though there are only two actions available for the agent and the size of the observation after the convolution layers is 2592. In order to solve their numerical instabilities Zhu et al. created an embedding layer for the actions so that they can be represented as high-dimensional vectors. They proved that adding action information as input to DRQN can increase the performance of the model. The architecture proposed in this thesis not only provides extra information to the DRQN (the probability of the chosen action) but also gives the model an extra recurrent connection that updates the weights through the policy.

11 Discussion

When aiming to solve a POMDP with deep learning a lot of research has made use of recurrent neural networks. Because of the success of using these networks in the reinforcement learning domain [7][15][21] future work will probably make more use of these models. In this thesis it was shown that even with a single, low resolution position matrix as input DRQN was able to learn a policy that performs just as well as a DQN model with two, higher resolution inputs. However, applying a recurrent DQN to a problem may not be as straightforward as it seems. The results in this thesis indicate that if the agent can remember more time steps, the performance will increase as well. Training the recurrent network on more time steps not only increases performance but the computational costs as well. A balance should be found between performance increase and its cost, but that depends on the available hardware.

When using sequences of data points it needs to be decided what to do with the error that is calculated at every time step. Summing all the errors in a sequence proved to be worse than only using the error of the last time step even after adjusting the learning rate to accommodate the increase in gradients. Using a certain number of time steps to initialize the hidden state of the recurrent network was an improvement but eventually the performance would decline. While keeping τ fixed and changing the size of the burn-in period, the number of time steps that are used for the update changes as well. Results in [15] suggest that the number of time steps used for the updates influence the learning performance as well. The results in this thesis point to a similar effect although it is difficult to say for sure because of the changing burn-in period. Nevertheless, the results

from using only the error of the final time step indicate that this is the best method to train DRQN.

Choosing the right freeze interval is not a triviality when learning the optimal Q-values. The freeze interval partitions the learning process into a number of regression problems. The number of partitions should allow the model to learn the discounted return up until the horizon that is set by the discount factor. This means that when using a discount rate $\gamma = 0.99$ the number of regression problems should be approximately 600 at least (see Figure 10). The freeze interval should be chosen to accommodate this number given the total number of iterations the model is to be trained. However, the interval should also give the model enough time to learn each of the defined regression problem otherwise the Q-values will be inaccurate because it was not able to minimize the TD-error.

In this thesis, a new architecture was presented: Action Motivation DRQN. This model appends the probability of the previously chosen action to the observation input. This enables an agent to make decisions based on sequences of observation-action pairs but is also creates a new path of gradients over which the squared TD-error can be propagated though the policy and the Q-values at every time step of the sequence. When trained given observations that only contain the positions of the cars (e.g. no information about the state of the traffic lights) AMDRQN is able to find a policy that performs just as well as DRQN using observations containing light information. However, DRQN that receives one-hot encoded action information together with the binary position matrix performs just as well as AMDRQN. To further explore if AMDRQN could provide more stability and/or performance increase, future work should apply this new architecture to other POMDP's.

Finally, an unbiased gradient estimation of a POMDP was derived and translated into a surrogate loss function. When this loss function was used to train a deep recurrent Q-network the results were worse than using the squared TD-error as loss function. The performance was even worse than choosing random actions. It is unclear as to why this loss function performs this badly but it could have been caused by a bug in the software. When using automatic differentiation procedures it is difficult to know if the program really does what it is expected to do. Unfortunately, there was not enough time left in this thesis project to tweak the hyper parameters for this loss function. Changing the learning rate and the clipping size of the gradients is left future work. Leemon Baird introduced VAPS as a generalization of *residual gradient algorithms* which aim to minimize the mean squared Bellman residual instead of the mean squared TD-error [1]. These algorithms have some guarantees to converge even when using function approximations. Further exploring the application of this surrogate loss function and its relation to the VAPS algorithms may lead to deriving some convergence guarantees for DRQN.

