Deep Reinforcement Learning for Urban Traffic Light Control

Noé Casas

Department of Artificial Intelligence
Universidad Nacional de Educación a Distancia

This dissertation is submitted for the degree of
Master in Advanced Artificial Intelligence

Advisors:
Elena Gaudioso Vázquez, Félix Hernández del Olmo

February 2017

Abstract

Traffic light timing optimization is still an active line of research despite the wealth of scientific literature on the topic, and the problem remains unsolved for any non-toy scenario. One of the key issues with traffic light optimization is the large scale of the input information that is available for the controlling agent, namely all the traffic data that is continually sampled by the traffic detectors that cover the urban network. This issue has in the past forced researchers to focus on agents that work on localized parts of the traffic network, typically on individual intersections, and to coordinate every individual agent in a multi-agent setup. In order to overcome the large scale of the available state information, we propose to rely on the ability of deep Learning approaches to handle large input spaces, in the form of Deep Deterministic Policy Gradient (DDPG) algorithm. We performed several experiments with a range of models, from the very simple one (one intersection) to the more complex one (a big city section).

Contents

List of Figures

Chapter 1

Introduction

1.1 Presentation of the Problem

Cities are characterized by the evolution of their transit dynamics. Originally meant solely for pedestrians, urban streets soon shared usage with carriages and then with cars. Traffic organization became soon an issue that led to the introduction of signaling, traffic lights and transit planning.

Nowadays, traffic lights either have fixed programs or are actuated. Fixed programs (also referred to as *pretimed control*) are those where the timings of the traffic lights are fixed, that is, the sequences of red, yellow and green phases have fixed duration. Actuated traffic lights change their phase to green or red depending on traffic detectors that are located near the intersection; this way, actuated traffic light are dynamic and adapt to the traffic conditions to some degree; however, they only take into account the conditions local to the intersection. This also leads to dis-coordination with the traffic light cycles of other nearby intersections and hence are not used in dense urban areas.

Neither pretimed or actuated traffic lights take into account the current traffic flow conditions at the city level. Nevertheless, cities have large vehicle detector infrastructures that feed traffic volume forecasting tools used to predict congestion situations. Such information is normally only used to apply classic traffic management actions like sending police officers to divert part of the traffic.

This way, traffic light timings could be improved by means of machine learning algorithms that take advantage of the knowledge about traffic conditions by optimizing

the flow of vehicles.

This has been the subject of several lines of research in the past. For instance, Wiering proposed different variants of reinforcement learning to be applied to traffic light control [97], and created the *Green Light District* (GLD) simulator to demonstrate them, which was further used in other works like [68]. Several authors explored the feasibility of applying fuzzy logic, like [32] and [20]. Multi-agent systems where also applied to this problem, like [16] and [74].

Most of the aforementioned approaches simplify the scenario to a single intersection or a reduced group of them. Other authors propose multi-agent systems where each agent controls a single intersection and where agents may communicate with each other to share information to improve coordination (e.g. in a *connected vehicle* setup [34]) or may receive a piece of shared information to be aware of the crossed effects on other agents' performance ([29]). However, none of the aforementioned approaches fully profited from the availability of *all* the vehicle flow information, that is, the decisions taken by those agents were in all cases partially informed.

The main justification for the lack of *holistic* traffic light control algorithms is the poor scalability of most algorithms. In a big city there can be thousands of vehicle detectors and tenths of hundreds of traffic lights. Those numbers amount for huge space and action spaces, which are difficult to handle by classical approaches.

This way, the problem addressed in this works is the devisal of an agent that receives traffic data and, based on these, controls the traffic lights in order to improve the flow of traffic, doing it at a large scale.

1.2 Traffic Simulation Concepts

Before further exploring the problem, we shall briefly describe some concepts related to traffic and traffic simulation, to provide some context to better understand the this work.

In order to evaluate the performance of our work, we make use of a traffic simulator. We chose a third party traffic simulator software that allows to model a traffic scenario, with roads, streets, traffic lights, etc. The traffic simulation concepts described in this section be specific to the microscopic simulator used in our experiments, namely

Aimsun (see section A.1 for details on the justification for the selection).

The base of a traffic simulation is the **network**, that is, the representation of roads and intersections where the vehicles are to move. Connected to some roads, there are **centroids**, that act as sources/sinks of vehicles. The amount of vehicles generated/absorbed by centroids is expressed in a **traffic demand matrix**, which contains one cell per each pair of origin and destination centroids. During a simulation, different OD matrices can be applied to different periods of time in order to mimic the dynamics of the real traffic through time.

In the roads of the network, there can be **traffic detectors**, that mimic induction loops beneath the ground that are able to measure traffic data as vehicles go pass through them. Typical measurements that can be taken with traffic detectors include vehicle counts, average speed and percentage of occupancy.

There can also be traffic lights. In many cases they are used to regulate the traffic at intersections. In those cases, all the traffic lights in an intersection are coordinated so that when one is red, another one is green, and vice versa (this way, the use of the intersection is regulated so that vehicles don't block the intersection due to their intention to reach an exit of the intersection that is currently in use) . All the traffic lights in the intersection change their state at the same time. This intersection-level configuration of the traffic lights is called a **phase**, and it is completely defined by the states of each traffic light in the intersection plus its duration. The different phases in an intersection form its *control plan*. The phases in the control plan are applied cyclically, so the phases are repeated after the *cycle duration* elapses. Normally, control plans of adjacent intersections are synchronized to maximize the flow of traffic avoiding unnecessary stops.

When the simulation starts, the emitting centroids start generating vehicles that are headed towards their destination centroids, according to the amounts expressed in the demand matrix. In order for the simulation not to start *too empty*, it is possible to set up a **warm-up** period, during which the centroids generate vehicles so that when the simulation starts, there are more realistic volume conditions.

In figure 1.1, it is shown a very simple traffic network in Aimsun. In consists of a single intersection. The image belongs to a simulation step where we can see vehicles stoped waiting because the traffic light is red, other vehicles circulating with the green traffic

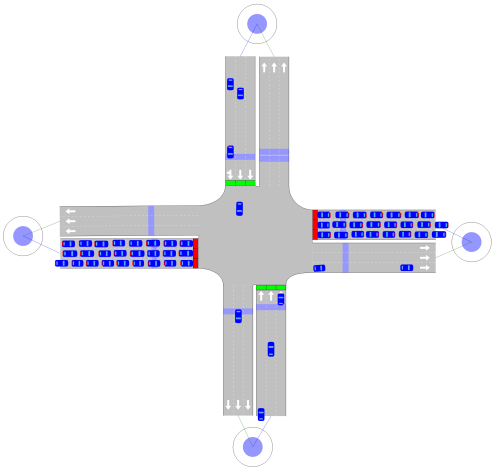

Figure 1.1 Simple traffic network in Aimsun microscopic simulator.

light, centroids emitting and absorbing vehicles and traffic detectors at the entrances and exists of the intersection.

1.3 Preliminary Analysis

The main factor that has prevented further advance in the traffic light timing control problem is the large scale of any realistic experiment. On the other hand, there is a family of machine learning algorithms whose very strenght is their ability of handle large input spaces, namely **deep learning**. Recently, deep learning has been successfully applied to reinforcement learning, gaining much attention due to the effectiveness of Deep Q-Networks (DQN) at playing Atari games using as input the raw pixels of the game [60, 61]. Subsequent successes of a similar approach called Deep Deterministic Policy Gradient (DDPG) were achieved in [77] and [57], which will be used in our work as reference articles, given the similarity of the nature of the problems addressed there, namely large continuous state and action spaces.

This way, the theme of this thesis is the **application of Deep Reinforcement Learning to the traffic light optimization problem** with an holistic approach, by leveraging deep learning to cope with the large state and action spaces. Specifically, the **hypothesis** that drives this work is that *Deep reinforcement learning can be successfully applied to urban traffic light control, having similar or better performance than other approaches.*

This is hence the main contribution of the present work, along with the different techniques applied to make this application possible and effective.

Taking into account the nature of the problem and the abundant literature on the subject (explored in detail in chapter 5), we know that some of the challenges of devising a traffic light timing control algorithm that acts at a large scale are:

- Define a sensible **input space**. This includes finding a suitable representation of the traffic information. Deep learning is normally used with input signals over which convolution is easily computable, like images (i.e. pixel matrices) or sounds (i.e. 1-D signals). Traffic information may not be easily represented as a matrix, but as a labelled graph.
- Define a proper **action space** that our agent is able to perform. The naive approach would be to let the controller simply control the traffic light timing directly (i.e. setting the color of each traffic light individually at each simulation step). This, however, may lead to breaking the normal routing rules, as the traffic lights in an intersection have to be synchronized so that the different intersection exit routes do not interfere with each other. Therefore a careful definition of the agent's actions is needed.
- Study and ensure the convergence of the approach: despite the successes of Deep Q-Networks and DDPG, granted by their numerous contributions to the stability of reinforcement learning with value function approximation, convergence of such approaches is not guaranteed. Stability of the training is studied and measures for palliating divergence are put in place.
- Create a sensible **test bed**: a proper test bed should simulate relatively realistically the traffic of a big city, including a realistic design of the city itself.

1.4 Structure

This thesis is presents the following structure: in the present chapter we have provided an overview of the problem, stating clearly the goals of this work. In chapter 2 we provide background on the techniques on which our proposed approach relies on. In chapter 3 we describe the approach itself, while in chapter 4 we explain the experiments undergone to test our proposal, as well as the obtained results. In chapter 5, we provide an overview of previous works that are closest to ours in terms of application domain

and techniques used. Finally, in chapter 6 we discuss the obtained results, reflect on the conclusions that can be drawn from them and propose further lines of research. As an addendum, in appendix A we provide justification of the chosen traffic simulator and also provide information on the hardware setup used to execute all our tests, and in appendix B we provide an overview of techniques that we applied to the problem but did not succeed.

Chapter 2

Background

In this chapter we provide a thorough review of the algorithms on top of which the proposed approach relies, starting with neural networks and deep learning in section 2.1, followed by the classical reinforcement learning theory in section 2.2 and finally making both fields converge into deep reinforcement learning in section 2.3.

2.1 Deep Learning

Artificial Neural Networks (ANNs) are machine learning models loosely inspired in biological neural networks, in that there are discrete units referred to as *neurons* that are interconnected. The organization of the network defines its *architecture*. There are neural network architectures that are meant for supervised learning (e.g. multi-layer perceptrons) while others are for unsupervised learning (e.g. self-organizing maps).

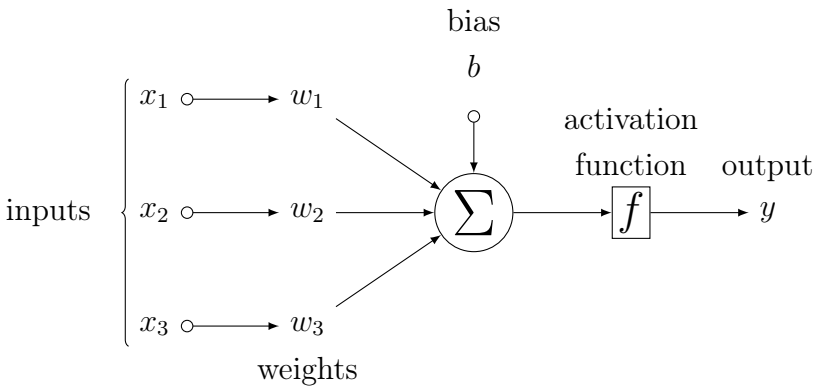

Figure 2.1 Perceptron

The history of ANNs formally started in 1958 with the Perceptron (MLP) by Rosenblatt [70], depicted in figure 2.1, which works as a linear approximator followed by a step function and could be used for classification problems.

Despite the initial enthusiasm over the perceptron, some flaws where identified on it in [40], most remarkably the impossibility for the perceptron to address not linearly separable problems, like the exclusive or (XOR) operation. This led to the abandonment of the research in ANNs during several years, contributing to the so-called AI winter, a period in which the funding for AI research in the USA was cut down drastically.

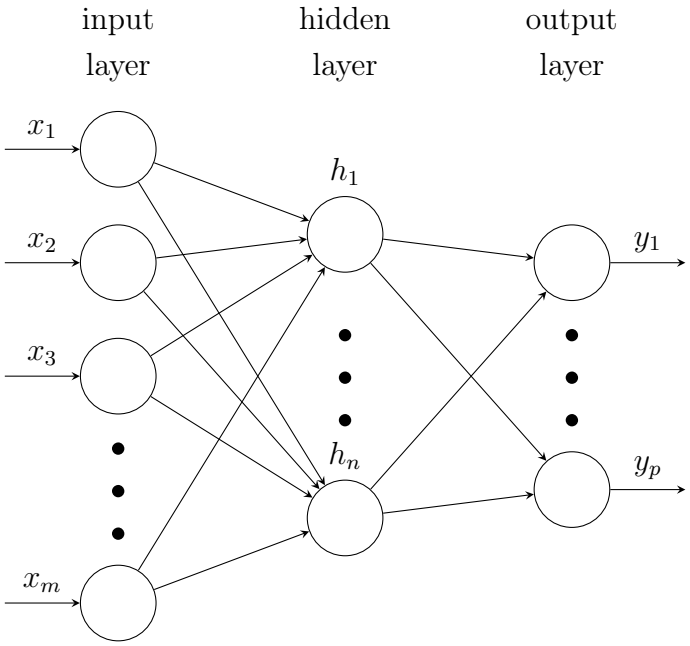

Figure 2.2 Multi-Layer Perceptron

The interest in *connectionist* approaches raised again in the 70s and 80s due to the work of Werbos [95], who proposed a multi-layer version of the perceptron that was able to address the XOR problem plus a new approach to train it: the **backpropagation algorithm**, which is currently the preferred training methods for most neural network architectures. The multi-layer perceptron is depicted in figure 2.2.