12 Conclusion

This thesis applied a deep recurrent neural network to the POMDP of traffic light control to approximate the Q-function in order to find the optimal policy. Several design choices for recurrent neural networks were investigated in several experiments.

Q_1. *Can a deep reinforcement learning agent, using a recurrent neural network, learn to optimize the flow of traffic based only on one top-down image per time step of the traffic situation?*

When using a single 84x84 position matrix of the top-down image of the traffic crossing DRQN is able to learn a policy that outperforms its non-recurrent counterpart that uses two stacked matrices as input. It is even able to perform just as well as models using a position matrix with twice that resolution. This shows how well a recurrent neural network can extract information from incomplete data and why it is such a good tool to solve POMDP's.

Q_2. *How does the number of time steps the agent learns to remember affect the performance?*

A recurrent neural network is trained on sequences of data. The length of these sequences indicates how many time steps the gradients are backpropagated through time. The performance of the trained policies appears to increase along with the number of time steps that was used for the truncation of the BPTT algorithm, but not at the same rate. Meaning that the difference between $\tau = 4$ and $\tau = 10$ is greater than the difference between $\tau = 10$ and $\tau = 20$. The time it took to train the models increased linearly with respect to τ. The DRQN model with $\tau = 10$ is decided to be the best model as the increase in performance between $\tau = 10$ and $\tau = 20$ is not worth almost twice as much the training time.

The length of a sequences the network is trained on is not the only factor that affects the learning process. Each time step produces an output that can be evaluated and used to update the model. The results in this thesis indicate that the recurrent Q-learning agent is able to learn a stable policy that can be generalized for longer sequences by only using the error signal of the final time step of a sequence. Using more time steps in the parameter updates leads to a decline in performance and instability during learning.

Q_3. What is the effect of the addition of action probabilities to the input of the model on the performance?

This thesis introduced a new architecture called Action Motivation DRQN. When using observations that only contain information about the position of the cars, AMDRQN is able to outperform "normal" DRQN. However, it performs just as well as "normal" DRQN that is given action information as one-hot encoded vectors. Given action information DRQN finds a policy that performs just as well as "normal" DRQN that uses observations that include the traffic light configurations. However, AMDRQN seems slightly more stable when using action information as input which is a promising sign.

Q_4. Does using an unbiased gradient estimation stabilize and/or improve deep recurrent Q-learning?

The computation graph of a DRQN was created and from it an unbiased gradient estimation was derived. This gradient could be seen as a new version of the update rule of the Value and Policy Search algorithm [1]. It both includes the gradient of the squared TD-error and the gradient of the policy. These gradients were translated into a surrogate loss function and used to train a DRQN. The results presented in this thesis should be seen as preliminary results on this subject. Even though the results presented here are not very promising, they also indicate that there is a lot of instability that could be fixed. This thesis was not able to address these issues but that means that there is room to further explore the usage of the surrogate loss function.

Recurrent neural networks are a powerful tool in the field of deep reinforcement learning. They can outperform their feed forward counterparts with less input data, making them ideal to solve POMDP's. There is still a lot to explore in order to fully understand how to use these networks optimally in a reinforcement learning setting. Ideas about future work are presented below.

12.1 Future Work

The field of deep reinforcement learning has been very active for the last couple of years and new research is presented at a very high pace. This means that there are a lot of ideas that can be used to extend the work of this thesis or solve problems that have been identified.

12.1.1 Size of the Internal State

A recurrent neural network gets its unique properties from passing an internal state to itself over time. But what does this state represent? Is it an approximation of the state of the environment? If so, then the size of the internal state should be large enough to hold all the relevant information about a state. In the SUMO scenario that was used in this thesis, the agent received information about a cross-shaped intersection translated to a 84x84 matrix. Because every road consisted of two lanes the relevant information was limited to 4x84=336 positions of the matrix. It would be nice to test if an internal state with a dimension of 336 would have worked better.