During the 90s and early 2000s, artificial neural network research continued at a lower profile until there was a breakthrough: an implementation of a convolutional neural network running on a GPU outperformed all other competitors at the ImageNet 2012 contest [54]. That event started a stage where deep learning approaches succeeded in several areas that previously were not addressable, like the case of AlphaGo, a deep

learning-based Go player by Google DeepMind that beat the human world champion Lee Sedol [77].

The field of deep learning is very wide, comprising both supervised and unsupervised learning realms. Within the deep learning landscape, the following techniques are used in the present work:

- **Stochastic Gradient Descent with minibatches** (mini-batch SGD): feedforward neural networks are in most cases trained by means of the backpropagation algorithm, which updates the weights of the different layers by differentiating the loss function with respect of the network weights, and performing gradient descent optimization. This implies summing up the error committed for every individual in the training set. This does not scale well when the training set is very large, as it becomes computationally unaffordable to iterate over the whole set in each iteration. This led to stochastic gradient descent, which only computes the loss of a single randomly selected training individual at each iteration. The hybrid approach is to select a collection of randomly selected individuals (i.e. the mini batch) from the training set in each iteration and compute the accumulated loss over it. Mini-batch stochastic gradient descent also presents better convergence behaviour than the classic gradient descent due to its ability to better escape local minima. There are several variations of the SGD algorithm, mainly proposing different ways of adding momentum (e.g. Nesterov momentum [63], Adam optimizer [51]).

- **Rectified Linear Units** (ReLU): neural network layers have an *activation function* that characterizes the output of the layer. In hidden layers, it is common for the activation to be non-linear to add expressive power to the network. Activation functions need to be differentiable in order to allow backpropagation training. Typical activations are the sigmoid function and the hyperbolic tangent, both resembling a step function. Such activation functions suffered an important problem when the network comprised several layers: the *vanishing gradients* problem: when the input weights of a neuron led it to enter its saturation regimes (the values of the input for which the output is 0 or 1), the neuron *got trapped* and could not return back to the non-saturating regime, hence being unable to further learn through training. This was first described in [11] and further studied in [65]. However, in [62] it was proposed to use a rectified linear activation unit (i.e. $f(x) = \max(0, x)$), known as ReLU, which mitigates the vanishing gradients

problem due to being non-saturating on the positive semi axis. Despite being a linear function in half on its domain, it is considered in all regards a non-linearity. The key difference between a linear activation and a ReLU is that adding up multiple linear activations result always in a linear function, while adding up multiple ReLUs result in non-linear functions that can be arbitrarily complex.

- **Leaky ReLU** [58]: a small modification to the standard ReLU that instead of clipping the output when the input is in the range $[-\infty, 0]$, outputs a linear signal with small negative slope, as described in (2.1)

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases} \tag{2.1}$$

  The purpose of the Leaky ReLU is to avoid the problem of the *dying ReLU* ([50]). A dead ReLU always outputs zero, normally due to having learned a large negative bias for its weights, which makes it impossible to recover. The small negative slope of the Leaky ReLU makes it impossible to get *trapped* into such an state.

  Other approaches proposed to handle the dying ReLU problem are Exponential Linear Units (ELU) ([22]) and Maxout units ([39]).

- **Dropout layers** [78]: a regularization method that disables randomly neurons in a layer with certain probability. This forces the network to learn several different representations, therefore avoiding overfitting. As noted in [9] the effect of dropout in one logistic unit is that of *geometric averaging* over the ensemble of the networks resulting from the removal of units, therefore providing more robust learning. The same effect is conjectured to multilayer networks.

- **Convolutional networks**: convolutional networks is an umbrella term, normally used to refer to an arbitrary combination of convolutional layers, pooling layers and ReLU activations, together with a final block of dense layer forming a multilayer perceptron. A **convolutional layer** consists of one or several matrices (referred to as patches) of small size (3x3 or 5x5 at most) that slide throughout the input matrix being applied as a discrete convolution. This way, each cell of the output matrix is computed by centering the patch on a pixel of the input matrix and multiplying the matrix values with the associated input matrix cells and adding them up. The *weights* to be optimized in a convolutional layer are the

cells of the kernels used for the convolution. Convolutional layers effectively are *feature detectors*, that is, they detect local features of the input data. Following a convolutional layer, there is normally a *pooling layer*, which subsamples the output of the convolution. The main variants are max pooling and average pooling, which respectively take the maximum value in a region or the average one. The pooling regions are normally very small in order not to loose to much information. Finally, there is usually a **ReLU activation**.

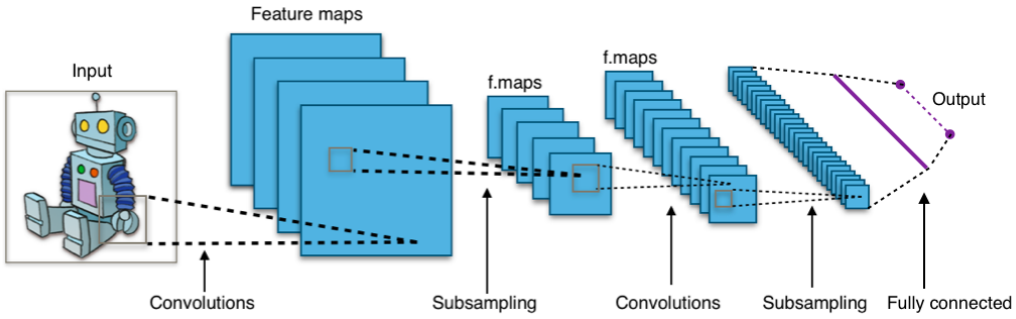

Figure 2.3 Typical architecture of a convolutional neural network

As illustrated in figure 2.3 [1], this **convolutional block** comprised of convolution, pooling and ReLU has become a *de facto* building block for very deep neural networks in the computer vision domain. The concatenation of several of such blocks result in the detection of *hierarchical features*, that is, features that are themselves composed of lower level features.

- **Weight initialization**: the learning process of neural networks consists in optimizing the loss function over the network parameter space (i.e. the network's weights), which is non-convex. In convex function iterative optimization, the point where the search begins is irrelevant, because regardless of it, the final point will be the optimum. However, in non-convex optimization like neural network learning, the initial point where the parameter space exploration begins, highly influences the local optimum reached (together with the loss function shape, the activations and the optimization algorithm). One common weight initialization strategy is to assign random weights, but scaling with the number of inputs in order to keep the total variance in the levels that allow sigmoid and tanh activations to work in the linear regime, therefore improving the convergence speed. An analysis by Glorot et al. in [37] proposed to use as initialization scale

[1] This image is under Creative Commons Attribution-Share Alike 4.0 International license and was taken from https://commons.wikimedia.org/wiki/File:Typical_cnn.png

$2/(N_{in} + N_{out})$, while the improved analysis by He et al. in [44] extended the analysis to ReLU's, proposing $2/n$ as the scaling factor for weights, where n is the number of neurons in the network.

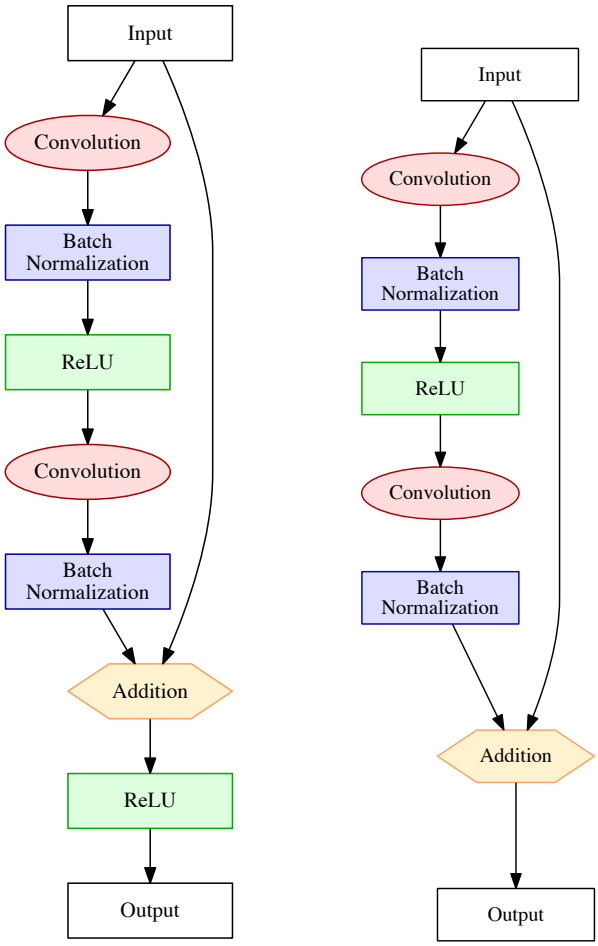

Figure 2.4 Different variants of residual learning

- **Residual learning**: expressive power of neural networks is directly dependent on their number of layers (i.e. their depth)[1]. However, adding more layers to a network induces overfitting to the training set. That problem is controlled via

[1]Note that despite the fact that multilayer percentrons with one single hidden layer with nonlinear activations are universal approximators ([46]), there is no theoretical results that determine the number of units that are needed to approximate an arbitrary given function. There is, nervertheless, documented evidence of the improvement in the performance of neural networks when more hidden layers are added, which suggests that having more hidden layers is related to the *ability* of the network to model non linear functions [38]

regularization, dropout and batch normalization [47], there is a limit in the gain of accuracy by simply adding layers: beyond a certain number of layers, there is higher training and validation error. He et al. propose in [43] a strategy called *residual learning* that enables networks to gain accuracy by adding more layers, consisting in setting up bypassing connections, that is, adding a connection from the input to the output of a block of processing layers. Residual learning is usually applied to convolutional networks, which tend to combine convolutional layers with ReLU activations and batch normalization; its application consists of dividing the neural architecture in processing blocks and adding a bypassing connection from the block input to the block output, and combining it with the normal block processing by adding them up and optionally adding a ReLU block afterwards, as shown in figure 2.4. There are other approaches that also allow deeper architectures, like *highway networks* [79], but they introduce high complexity and do not present significant improvement over residual learning.

2.2 Reinforcement Learning

Machine learning algorithms are typically categorized as either **supervised** learning algorithms or **unsupervised** learning algorithms. Supervised learning refers to algorithms that are trained to cast certain output (e.g. label) when presented with certain input. Some examples of supervised learning problems are regression and classification. On the other hand, unsupervised learning algorithms are only presented with input data and the training consists in finding a representation of those data describing its *hidden structure*. Some examples of unsupervised learning problems are clustering and dimensionality reduction.

However, there is a third category of machine learning algorithms referred to as ***reinforcement learning*** (RL), where the goal is to train an agent so that it behaves optimally in an *environment*, with the downside that it is not known which actions are good or bad, but it is possible to evaluate the goodness of their effects after they are applied. Using RL terminology, the goal of the algorithm is to learn an optimal policy for the agent, based on the observable state of the environment and on a *reinforcement signal* that represents the reward (either positive or negative) obtained when an action has been applied. The underlying problem that reinforcement learning tries to solve is that of the *credit assignment*. For this, the algorithm normally tries to estimate the expected cumulative future reward to be obtained when applying certain action when

in certain state of the environment.

RL algorithms act at discrete points in time. At each time step t, the agent tries to maximize the expected total return R_T, that is, the accumulated rewards obtained after each performed action: $R_t = r_{t+1} + r_{t+2} + \cdots + r_T$, where T is the number of time steps ahead until the problem finishes. However, as normally T is dynamic or even infinite (i.e. the problem has no end), instead of the summation of the rewards,the discounted return is used:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

The state of the environment is observable, either totally or partially. The definition of the state is specific to each problem. One example of state of the environment is the position x of a vehicle that moves in one dimension. Note that the state can certainly contain information that condenses pasts states of the environment. For instance, apart from the position x from the previous example, we could also include the speed $\dot{x}$ and acceleration $\ddot{x}$ in the state vector.

Reinforcement Learning problems that depend only on the current state of the environment are said to comply with the *Markov property* and are referred to as *Markov Decision Processes*. Their dynamics are therefore defined by the probability of reaching from a state s to a state s' by means of action a:

$$p(s'|s,a) = P(S_{t+1} = s'|S_t = s, A_t = a) \tag{2.3}$$

This way, we can define the reward obtained when transitioning from state s to s' by means of action a:

$$r(s,a,s') = \mathbb{E}\left[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'\right] \tag{2.4}$$

In the following sections, we review different classic algorithms used to address RL problems. They are the base of modern **deep reinforcement learning** algorithms, which the current work relies upon, and which are described in section 2.3.

2.2.1 Value Iteration Algorithms

The *value* of a state, $v_\pi(s)$ is a measurement of the expected total return R obtained when we start from that state s and follow the policy π. The value function can be hence defined as:

$$v_\pi(s) = \mathbb{E}_\pi[R_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t = s\right] \tag{2.5}$$

Value iteration RL algorithms try to estimate the value function in an iterative way, that is, modifying at each time step the previous estimation for each state with the posible rewards that can be gained by the best possible action:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(s_{t+1})|S_t = s, A_t = a\right] = \\ &= \max_a \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma v_k(s')\right] \end{aligned} \tag{2.6}$$

With the value function, we can devise a greedy policy that always chooses the action for the current state that will provide the greatest reward, that is:

$$\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')] \tag{2.7}$$

However, in order to be able to compute the value function, we need full knowledge on the state transition probabilities and the associated rewards.