12.1.2 Synthetic Gradients

From the experiments with the history size it seems that the longer the sequences are, the better the model performed. The computational costs, however, grow linearly with respect to the size of the history. The size of the history could be further extended more efficiently using synthetic gradients [10].

12.1.3 Dynamic Freeze Interval

The results from the freeze interval experiments indicate that the number of times the action and target networks are synchronized should be the same as the number of time steps the discounted return looks ahead. However, these intervals should be long enough to enable the model to minimize the TD-error. Instead of searching for the right parameter settings, the intervals could go on until the TD-error is minimized to a certain degree.

12.1.4 Prioritized Experience Replay

As the results in Section 9 suggest as well: the data presented to the model can have a great influence on the learning process. A lot of research on deep reinforcement learning uses or experiments with prioritized experience replay [29][37]. Using prioritized experience replay could further improve the performance and stability of DRQN, but it can also introduce new questions like: what is the priority of a sequence of transitions?

12.1.5 Further Exploration Hyper Parameters

The results in both Section 6 and 7 indicate that changing the learning rate in these experiments could have a big impact on the outcome. When using the errors at every time step, a smaller learning rate compensates for this increase in gradients and improve learning. When decreasing the freeze interval, the model has less time steps to minimize the squared TD-error making the predictions of the network imprecise. To compensate for these smaller intervals, the learning rate could be increased to speed up the minimization of the loss.

12.1.6 Action Motivation DRQN

The AMDRQN architecture was introduced in Section 8 and did not show any major contributions to the performance of deep recurrent Q-learning. Applying this model to other problems can give more insight in the possible contributions of this architecture.

A True Loss Derivative

The unbiased gradient estimation of the loss function defined in 23 following Theorem 1 of [30].

$$
\begin{aligned}
\mathcal{L}_\tau = \mathbb{E}_{H_\tau \sim p_\theta}\left[L_\tau\right] &= \sum_{H_\tau} p_\theta \; L_\tau \\
\frac{\partial}{\partial\theta}\mathcal{L}_\tau = \frac{\partial}{\partial\theta}\mathbb{E}_{H_\tau \sim p_\theta}\left[L_\tau\right] &= \sum_{H_\tau} \frac{\partial}{\partial\theta} p_\theta \; L_\tau \\
&= \sum_{H_\tau} \left(\frac{\partial}{\partial\theta} p_\theta\right) L_\tau + p_\theta \left(\frac{\partial}{\partial\theta} L_\tau\right) \\
&= \sum_{H_\tau} \left(p_\theta \frac{\partial}{\partial\theta} \log p_\theta\right) L_\tau + p_\theta \; \left(\frac{\partial}{\partial\theta} L_\tau\right) \\
&= \sum_{H_\tau} p_\theta \left[\left(\frac{\partial}{\partial\theta} \log p_\theta\right) L_\tau + \frac{\partial}{\partial\theta} L_\tau\right] \\
&= \mathbb{E}_{H_\tau \sim p_\theta} \left[\left(\frac{\partial}{\partial\theta} \log p_\theta\right) \hat{L}_\tau + \frac{\partial}{\partial\theta} L_\tau\right]
\end{aligned}
\tag{29}
$$

The step from line 3 to 4 is the log-derivative trick. This trick is the application of the rule for the gradient with respect to parameters θ of the logarithm of a function $p(x;\theta)$.

$$
\nabla_\theta \log p(x;\theta) = \frac{\nabla_\theta p(x;\theta)}{p(x;\theta)} \tag{30}
$$

The significance of this trick is realized when the function $p(x;\theta)$ is a likelihood function, i.e. a function of parameters θ that provides the probability of a random variable x. In this special case, the function $\nabla_\theta \log p(x;\theta)$ is called a score function, and the right-hand side is the score ratio.

$$
\begin{aligned}
\nabla_\theta p(x;\theta) &= \frac{p(x;\theta)}{p(x;\theta)} \nabla_\theta p(x;\theta) \\
&= p(x;\theta) \frac{\nabla_\theta p(x;\theta)}{p(x;\theta)} \\
&= p(x;\theta) \nabla_\theta \log p(x;\theta)
\end{aligned}
\tag{31}
$$

References

[1] Leemon C Baird III. *Reinforcement learning through gradient descent*. PhD thesis, US Air Force Academy, US, 1999.