2.2.2 Q-learning and SARSA

Most RL algorithms rely on Bellman's equation (2.8), which describes the expected long term reward (i.e. the value V) for taking the action prescribed by some policy π when in state s knowing its immediate reward $R(s, \pi(s))$.

$$V^\pi(s) = R(s,\pi(s)) + \gamma \sum_{s'} P(s'|s,\pi(s))V^\pi(s') \tag{2.8}$$

Q-learning, which is one of the landmark RL algorithms, does not focus on the value function, but defines a different measure Q (2.9) that does not need *a priori* knowledge on the state transition probabilities or rewards.

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha\left(R_{t+1} + \gamma \max_a Q_t(s_{t+1},a) - Q_t(s_t,a_t)\right) \tag{2.9}$$

This way, the complete Q-Learning algorithm is as defined as follows:

Algorithm 1 Q-learning algorithm

1: Initialize $Q(s, a)$ arbitrarily
2: **for** each episode **do**
3: Initialize s
4: **for** each step of episode (until s is terminal) **do**
5: Choose a from s using policy derived from Q (e.q. ϵ-greedy)
6: Take action a, observe r, s'
7: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \cdot max_{a'} Q(s', a') - Q(s, a)]$
8: $s \leftarrow s'$
9: **end for**
10: **end for**

The SARSA (State-Action-Reward-State-Action) algorithm is very similar to Q-Learning. Their difference is that SARSA learns action values relative to the policy it follows, while Q-Learning learns the Q value is not constrained to only learning the Q values from the actions it takes at every step. This way, as shown in the complete SARSA algorithm below, in line 7, a' is chosen based on the followed policy and then in line 8 the Q values are updated on the basis of the chosen action a' and resulting state s' Q value, therefore only learning from the policy it follows (i.e. *on-policy*). On the other hand, in Q-learning the Q function is updated not only from the Q values followed, as shown in line 7 of algorithm 1 (i.e. *off-policy*).

Algorithm 2 SARSA algorithm

1: Initialize $Q(s, a)$ arbitrarily
2: **for** each episode **do**
3: Initialize s
4: Choose a from s using policy derived from Q (e.q. ϵ-greedy)
5: **for** each step of episode (until s is terminal) **do**
6: Take action a, observe r, s'
7: Choose a' from s' using policy derived from Q (e.q. ϵ-greedy)
8: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
9: $s \leftarrow s', a \leftarrow a'$
10: **end for**
11: **end for**

The original formulation of Q-learning and SARSA ([82]) represents $Q(s, a)$ in tabular form, that is, keeping a table with a cell for every combination of state and action. This implies that both the state space and the action space are discrete. For the cases where the state space or the action space were continuous, it was frequent to use *tile coding*, which consists of partitioning the space into non-overlapping regions (e.g. [80]), each of which was assimilated to a discrete value that could be used to index the tabular form of Q.

2.2.3 Value Function Approximation

In order to avoid having to partition of the state space, it is common to use a value function approximator instead of a table. Neural networks are a popular choice as approximator. The earliest success story for such an approach was TD-Gammon [85], in 1995, based on temporal difference learning [81, 84], with eligibility traces algorithm, usually referred to as, TD(λ). The neural network is meant to estimate the value of a certain state and its the parameter vector θ is updated as expressed in (2.10).

$$\theta_{t+1} = \theta_t + \alpha\big[Y(s_{t+1}) - Y(s_t)\big] \sum_{k=1}^{t} \lambda^{t-k} \nabla_\theta Y_k \tag{2.10}$$

Where $Y(s_i)$ is the output of the network when its input is the state s_i, α is the learning rate, λ is the eligibility trace factor, k is the number of traces to bufferize and $\nabla_\theta y_k$ is the gradient of the network output with respect to the weights θ.

This is the very origin of deep reinforcement learning, which is described in detail in section 2.3.

2.2.4 Actor-Critic

Actor-Critic methods separate the representation of the policy (referred to as *actor*) and the value function (known as *critic*). The role of the actor is to generate actions, while the role of the critic is to measure the performance of the actions and generating a temporal difference error (2.11) ([82]).

$$\delta_t = r_{t+q} + \gamma \cdot V(S_{t+1}) - V(s_t) \tag{2.11}$$

The actions are normally decided by means of the softmax method (2.12):

$$\pi_t(s,a) = P(a_t = a|s_t = s) = \frac{e^{p(s,a)}}{\sum_b e^{p(s,b)}} \tag{2.12}$$

And the actor is then updated with the TD error and a step size β as in (2.13).

$$p(s_t, a_t) \leftarrow p(s_y, a_t) + \beta\delta_t \tag{2.13}$$

2.2.5 Multi-Agent Reinforcement Learning

When reinforcement learning is applied to a problem, one of the aspects that most influences the set of applicable approaches is the nature of the actions available to the agent. Some examples of different alternative action spaces and the type of agents usually applied to them are:

- There is only a few amount of enumerable actions that can be applied one at a time. Most basic reinforcement learning algorithms (e.g. tabular Q-learning, SARSA) are specifically designed for this kind of scenario, normally coupled with an also low dimensionality and easily quantizable state space.

- The action space is continuous with low dimensionality. Actor-critic methods are suitable to address such a case, as the actor can generate any type of action that is accepted by the critic (and also by the environment). Nevertheless, the capacity of the actor bounds the dimensionality of the action space, that is, the more dimensions, the more capacity[1] the actor needs.

- The action space is high dimensional, either continuous or discrete. They are challenging for traditional reinforcement learning approaches due to their reduced scalability. A usual approach is to divide the problem into smaller sub-problems where the action space is manageable by a traditional RL agent, therefore turning the solution into a multi-agent system, and hence being referred to as Multi-Agent Reinforcement Learning (MARL). A comprehensive and up to date survey of the status of such a field can be found in [15]. Some examples of their application to traffic control are [10] and [55]. The problem with MARL approaches is that they introduce further challenges due to the need for coordinating the different agents in order to avoid them obstructing each other.

[1] The term *capacity* refers to the concept denoted by the VC dimension ([92])

2.3 Deep Reinforcement Learning

Deep Reinforcement Learning refers to reinforcement learning algorithms that use a deep neural network as value function approximator, as described in section 2.2.3. Their recent rise in popularity is due to the success of Deep Q-Networks (DQN) at playing Atari games using as input the raw pixels of the game [60, 61].

$$\mathcal{L}(\theta) = \mathbb{E}\left[(y - Q(s, a; \theta))^2\right] \tag{2.14}$$

In DQNs, there is a neural network that receives the environment state as input and generates as output the Q-values for each of the possible actions, using the loss function (2.14), which implies following the direction of the gradient (2.15):

$$\nabla_\theta \mathcal{L}(\theta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)\right) \nabla_\theta Q(s, a; \theta)\right] \tag{2.15}$$

The first success of reinforcement learning with neural networks as function approximation was TD-Gammon [85]. Despite the initial enthusiasm in the scientific community, the approach did not succeed when applied to other problems, which led to its abandonment ([67]). The main reason for its failure was **lack of stability** derived from:

- The neural network was trained with the values that were generated *on the go*, therefore such values were sequential in nature and thus **highly correlated with the near past values** (i.e. not independently and identically distributed).
- **Oscillation of the policy** with small changes to Q-values that change the data distribution.
- Too large optimization steps when **large rewards** are got.

In order to mitigate such stability problems, in [60, 61], the authors applied the following measures:

- **Experience replay**: keep a memory of past action-rewards and train the neural network with random samples from it instead of using the real time data, therefore eliminating the temporal autocorrelation problem. This is possible thanks to the ***off-policy*** nature of Q-learning that allows to feed it with updates not necessarily coming from the policy being followed.
- **Reward clipping**: scale and clip the values of the rewards to the range $[-1, +1]$ so that the weights do not boost when backpropagating.

- **Target network**: keep a separate DQN so that one is used to compute the target values and the other one accumulates the weight updates, which are periodically loaded onto the first one. This avoid oscillations in the policy upon small changes to Q-values.

However, DQNs are meant for problems with a few possible actions, and are therefore not appropriate for continuous space actions, like in our case. On the other hand, **Deep Deterministic Policy Gradient** or DDPG ([57]) naturally accommodates this kind of problems. As its name suggests, it combines the actor-critic classical RL approach [82] with Deterministic Policy Gradient [76].

The original formulation of the policy gradient algorithm was proposed in [83], which proved the policy gradient theorem (theorem 1) for a stochastic policy $\pi(s, a; \theta)$:

Theorem 1. *(**policy gradient**) For any MDP, if the parameters θ of the policy are updated proportionally to the gradient of its performance ρ then θ can be assured to converge to a locally optimal policy in ρ, being the gradient computed as*

$$\Delta\theta \approx \alpha \frac{\partial \rho}{\partial \theta} = \alpha \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \theta} Q^\pi(s,a)$$

with α being a positive step size and where d^π is defined as the discounted weighting of states encountered starting at s_0 and then following π: $d^\pi(s) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s | s_0, \pi)$

This theorem was further extended in the same article for the case where an approximation function f is used in place of the policy π, as shown in theorem 2:

Theorem 2. *(**policy gradient with function approximation**) The policy gradient theorem holds valid for a function approximation $f(s, a; w)$ of the policy π if the updates of the weights w tend to zero upon convergence to π:*

$$\sum_s d^\pi(s) \sum_a \pi(s,a)[Q^\pi(s,a) - f(s,a;w)] \frac{\partial f(s,a;w)}{\partial w} = 0$$

and if f is compatible with the policy parameterization in the sense that:

$$\frac{\partial f(s,a;w)}{\partial w} = \frac{\partial \pi(s,a)}{\partial \theta} \frac{1}{\pi(s,a)}$$

Then,

$$\frac{\partial \rho}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \theta} f(s,a;w)$$

In our reference articles [76] and [57], the authors propose to use a **deterministic policy** (as opposed to stochastic) approximated by a neural network actor $\pi(s;\theta^\pi)$ that depends on the state of the environment s and has weights θ^π, and another separate network $Q(s,a;\theta^Q)$ implementing the critic, which is updated by means of the Bellman equation (2.8) like DQN (2.15):

$$Q(s_t,a_t) = \mathbb{E}_{r_t,s_{t+1}}\left[r(s_t,a_t) + \gamma Q(s_{t+1},\pi(s_{t+1}))\right] \tag{2.16}$$

And the actor is updated by applying the chain rule to the loss function (2.14) and updating the weights θ^π by following the gradient of the loss with respect to them:

$$\nabla_{\theta^\pi}\mathcal{L} \approx \mathbb{E}_s\left[\nabla_{\theta^\pi} Q(s,\pi(s|\theta^\pi)|\theta^Q)\right] = \mathbb{E}_s\left[\nabla_a Q(s,a|\theta^Q)|_{a=\pi(s|\theta^\pi)} \nabla_{\theta^\pi}\pi(s|\theta^\pi)\right] \tag{2.17}$$

In order to introduce exploration behaviour, thanks to the DDPG algorithm being off-policy, we can add random noise $\mathcal{N}$ to the policy:

$$\pi'(s) = \pi(s;\theta^\pi) + \mathcal{N} \tag{2.18}$$

This enables the algorithm to try unexplored areas from the action space to discover improvement opportunities, much like the role of ε in ε-greedy policies in Q-learning.

In order to improve stability, DDPG also can be applied the same measures as DQNs, namely reward clipping, experience replay (by means of a *replay buffer* referred to as R in algorithm 3) and separate target network. In order to implement this last measure for DDPG, two extra target actor and critic networks (referred to as π' and Q' in algorithm 3) to compute the target Q values, separated from the normal actor and critic (referred to as π and Q in algorithm 3) that are updated at every step and which weights are used to compute *small updates* to the target networks.

The complete resulting algorithm, as proposed in [57], is summarized as follows:

Algorithm 3 Deep Deterministic Policy Gradient algorithm

Randomly initialize critic $Q(s, a|\theta^Q)$ and actor $\pi(s|\theta^\pi)$ with weights θ^Q and θ^π.
Initialize target network Q' and π' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\pi'} \leftarrow \theta^\pi$.
Initialize replay buffer R.
for each episode **do**
  Initialize random process $\mathcal{N}$ for action exploration.
  Receive initial observation state s_1.
  for each step t of episode **do**
    Select action $a_t = \pi(s_t|\theta^\pi) + \mathcal{N}_t$.
    Execute action a_t and observe reward r_t and new state s_{t+1}.
    Store transition (s_t, a_t, r_t, s_{t+1}) in R.
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R.
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \pi'(s_i + 1|\theta^{\pi'})|\theta^{Q'})$.
    Update critic by minimizing the loss: $\mathcal{L} = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$.
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\pi}\mathcal{L} \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\pi(s_i)} \nabla_{\theta^\pi}\pi(s|\theta^\pi)|s_i.$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\pi'} \leftarrow \tau\theta^\pi + (1-\tau)\theta^{\pi'}.$$

  end for
end for

Chapter 3

Proposed Approach

In this chapter we explain the approach we are proposing to address the control of urban traffic lights, as described in chapter 1, along with the rationale that led to it.

We begin with section 3.1 by defining which information shall be used as input to our algorithm among all the data that is available from our simulation environment. We proceed by choosing a problem representation for such information to be fed into our algorithm in section 3.4 for the traffic state and section 3.6 for the rewards. Note that the contents of this chapter rely on the concepts about traffic simulation and control described in section 1.2.

3.1 Input Information

The fact that we are using a simulation system to evaluate the performance of our proposed application of deep learning to traffic control, makes the traffic situation fully observable to us. However, in order for our system to be applied to the real world, it must be possible for our input information to be derived from data that is available in a typical urban traffic setup.

The most remarkable examples of readily available data are the ones sourced by **traffic detectors**. They are sensors located throughout the traffic network that provide measurements about the traffic passing through them. Although there are different types of traffic detectors, the most usual ones are induction loops placed under the pavement that send real time information about the vehicles going over them. The information that can normally be taken from such type of detectors is:

- Vehicle count, that is, the number of vehicles that went over the detector during the sampling period.
- Vehicle average speed during the sampling period.
- Occupancy, that is, the percentage of time in which there was a vehicle located over the detector. This is especially useful to detect congestion situations.

This way, we shall constrain the input information that we receive as input about the state of the network to the vehicle counts, average speed and occupancy of every detector in our traffic networks, along with the complete description of the network itself, comprising the location of all roads, their connections, etc.

3.2 Congestion Measurement

Following the self-imposed constraint to use only data that is actually available in a real scenario, we shall elaborate a summary of the state of the traffic based on vehicle counts, average speeds and occupancy. This way, we defined a measured called **speed score**, that is defined for detector i as:

$$speed\_score_i = \min\left(\frac{avg\_speed_i}{max\_speed_i}, 1.0\right) \tag{3.1}$$

where avg_speed_i refers to the average of the speeds measured by traffic detector i and max_speed_i refers to the maximum speed in the road where detector i is located. Note that the speed score hence ranges in $[0, 1]$. This measure will be the base to elaborate the representation of both the state of the environment (section 3.4) and the rewards for our reinforcement learning algorithm (section 3.6).

3.3 Data Aggregation Period

The microscopic traffic simulator used for our experiments (see section A.1) divides the simulation into steps. At each step, a small fixed amount of time is simulated and the state of the vehicles (e.g. position, speed, acceleration) is updated according to the dynamics of the system. This amount of time is configured to be 0.75 seconds by default, and we have kept this parameter.

However, such an amount of time is too short to imply a change in the vehicle counts of the detectors. Therefore, it is needed to have a larger period over which the data is aggregated; we refer to this period as ***episode step***, or simply "step" when there is no risk of confusion. This way, the data is collected at each simulation step and then it is aggregated every episode step for the DDPG algorithm to receive it as input. In order to properly combine the speed scores of several simulation steps, we take their weighted average, using the proportion of vehicle counts. In an analogous way, the traffic light timings generated by the DDPG algorithm are used during the following episode step.

The duration of the episode step was chosen by means of grid search, determining an optimum value of 120 seconds.

3.4 State Representation

In order to keep a state vector of the environment, we make direct use of the speed score described in section 3.2, as it not only summarizes properly the congestion of the network, but also incorporates the notion of maximum speed of each road. This way, the state vector has one component per detector, each one defined as shown in (3.2).

$$state_i = speed\_score_i \tag{3.2}$$

The rationale for choosing the speed score is that, the higher the speed score, the higher the speed of the vehicles relative to the maximum speed of the road, and hence the higher the traffic flow.

3.5 Actions

In the real world there are several instruments to dynamically regulate traffic: traffic lights, police agents, traffic information displays, temporal traffic signs (e.g. to block a road where there is an accident), etc. Although it is possible to apply many of these alternatives in traffic simulation software, we opted to keep the problem at a manageable level and constrain the actions to be applied only to **traffic lights**.

The naive approach would be to let our agent simply control the traffic lights directly by setting the color of each traffic light individually at every simulation step, that is,

the actions generated by our agent would be a list with the color (red, green or yellow) for each traffic light. However, traffic lights in an intersection are synchronized: when one of the traffic lights of the intersection is green, the traffic in the perpendicular direction is forbidden by setting the traffic lights of such a direction to red. This allows to *multiplex* the usage of the intersection. Therefore, letting our agent freely control the colors of the traffic lights would probably lead to chaotic situations.

In order to avoid that, we should keep the *phases* of the traffic lights in each intersection. With that premise, we shall only control the *phase duration*, hence the dynamics are kept the same, only being accelerated or decelerated. This way, if the network has N different phases, the action vector has N components, each of them being a real number that has a scaling effect on the duration of the phase.

However, for each intersection, the total duration of the cycle (i.e. the sum of all phases in the intersection) should be kept unchanged. This is important because in most cases, the cycles of nearby intersections are synchronized so that vehicles travelling from one intersection to the other can catch the proper phase, thus improving the traffic flow. In order to ensure that the intersection cycle is kept, the scaling factor of the phases from the same intersection are passed through a softmax function (also known as normalized exponential function). The result is the ratio of the phase duration over the total cycle duration. In order to ensure a minimum phase duration, the scaling factor is only applied to 80% of the duration.

3.6 Rewards

The role of the rewards is to provide feedback to the reinforcement learning algorithm about the performance of the actions taken previously. As commented in previous section, it would be possible for us to define a reward scheme that makes use of information about the travel times of the vehicles. Some examples described in the literature are summarized in table 5.1, and include the total number of vehicles waiting to enter each intersection, waiting queue lengths, total delay, etc.

However, as we are self-constraining to the information that is available in real world scenarios, we can not rely on other measures apart from detector data, e.g. vehicle counts, speeds. This way, we shall use the speed score described in section 3.2. But the speed score alone does not tell whether the actions taken by our agent actually

improve the situation or make it worse. Therefore, in order to capture such information, we shall introduce the concept of ***baseline***, defined as the speed score for a detector during a hypothetical simulation that is exactly like the one under evaluation but with no intervention by the agent, recorded at the same time step. This way, our reward is the difference between the speed score and the baseline, scaled by the vehicle counts passing through each detector (in order to give more weight to scores where the number of vehicles is higher), and further scaled by a factor α to keep the reward in a narrow range, as shown in (3.3).

$$\begin{aligned} reward_i &= \alpha \cdot count_i \cdot (speed\_score_i - baseline_i) \\ &= \alpha \cdot count_i \cdot \left[\min\left(\frac{avg\_speed_i}{max\_speed_i}, 1.0 \right) - baseline_i \right] \end{aligned} \tag{3.3}$$

Note that we may want to normalize the weights by dividing by the total vehicles traversing all the detectors. This would restrain the rewards in the range $[-1, +1]$. This, however, would make the rewards obtained in different simulation steps not comparable (i.e. a lower total number of vehicles in the simulation at instant t would lead to higher rewards). The factor α was chosen to be $1/50$ empirically, by observing the unscaled values of different networks and choosing a value in an order of magnitude that leaves the scaled value around 1.0. This is important in order to control the scale of the resulting gradients. Another alternative used in [60, 61] with this very purpose is reward clipping; this, however, implies losing information about the scale of the rewards. Therefore, we chose to apply a proper scaling instead.

There is a reward computed for each detector at each simulation time step. Such rewards are not combined in any way, but are all used for the DDPG optimization, as described in section 3.7.2.

Given the stochastic nature of the microsimulator used (A.1), the results obtained depend on the random seed set for the simulation. This way, when computing the reward, the baseline is taken from a simulation with the same seed as the one under evaluation.

3.7 Deep Network

3.7.1 Architecture

Our neural architecture consists in a Deep Deterministic Actor-Critic Policy Gradient approach (as described in section 2.3). The architecture is comprised of two networks: the actor network and the critic network (referred to as π and Q in algorithm 3).

The **actor network** receives the current state of the simulation (as described in section 3.4) and outputs the actions, as described in 3.5. As shown in figure 3.1, the network is comprised of several layers. It starts with several fully connected layers (also known as *dense* layers) with Leaky ReLU activations. Across those many layers, the width of the network increases and then decreases, up to having as many units as actions, that is, the last mentioned dense layer has as many units as traffic light phases in the network. At that point, we introduce a batch normalization layer and another fully connected layer with ReLU activation. The output of the last mentioned layer are real numbers in the range $[0, +\infty]$, so we should apply some kind of transformation that allows us to use them as scaling factors for the phase durations (e.g. clipping to the range $[0.2, 3.0]$). However, as mentioned in section 3.5, we want to keep the traffic light cycles constant. Therefore, we shall apply an element-wise scaling computed on the summation of the actions of the phases in the same traffic light cycle, that is, for each *scaling factory* we divide by the sum of all the factors for phases belonging to the same group (hence obtaining the new ratios of each phase over the cycle duration) and then multiply by the original duration of the cycle. In order to keep a minimum duration for each phase, such computation is only applied to the 80% of the duration of the cycle. Such a computation can be pre-calculated into a matrix, which we call the *phase adjustment matrix*, which is applied in the layer labeled as "Phase adjustment" in figure 3.1, and which finally gives the scaling factors to be applied to phase durations. This careful scaling meant to keep the total cycle duration can be ruined by the exploration component of the algorithm, as described in 3, which consists of adding noise to the actions (and therefore likely breaking the total cycle duration), This way, we implement the injection of noise as another layer prior to the phase adjustment.

The **critic network** receives the current state of the simulation plus the action generated by the actor, and outputs the Q-values associated to them. Like the actor, it is comprised of several fully connected layers with leaky ReLU activations, plus a final dense layer with linear activation.

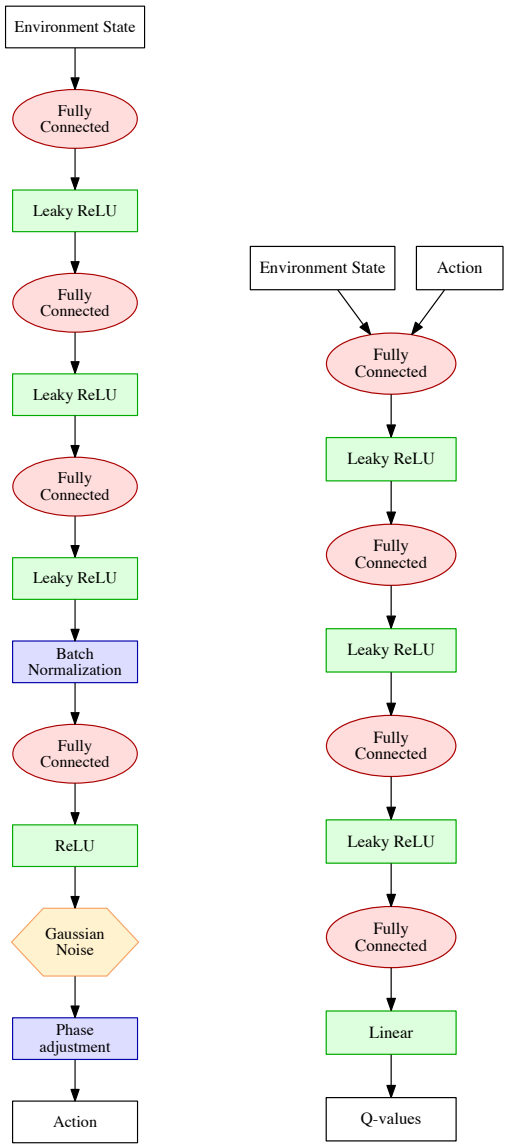

Figure 3.1 Actor (left) and critic (right) networks of our basic architecture

3.7.2 Leverage of the reward vector

In our reference article [57], as well as all landmark ones like [60] and [61], the reward is a single scalar value. However, in our case we build a reward value for each detector in the network.

One option to use such a vector of rewards could be to *scalarize* them into a single value. This, however, would imply losing valuable information regarding the location of the effects of the actions taken by the actor.

We will leverage the structure of the DDPG algorithm, which climbs in the direction of the gradient of the critic. This is partially analogous to a regression problem on the Q-value and hence does not impose a specific structure on the rewards apart from the ability to compute the loss over the generated values. This way, we will have a N-dimensional reward vector, where N is the number of detectors in the network. This extends the policy gradient theorem from [76] so that the reward function is no longer defined as $r : S \times A \rightarrow \mathbb{R}$ but as $r : S \times A \rightarrow \mathbb{R}^N$. This is analogous to having N agents sharing the same actor and critic networks (i.e. sharing weights θ^π and θ^Q) and being trained simultaneously over N different unidimensional reward functions. This, effectively, implements multiobjective reinforcement learning.

Such an approach could be further refined by weighting rewards according to traffic control expert knowledge, which will then be incorporated in the computation of the policy gradients.

To the best of our knowledge, the use of ***disaggregated rewards*** **has not been used before** in the reinforcement learning literature. Despite having proved useful in our experiments, further study is needed in order to fully characterize the effect of disaggregated rewards on benchmark problems. This is one of the future lines of research that can be spawned from this work.

3.7.3 Convergence

In this section we study different aspects of the tuning of the algorithm that were key to the convergence of the learning process.

Weight Initialization

Weight initialization has been a key issue in the results cast by deep learning algorithms. The early architectures could only achieve acceptable results if they were pre-trained by means of unsupervised learning so that they could have *learned* the input data structure [31]. The use of sigmoid or hyperbolic tangent activations makes it difficult to

optimize neural networks due to the numerous local minima in the function loss defined over the parameter space. With pre-training, the exploration of the parameter space does not begin in a random point, but in a point that *hopefully* is not too far from a good local minimum. Pretraining became no longer necessary to achieve convergence thanks to the use of rectified linear activation units (ReLUs) [62], residual learning [43, 45] and sensible weight initialization strategies.

In our case, different random weight initializations (i.e. Glorot's [37] and He's [44]) gave the best results, finally selecting He's approach.

Updates to the Critic

After our first experiments it became evident the divergence of the learning of the network. Careful inspection of the algorithm byproducts revealed that the cause of the divergence was that the critic network Q' predicted higher outcomes at every iteration, as trained according to equation (3.4) extracted from algorithm 3.

$$y_i = r_i + \gamma Q'(s_{i+1}, \pi'(s_i + 1|\theta^{\pi'})|\theta^{Q'}) \tag{3.4}$$

As DDPG learning -like any other reinforcement learning with value function approximation approach- is a closed loop system in which the target value at step $t+1$ is biased by the training at steps t, drifts can be amplified, thus ruining the learning, as the distance between the desired value for Q and the obtained one differ more and more. This is shown in figure 3.2 with the loss function of the critic network Q (i.e. the mean squared error) increasing instead of decreasing.