[2] Bram Bakker. Reinforcement learning with long short-term memory. In *Advances in neural information processing systems*, pages 1475–1482, 2002.

[3] Anthony R Cassandra and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In *Machine Learning Proceedings 1995: Proceedings of the Twelfth International Conference on Machine Learning, Tahoe City, California, July 9-12 1995*, page 362. Morgan Kaufmann, 1995.

[4] Jakob N Foerster, Yannis M Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate to solve riddles with deep distributed recurrent q-networks. *arXiv preprint arXiv:1602.02672*, 2016.

[5] Michael C Fu. Gradient estimation. *Handbooks in operations research and management science*, 13:575–616, 2006.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[7] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2015.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] Tommi Jaakkola, Satinder P Singh, and Michael I Jordan. Reinforcement learning algorithm for partially observable markov decision problems. In *Advances in neural information processing systems*, pages 345–352, 1995.

[10] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. *arXiv preprint arXiv:1608.05343*, 2016.

[11] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *NIPS*, volume 13, pages 1008–1014, 1999.

[13] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. Recent development and applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, 5(3&4):128–138, December 2012.

[14] Stefan Krauß. *Microscopic modeling of traffic flow: Investigation of collision free vehicle dynamics*. PhD thesis, 1998.

[15] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. *arXiv preprint arXiv:1609.05521*, 2016.

[16] Erich Leo Lehmann and George Casella. *Theory of point estimation*. Springer Science & Business Media, 2006.

[17] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[18] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[19] Long-Ji Lin and Tom M Mitchell. *Memory approaches to reinforcement learning in non-Markovian domains*. Carnegie-Mellon University. Department of Computer Science, 1992.

[20] R Andrew McCallum. Instance-based utile distinctions for reinforcement learning with hidden state. In *ICML*, pages 387–395, 1995.

[21] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[24] Ronald Parr and Stuart Russell. Approximating optimal policies for partially observable stochastic domains. In *IJCAI*, volume 95, pages 1088–1094, 1995.

[25] Silvia Richter, Douglas Aberdeen, Jin Yu, et al. Natural actor-critic for road traffic optimisation. *Advances in neural information processing systems*, 19:1169, 2007.

[26] Tobias Rijken. *DeepLight: Deep reinforcement learning for signalised traffic control*. PhD thesis, Masters Thesis. University College London, 2015.

[27] Christian P Robert. *Monte carlo methods*. Wiley Online Library, 2004.

[28] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

[29] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[30] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3528–3536, 2015.

[31] Satinder P Singh, Tommi S Jaakkola, Michael I Jordan, et al. Learning without state-estimation in partially observable markovian decision processes. In *ICML*, pages 284–292, 1994.

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

[33] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

[34] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[35] Elise van der Pol. *Deep reinforcement learning for coordination in traffic light control*. PhD thesis, Masters Thesis. University of Amsterdam, 2016.

[36] Elise Van der Pol and Frans A. Oliehoek. Coordinated deep reinforcement learners for traffic light control. In *NIPS'16 Workshop on Learning, Inference and Control of Multi-Agent Systems*, 2016.

[37] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[38] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[39] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[40] Brian Wolshon and Anurag Pande. *Traffic Engineering Handbook*. John Wiley & Sons, 2016.

[41] Pengfei Zhu, Xin Li, and Pascal Poupart. On improving deep reinforcement learning for pomdps. *arXiv preprint arXiv:1704.07978*, 2017.