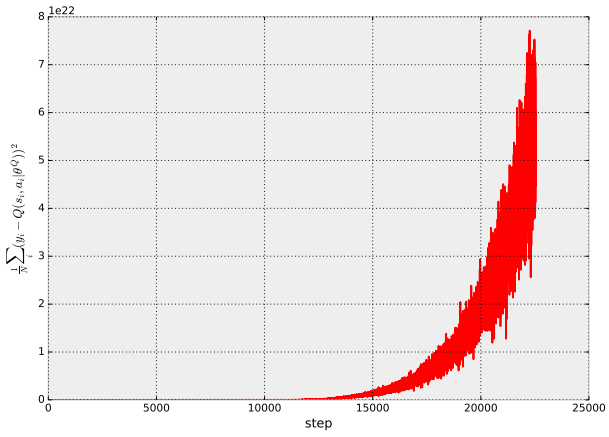

Figure 3.2 Loss function (MSE) of a diverging Q network

In order to mitigate the aforedescribed divergence problem, our proposal consists in reducing the coupling by means of the application of a schedule on the value of the discount factor γ from Bellman's equation (2.8), which is shown in figure 3.3.

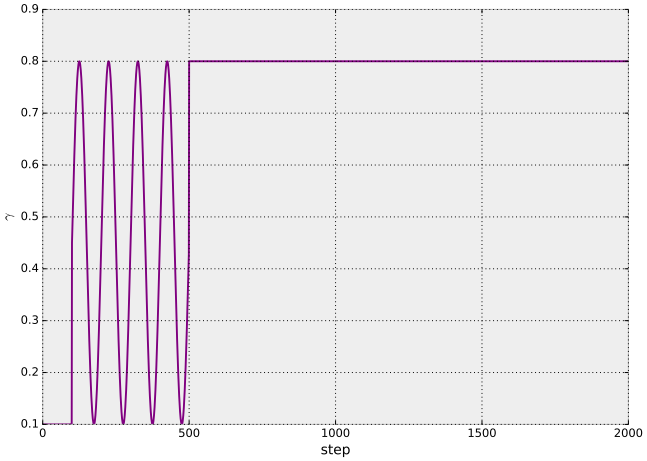

Figure 3.3 Schedule for the discount factor γ.

The schedule of γ is applied at the level of the experiment, not within the episode. The oscillation in γ shown in figure 3.3 is meant to enable the critic network not to enter in the regime where the feedback leads to divergence.

To the best of our knowledge, **discount factor scheduling has never been used before** in the literature to improve the convergence of reinforcement learning with value function approximation. Despite having proved useful in our experiments, further study is needed in order to fully characterize the effect of discount factor schedules on benchmark problems. This is one of the future lines of research that can be spawned from this work.

Gradient evolution

The convergence of the algorithm can be evaluated thanks to the norm of the gradient used to update the actor network π. If such a norm decreases over time and stagnates around a low value, it is a sign that the algorithm has reached a stable point and that the results might not further improve. This way, in the experiments described in subsequent chapters, monitoring of the gradient norm is used to track progress.

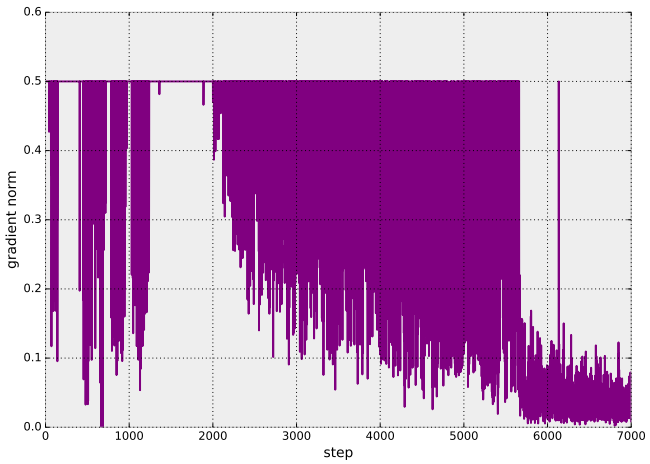

Figure 3.4 Sample of the evolution of the gradient norm.

The gradient norm can also be controlled in order to avoid too large updates that make the algorithm diverge, e.g. [60]. This mechanism is called *gradient norm clipping* and consists of scaling the gradient so that its norm is not over a certain value. Such a value was empirically established as 0.5 in our case. A sample of the gradient norm over one of the experiments is shown in figure 3.4, where the effect of the gradient norm clipping are evident.

3.8 Summary

This is a summary of the approach described throughout this chapter. Our proposal is to apply Deep Deterministic Policy Gradient, as formulated in [57], to the traffic optimization problem by controlling the traffic lights timing. We make use of a multilayer perceptron type of architecture, both for the actor and the critic networks, using leaky ReLU activations (except for the last activations, which are ReLU for the actor and linear for the critic). The actor is designed so that the modifications to the traffic light timings keep the cycle duration. In order to optimize the networks we make use of stochastic gradient descent. In order to improve convergence, we make use of a replay memory, gradient norm clipping and a schedule for the discount rate γ. The input state used to feed the network consists of traffic detector information, namely vehicle counts and average speeds, which are combined in a single *speed score*. The rewards used as reinforcement signal are the improvements over the measurements without any control action being performed (i.e. baseline). Such rewards are not aggregated but fed directly as expected values of the critic network.

Chapter 4

Experiments

In this chapter we describe the experiments conducted in order to evaluate the performance of the approach described in chapter 3. In section 4.1 we show the different traffic scenarios used while in section 4.2 we describe the results obtained in each one, along with lessons learned from the problems found, plus hints for future research.

4.1 Design of the Experiments

In order to evaluate our deep RL algorithm, we devised increasing complexity traffic networks. In the following sections we describe the characteristics of each of them.

For each network, we applied our **DDPG** algorithm to control the traffic light timing, but also applied a classical **Q-Learning** and **random timing** in order to have a reference to properly assess the performance of our approach.

At each experiment, the DDPG algorithm receives as input the information of all detectors in the network, and generates the timings of all traffic light phases.

The **Q-learning** agent only manages one intersection phase. It receives the information from the closest few detectors and generates the timings for the aforementioned phase. Given the tabular nature of Q-learning, both the state space and the action space need to be categorical. For this, we use tile coding, as described in section 2.2.2. Regarding the state space, the tiles are defined based on the same state space values as DDPG (see section 3.4), clustered in one the following 4 ranges $[-1.0, -0.2]$, $[-0.2, -0.001]$, $[-0.001, 0.02]$, $[0.02, 1.0]$, which were chosen empirically. As one Q-learning agent controls the N_i phases of the traffic lights of an intersection i, the number of states

for an agent is 4^{N_i}. The action space is analogous, being the generated timings one of the values 0.2, 0.5, 1.0, 2.0 or 3.5. The selected ratio (i.e. ratio over the original phase duration) is applied to the duration of the phase controlled by the Q-learning agent. As there is one agent per phase, this is a multi-agent reinforcement learning setup, where agents do not communicate with each other. They do have overlapping inputs, though, as the data from a detector can be fed to the agents of several phases. In order to keep the cycle times constant, we apply the same phase adjustment used for the DDPG agent, described in section 3.5.

The **random** agent generates random timings in the range $[0, 1]$, and then the previously mentioned phase adjustment is applied to keep the cycle durations constant (see section 3.5).
Given the **stochastic** nature of the microscopic traffic simulator used (A.1), the results obtained at the experiments depend on the random seed set for the simulation. In order to address the implications of this, we do as follows:

- In order for the algorithms not to overfit to the dynamics of a single simulation, we randomize the seed of each simulation. We take into account this also for the computation of the baseline, as described in section 3.6.
- We repeat the experiments several times, and present the results over all of them (showing the average, maximum or minimum data depending on the case).

4.1.1 Network A

This network, shown in figure 4.1 consists only of an intersection of two 2-lane roads. At the intersection vehicles can either go straight or turn to their right. It is forbidden to turn left, therefore simplifying the traffic dynamics and traffic light phases. There are 8 detectors (in each road there is one detector before the intersection and another one after it).

There are two phases in the traffic light group: phase 1 allows horizontal traffic while phase 2 allows vertical circulation. Phase 1 lasts 15 seconds and phase 2 lasts 70 seconds, with a 5-seconds inter-phase. Phases 1 and 2 have unbalanced duration on purpose, to have the horizontal road accumulate vehicles for long time. This gives our algorithm room to easily improve the traffic flow with phase duration changes. The simulation comprises 1 hour and the vehicle demand is constant: for each pair of centroids, there are 150 vehicles.

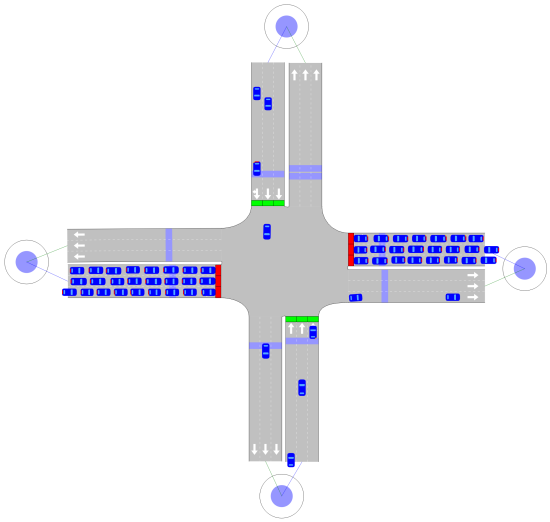

Figure 4.1 Network A

4.1.2 Network B

This network, shown in figure 4.2 consists of a grid layout of 3 vertical roads and 2 horizontal ones, crossing in 6 intersections that all have traffic lights.

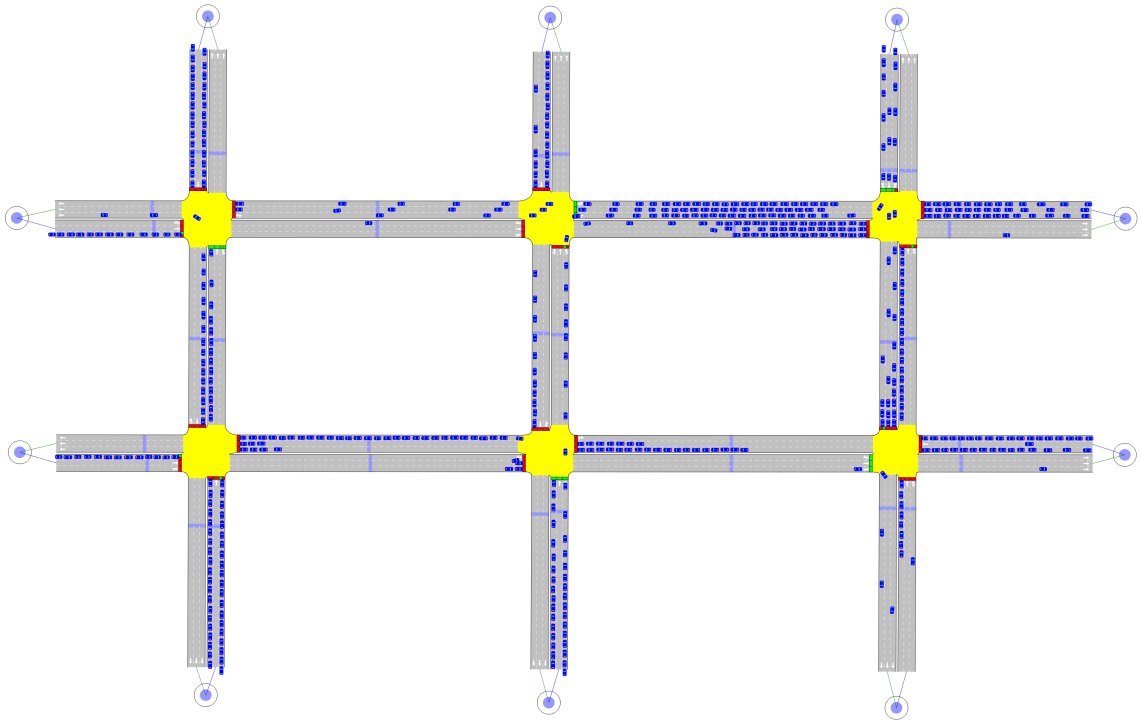

Figure 4.2 Network B

Traffic in an intersection can either go straight, left or right, that is, all turns are allowed, complicating the traffic light phases, which have been generated algorithmically by the software with the optimal timing, totalling 30 phases. There are detectors before and after each intersection, totalling 17 detectors.

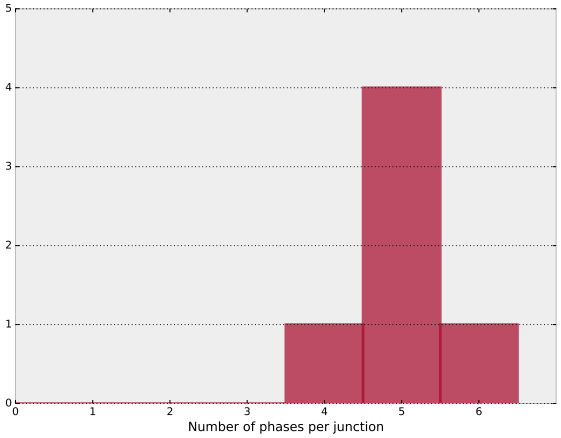

Figure 4.3 Histogram of number of phases per junction in network B

As shown in figure 4.3, 4 out of 6 junctions have 5 phases, while the remaining two junctions have 4 and 6 phases each.

The traffic demand has been created in a random manner, but ensuring enough vehicles are present and trying to collapse some of the sections of the network.

4.1.3 Network C

This network, shown in figure 4.4 is a replica of the Sants area in the city of Barcelona (Spain). There are 43 junctions, totalling 102 traffic light phases, and 29 traffic detectors. The locations of the detectors matches the real world and they are highlighted in figure 4.7. The traffic demand matches that of the peak hour in Barcelona, and it presents high degree of congestion.

As shown in figure 4.5, the number of controlled phases per junction [1] ranges from 1 to 6, having most of them only two phases.

[1]Note that phases from the network that have a very small duration (i.e. 2 seconds or less) are excluded from the control of the agent

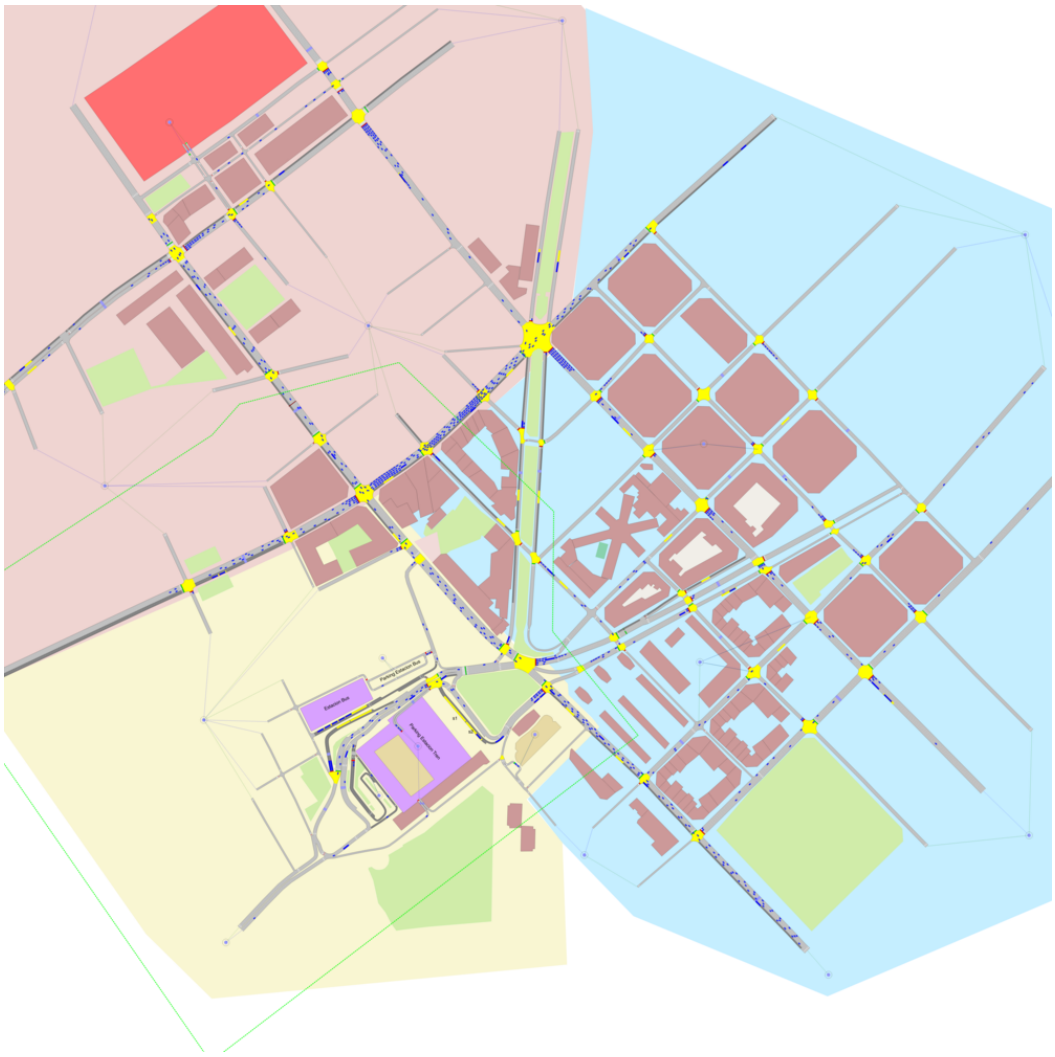

Figure 4.4 Network C

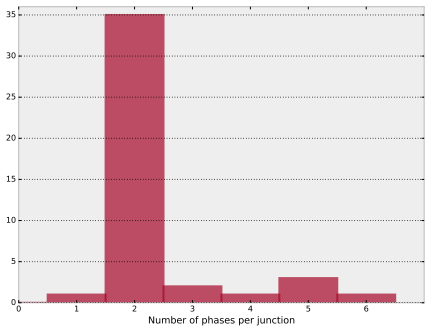

Figure 4.5 Histogram of number of phases per junction in network C

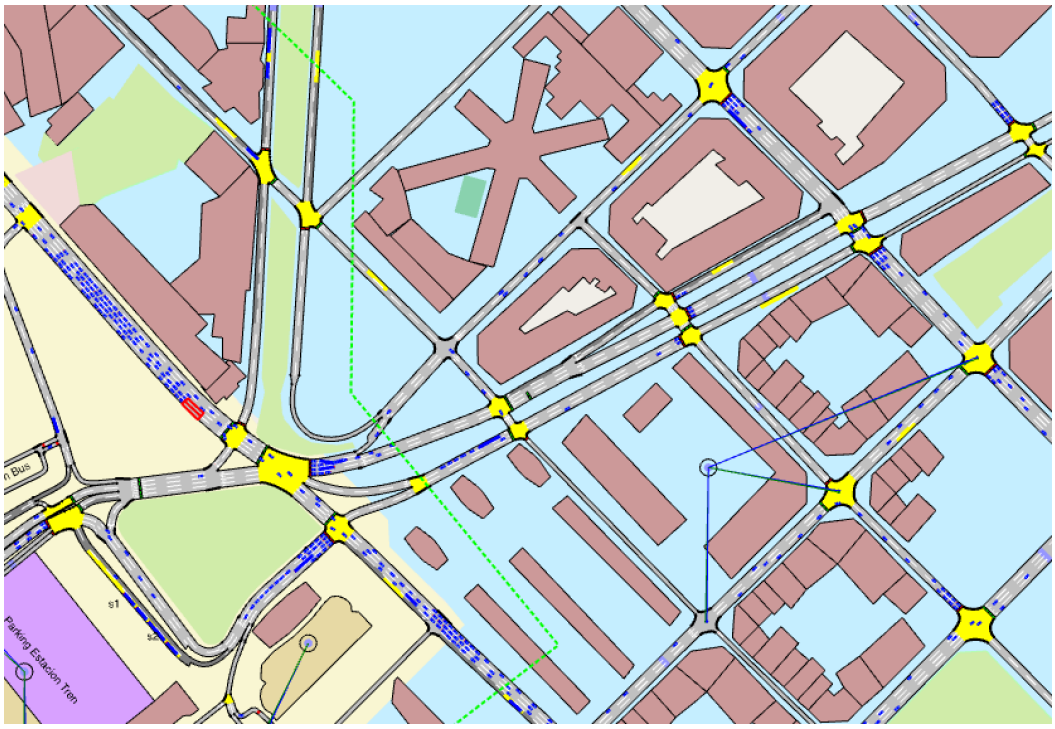

Figure 4.6 Network C (detail)

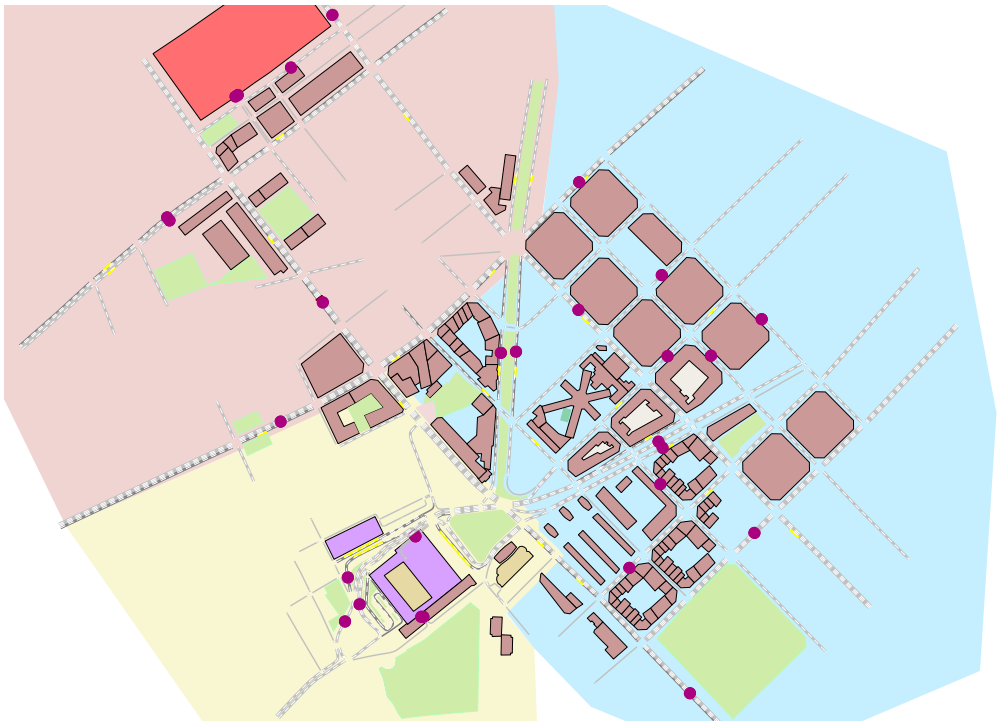

Figure 4.7 Traffic detectors in Network C

4.2 Results

In order to evaluate the performance of our DDPG approach compared to both normal Q-learning and random timings on each of our test networks, our main reference measure shall be the episode average reward (note that, as described in section 3.6 there is actually a vector of rewards, with one element per detector in the network, that is why we compute the average reward) of the *best* experiment trial, understanding "best" experiment as the one where the maximum episode average reward was obtained.

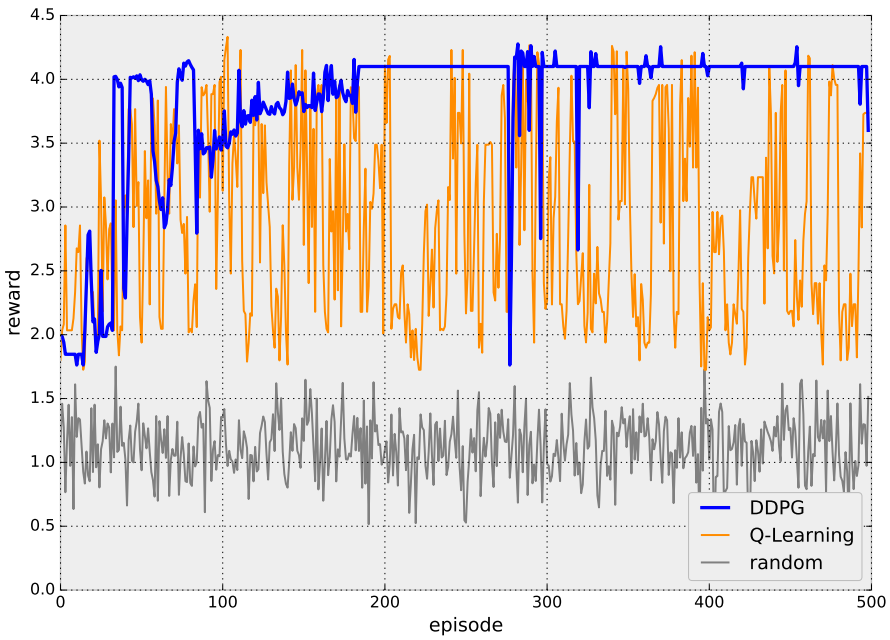

Figure 4.8 Algorithm performance comparison on network A

In figure 4.8 we can find the performance comparison for network A. Both the DDPG approach and the classical Q-learning reach the same levels of reward. On the other hand, it is noticeable the differences in the convergence of both approaches: while Q-learning is unstable, DDPG remains remarkably stable once it reached its peak performance.

We can further explore the behaviour of the algorithm by studying the intra-episode performance: in figure 4.9 it is shown the performance of the first episode of the DDPG algorithm and its performance at its best episode; the performances are shown as step average reward with min-max bands (the average and the minimum and maximum

are computed over all trials executed for the same experiment). We can see that the improvements over the baseline are not constant throughout the episode, but are prominent by the end.

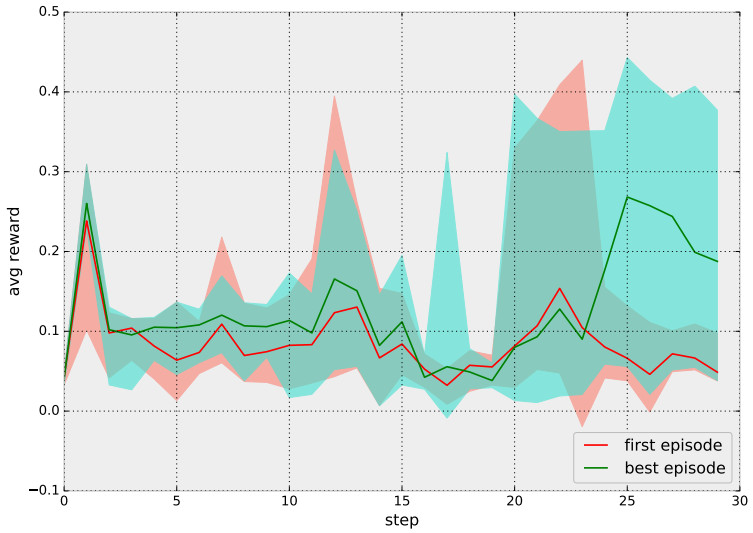

Figure 4.9 Intra-episode evolution of DDPG algorithm on network A

The same pattern can be seen in the intra-episode performance of the Q-learning algorithm shown in figure 4.10.

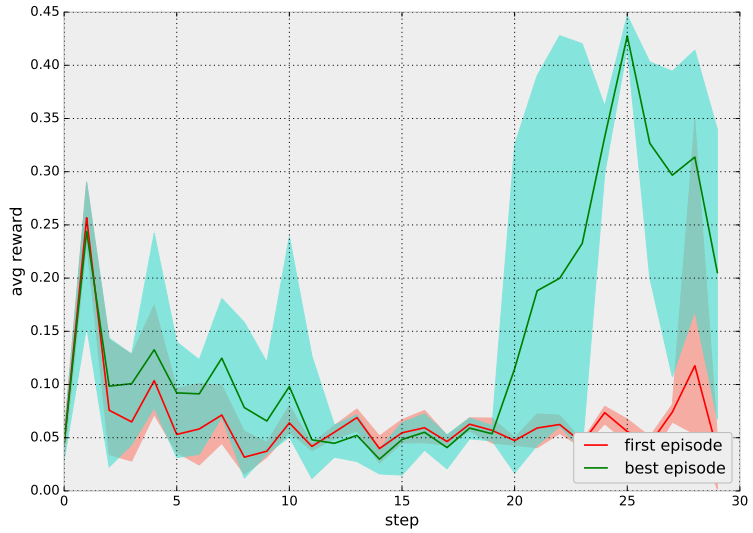

Figure 4.10 Intra-episode evolution of Q-learning algorithm on network A

In figure 4.11 we can find the performance comparison for network B. While Q-learning maintains the same band of variations along the simulations, DDPG starts to converge.

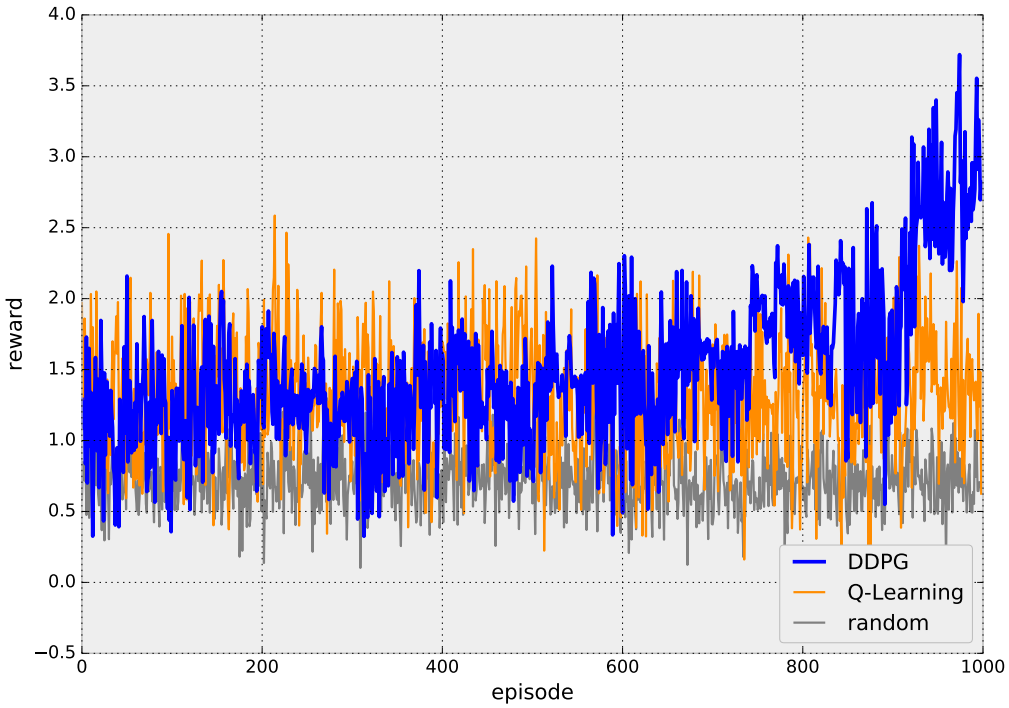

Figure 4.11 Algorithm performance comparison on network B

Given the great computational costs of running the full set of simulations for one network, it is simply not affordable to let it run indefinitely, despite the promising trend. For instance, simulating and episode plus training the algorithm takes 2 minutes; we run episodes of 1000 steps; we run 5 trials for each of the 3 algorithms (DDPG, Q-learning and random), which accounts for a total of more than 20 days of computing time (2 mins/step x 1000 steps/episode x 5 episodes/algorithm x 3 algorithms).

In figures 4.12 and 4.13 we can appreciate the different ways in which DDPG and Q-learning respectively acted during the episodes.

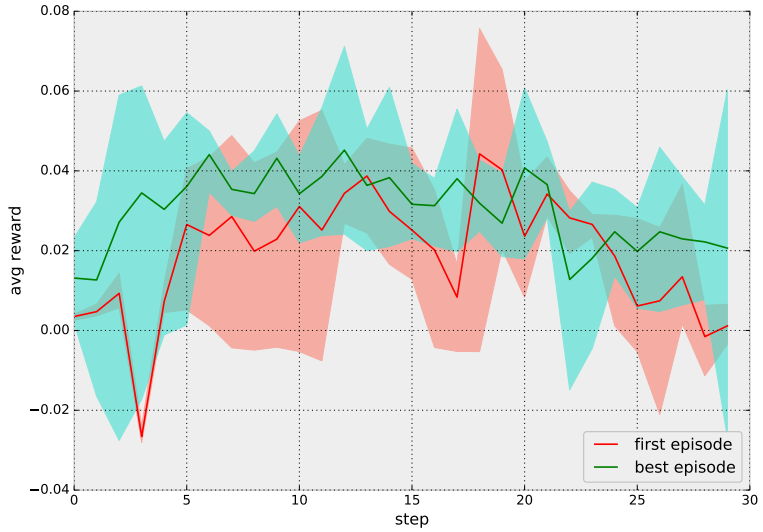

Figure 4.12 Intra-episode evolution of DDPG algorithm on network B

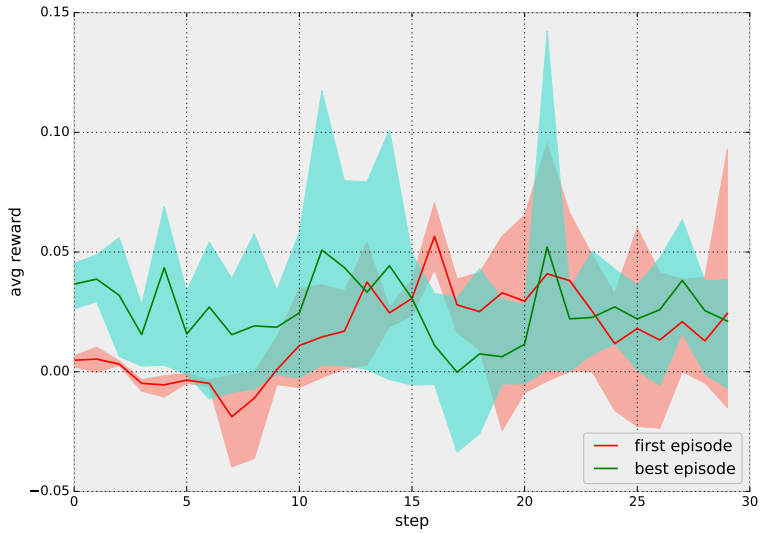

Figure 4.13 Intra-episode evolution of Q-learning algorithm on network B

Figure 4.14 shows the performance comparison for network C, from which we can appreciate that both DDPG and Q-learning performs at the same level, and that such a level is beneath zero, from which we know that they are actually worse than doing nothing.

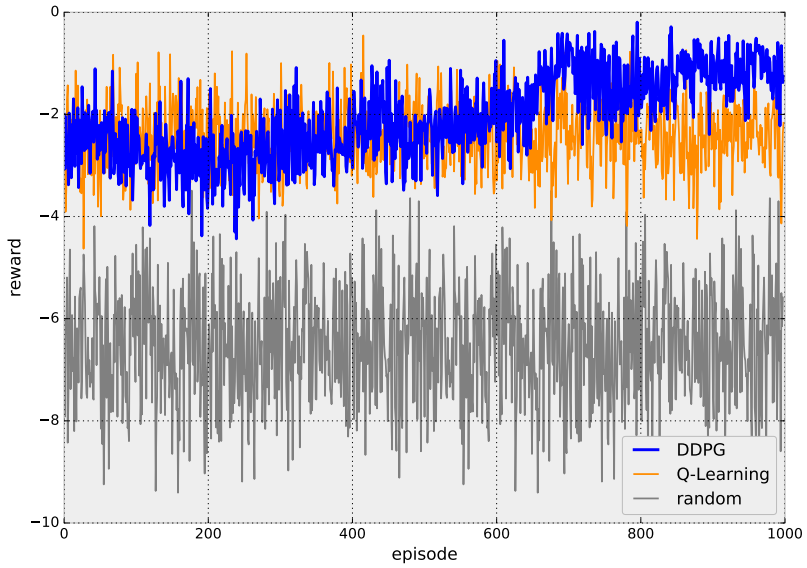

Figure 4.14 Algorithm performance comparison on network C

This way, the performance of DDPG is clearly superior to Q-learning for the simplest scenario (network A), slightly better for scenarios with a few intersections (network B) and at the same level for real world networks.

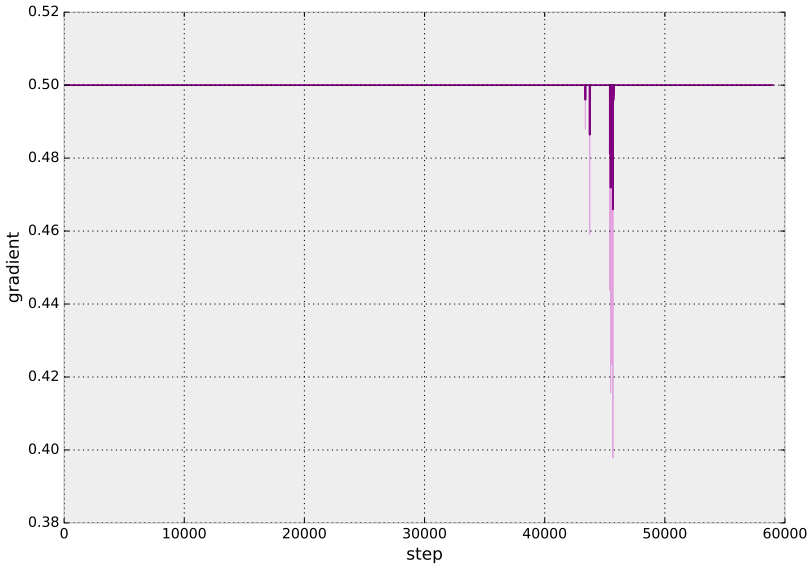

Figure 4.15 Evolution of the gradient norm in the best experiment on network B.

From the evolution of the gradient for medium and large networks shown in figure 4.15, we know that convergence is not achieved, as it remains always at the maximum value induced by the gradient norm clipping. This suggests that the algorithm needs more training time to converge (probable for network B) or that it diverges (probable for network C). In any case, further study would be needed in order to assess the needed training times and the needed convergence improvement techniques.

Chapter 5

Related Work

In this chapter we identify and explore research lines that are especially close to ours, either regarding the techniques used, or regarding the problem they address.

5.1 Classic Reinforcement Learning

Reinforcement Learning has been applied in the past to urban traffic light control. Most of the instances from the literature consist of a classical algorithm like Q-Learning, SARSA or TD(λ) to control the timing of a single intersection. Rewards are typically based on the reduction of the travel time of the vehicles or the queue lengths at the traffic lights. Table 5.1 (reproduced from [30]) shows a summary of the different approaches followed by a subset of articles from the literature that apply reinforcement learning to traffic light timing control. As shown there, many studies use as state space information such as the length of the queues and the travel time delay; such type of measures are rarely available in a real-world setup and can therefore only be obtained in a simulated environment. Most of the approaches use discrete actions (or alternatively, discretize the continuous actions by means of tile coding (see section 2.2.2), and use either ε-greedy selection (choose the action with highest Q value with $1 - \varepsilon$ probability, or random action otherwise) or softmax selection (turn Q values into probabilities by means of the softmax function and then choose stochastically accordingly). In most of the applications of reinforcement learning to traffic control, the validation scenario consists of a single intersection, like in [87]. This is due to the scalability problems of classical RL tabular approaches: as the number of controlled intersections increases, so grows the state space, making the learning unfeasible due to the impossibility for the agent to apply every action under every possible state.

Study	RL method	State space	Reward/Penalty	Action
[87]	SARSA(λ)	Vehicle counts in the links leading to intersection	(Penalty) Time required to release a fixed volume of traffic through a road network	ε-greedy
[98]	Model-based Q-learning	Number and location of vehicles in the links leading to intersection	(Penalty) Total delay incurred between successive decision points	ε-greedy
[3]	Q-learning	Queue lengths in the links leading to intersection	(Penalty) Total delay incurred between successive decision points	softmax
[18]	Q-learning	Number and location of vehicles on the links leading to intersection	(Penalty) Number of vehicles waiting at intersection	ε-greedy
[25]	Similar to Q-Learning	Vehicle counts in the links leading to intersection	(Penalty) Squared sum of the incoming links queues	ε-greedy
[69]	Modified Q-Learning	Current cycle length, current phase durations, detectors status	(Penalty) Number of cars that entered intersection over the last time step	softmax
[75]	Q-learning	Total delay of the intersection	Total delay of the intersection	ε-greedy
[72]	Modified Q-Learning	Vehicle counts in the links leading to intersection	(Reward) Number of vehicles getting out of the junction minus number of vehicles still waiting	softmax
[7]	Q-learning	Change in the total queue length in the last time step	(Penalty) Change in total queue length in last time step	ε-greedy
[5]	Q-Learning	Relative delay at each lane leading to intersection	(Reward) Savings in the delay	ε-greedy

Table 5.1 Survey of RL methods applied to traffic control (reproduced from [30]).

This led some researchers to study multi-agent approaches, with varying degrees of complexity: some approaches like that from [5] train each agent separately, without notion that more agents even exist, despite the coordination problems that this approach poses. Others like [98] train each agent separately, but only the intersection with maximum reward executes the action. More elaborated approaches, like in [18], train several agents together modeling their interaction as a competitive stochastic game. Alternatively, some lines of research like [55] and [6] study cooperative interaction of agents by means of coordination mechanisms, like coordination graphs ([41]).

As described throughout this section, there are several examples in the literature of the application of classical reinforcement learning to traffic light control. Many of them focus on a single intersection. Others apply multi-agent reinforcement learning techniques to address the problems derived from the high dimensionality of state and action spaces. Two characteristics of most of the explored approaches are that the information used to elaborate the state space is hardly available in a real-world environment and that there are no realistic testing environments used.

5.2 Deep Reinforcement Learning

There are some recent works that, like ours, study the applicability of **deep** reinforcement learning to traffic light controls. In this section we explore in detail their specific approaches, identifying similarities with our work plus strengths and weaknesses in each other:

- Li et al. studied in [56] the application of deep learning to traffic light timing in a single intersection.

  Their testing setup consists of a single cross-shape intersection with two lanes per direction, where no turns are allowed at all (i.e. all traffic either flows North-South (and South-North) or East-West (and West-East), hence the traffic light set only has two phases. This scenario is therefore simpler than our simple network A presented in 4.1.1. For the traffic simulation, they use the proprietary software PARAllel MICroscopic Simulation (PARAMICS) [17], which implements the Cell Transmission Model. This model does not take into account individual vehicles. Instead, it divides the road into cells and computes the flow of vehicles moving inside each cell. Therefore, it implements a very simplified version of the traffic dynamics.

Their approach consists of a Deep Q-Network (as described in section 2.3) comprised of a heap of stacked auto-encoders [12, 93], with sigmoid activation functions where the input is the state of the network and the output is the Q function value for each action.

The inputs to the deep Q network are the queue lengths of each lane at time t (measured in meters), totalling 8 inputs. The actions generated by the network are 2: remain in the current phase or switch to the other one. The reward is the absolute value of the difference between the maximum North-Source flow and the maximum East-West flow.

The stacked autoencoders are pre-trained (i.e. trained using the state of the traffic as both input and output) layer-wise so that an internal representation of the traffic state is learned, which should improve the stability of the learning in further fine tuning to obtain the Q function as output ([8, 31]. The authors use an experience-replay memory to improve learning convergence.

In order to balance exploration and exploitation, the authors use an ϵ-greedy policy, choosing a random action with a small probability p.

For evaluating the performance of the algorithm, the authors compare it with normal Q-learning (as described in section 2.2.2). For each algorithm, they show the queue lengths over time and perform a linear regression plot on the queue lengths for each direction (in order to check the *balance* of their queue length).

The work in [56] compares with our present work in the following ways:

- The actions are discrete, which enables them to use Deep Q-learning. In our case, the actions are continuous, so we use Deep Deterministic Policy Gradient.
- The input and action spaces are small (8 inputs, 2 actions), and the test setup is not realistic (e.g. no turns are allowed at all). In our case, the input and action spaces of the most complex of our examples are two orders of magnitude larger, and the testing setups range from simple (but realistic) to complete cities.
- The reward function reflect the balance between the flow in horizontal and vertical directions at every sampling time step. This presents scalability problems and introduces the assumption that in the flows in both directions should be similar, which does not hold true in real world scenarios. Our

reward function directly evaluates the lack of congestion, which scales well and is well suited for real environments.

  - The the network is a stacked auto-encoder with sigmoid activations and pre-training. This architecture is known to suffer from the vanishing gradients problem. Our network uses ReLU as basic activation, therefore avoiding the vanishing gradients and making pre-training unnecessary.
  - The input space of the network consists of the queue lengths, which prevents it to be applied to real world scenarios, in which such data is seldom available.
  - The simulation software loosely reflects the dynamics of real traffic, given the coarse grained approach of the cell transmission model. In our case, we use a full microscopic simulation, where individual vehicles are modeled.
  - The evaluation of the performance of the algorithm only offers weak evidence of the improvements of the algorithm over other alternatives. There is no comparison with the *baseline* behaviour where there is no algorithm controlling the traffic light timing but only a fixed control program or actuated traffic lights. Our evaluation of the performance is directly done against the baseline behaviour and we also compare different algorithms.

- Van der Pol explores in their Master Thesis [90] the application of deep learning to traffic light coordination, both in a single intersection and in a more complex configuration.

  Their testing setup consists of a single cross-shaped intersection with one lane per direction, where no turns are allowed. For the simulation software, the author uses SUMO (Simulation of Urban MObility), a popular open-source simulator that simulates vehicles individually, modeling the dynamics described by the Krauß model (further details can be found in section A.1 where we explore simulator candidates for the present work). Given that SUMO *teleports* vehicles that have been stuck for a long time, the author needs to take this into account in the reward function, in order to penalize traffic light configurations that favour vehicle teleportation.

  Their approach consists on a Deep Q-Network. The author experiments with two two alternative architectures, taken verbatim respectively from [60] and [61]. Those convolutional networks were meant to play Atari games and receive as input the pixel matrix with bare preprocessing (downscaling and graying). In order to enable those architectures to be fed with the traffic data as input, an

image is created by plotting a point on the location of each vehicle. The action space is comprised of the different legal traffic light configurations (i.e. those that do not lead to flow conflicts), among which the network chooses which to apply. The reward is a weighted sum of several factors: vehicle delay (defined as the road maximum speed minus the vehicle speed, divided by the road maximum speed), vehicle waiting time, the number of times the vehicle stops, the number of times the traffic light switches and the number of *teleportations*.

In order to improve convergence of the algorithm, the authors apply deep reinforcement learning techniques such as prioritized experience replay and keeping a shadow target network, but also experimented with double Q learning [42, 91]. They as well tested different optimization algorithms apart from the normal stochastic gradient optimization, such as the ADAM optimizer [51], Adagrad [28] or RMSProp [89].

The performance of the algorithm is evaluated visually by means of plots of the reward and average travel time during the training phase.

The author also explores the behaviour of their algorithm in a scenario with multiple intersections (up to four) by means of a multi-agent approach. This is achieved by training two neighbouring intersections on their mutual influence and then the learned joint Q function is *transferred* for higher number of intersections.

The work in [90] compares with our present work in the following ways:

- The original traffic information is *converted* into an image so that the convolutional neural networks from [60] and [61] can be used. On the other hand, in our case the raw information is kept in its very nature, and is only aggregated over time. It might be possible for the convolutional approach by Van der Pol to profit from the geometrical information of the urban traffic network, as it has the *spatial view* of the environment [1] (see appendix B for details on our attempts to use information about the connections of the roads in the network) but the author's approach does not actually profit from this, as the algorithms are not trained in realistic scenarios.
- The dynamics of the traffic vehicles are not realistic due to the behaviour of the traffic simulator used, and the author needs to mitigate *teleportations* as part of its algorithm. In our case, the dynamics follow real world behaviour

[1] Despite having the spatial information about the traffic network, given the information lost in the conversion to the pixel matrix, it is probable that the ability of the algorithm to profit from the geometrical information is low.

and therefore we do not need to address any kind of issue as part of our approach.

- The state space and the reward function are comprised of information that is not available in real-world scenarios, and has to deal with the abnormalities of the simulation software (i.e. *teleportations*). In our case, the reward function is fully available in any real world environment.
- The scalability of the algorithm to multiple intersections relies on the ability to coordinate multiple agents, while our approach leverages the ability of deep neural nets to receive huge input spaces to take coordinated decisions in a centralized manner.

- Genders et al. explore in [35] the the application of deep convolutional learning to traffic light timing in a single intersection.

  Their test setup consists of a single cross-shaped intersection with four lanes in each direction, where the inner lane is meant only for turning left and the outer lane is meant only for turning right. As simulation software, the authors use SUMO, like the work by Van der Pol [90] (see previous bullet). However, Genders et al do not address the teleportation problem and do not take into account its effect on the results.

  Their approach consists of a Deep Convolutional Q-Network. Like in [90], Genders et al. transform the vehicle positions into a matrix so that it becomes a suitable input for the convolutional network. They, however, scale the value of the pixels with the local density of vehicles. The authors refer to this representation as discrete traffic state encoding (DTSE). The actions generated by the Q-Network are the different phase configurations of the traffic light set in the intersection. The reward defined as the variation in cumulative vehicle delay since the last action was applied. The network is fed using experience replay.

  The work in [90] compares with our present work in the following ways:

  - The original traffic information is *converted* into an matrix to feed the convolutional network taking advantage of information locality. It would be interesting to explore the learned features, as done in [50] to visualize the weights or in [99] to infer the regions of the image that influenced the result most, to study the traffic patterns used to trigger different actions.

- The dynamics of the traffic vehicles are not realistic due to the vehicle teleportation behaviour of SUMO, and the authors do not take this into account.
- The state space and the reward function are comprised of information that is not available in real-world scenarios, that is, the exact location of each vehicle in the traffic network. In our case, the reward function is fully available in any real world environment.

Chapter 6

Conclusions

We studied the application of Deep Deterministic Policy Gradient (DDPG) to increasingly complex scenarios. We obtained good results in network A, which is analogous to most of the scenarios used to test reinforcement learning applied to traffic light control (see chapter 5 for details on this); nevertheless, for such a small network, vanilla Q-learning performs *on par*, but with less stability, though. However, when the complexity of the network increases, Q-learning can no longer scale, while DDPG still can improve consistently the obtained rewards. With a real world scenario, our DDPG approach is not able to properly control the traffic better than doing nothing. The good trend for network B shown in figure 4.11, suggests that longer training time may lead to better results. This might be also true for network C, but the extremely high computational costs could not be handled without large scale hardware infrastructure.

Our results show that DDPG is able to better scale to larger networks than classical tabular approaches like Q-learning. Therefore, DDPG is able to address the *curse of dimensionality* [38] regarding the traffic light control domain, at least partially. However, it is not clear that the chosen reward scheme (described in section 3.6) is appropriate. One of its many weaknesses is its *fairness* for judging the performance of the algorithm based on the individual detector information. In real life traffic optimization it is common to favour some areas so that traffic flow in arterials or large roads is improved, at the cost of worsening side small roads. The same principle could be applied to engineer a more realistic reward function from the point of view of traffic control theory.

Another aspect that needs further study is the effect of the amount and location of traffic detectors on the performance of the algorithm. In our networks A and B, there were detectors at every section of the network, while in network C their placement was

scattered, which is the norm in real world scenarios. We appreciate a loose relation between the *degree of observability* of the state of the network and the performance of our proposed traffic light timing control algorithm. Further assessment about the influence of observability of the state of the network would help characterize the performance of the DDPG algorithm and even turn it into a means for choosing potential locations for new detector in the real world.

An issue regarding the performance of our approach is the sudden drops in the rewards obtained through the training process. This suggests that the landscape of the reward function with respect to the actor and critic network parameters is very irregular, which leads the optimization to fall into *bad areas* when climbing in the direction of the gradient. A possible future line of research that addressed this problem could be applying Trusted Region Policy Optimization [73], that is, leveraging the simulated nature of our setup to explore more efficiently the solution space. This would allow it to be more data efficient, achieving comparable results with less training.

We have provided two contributions that, to the best of our knowledge, have not been used before in the deep reinforcement learning literature, namely the use of disaggregated rewards (described in section 3.7.2) and the scheduling of the discount factor γ (described in section 3.7.3). These techniques need to be studied in isolation from other factors on benchmark problems in order to properly assess their effect and contribution to the performance of the algorithms. This is another possible line of research to be spawned from this work.

On the other hand, we have failed to profit from the geometric information about the traffic network (see appendix B). This is clearly a possible future line of research, that can leverage recent advances in the application of convolutional networks to arbitrary graphs, similar to [26].

Finally, we have verified the applicability of simple deep learning architectures to the problem of traffic flow optimization by traffic light timing control on small and medium-sized traffic networks. However, for larger-sized networks further study is needed, probably in the lines of exploring the results with significantly larger training times, using the geometric information of the network and devising data efficiency improvements.

Bibliography

[1] Deeplearning4j: Open-source, distributed deep learning for the jvm, 2014 (accessed August 12, 2015). URL http://deeplearning4j.org/.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

[3] Baher Abdulhai, Rob Pringle, and Grigoris J Karakoulas. Reinforcement learning for true adaptive traffic signal control. *Journal of Transportation Engineering*, 129 (3):278–285, 2003.

[4] TSS Aimsun. Dynamic simulators users manual. *Transport Simulation Systems*, 20, 2012.

[5] Itamar Arel, Cong Liu, T Urbanik, and AG Kohls. Reinforcement learning-based multi-agent system for network traffic signal control. *IET Intelligent Transport Systems*, 4(2):128–135, 2010.

[6] Bram Bakker, Shimon Whiteson, Leon Kester, and Frans CA Groen. Traffic light control by multiagent reinforcement learning systems. In *Interactive Collaborative Information Systems*, pages 475–510. Springer, 2010.

[7] PG Balaji, X German, and Dipti Srinivasan. Urban traffic signal control using reinforcement learning agents. *IET Intelligent Transport Systems*, 4(3):177–188, 2010.

[8] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. *ICML unsupervised and transfer learning*, 27(37-50):1, 2012.

[9] Pierre Baldi and Peter J Sadowski. Understanding dropout. In *Advances in Neural Information Processing Systems*, pages 2814–2822, 2013.

[10] Ana LC Bazzan. Opportunities for multiagent systems and multiagent reinforcement learning in traffic control. *Autonomous Agents and Multi-Agent Systems*, 18 (3):342–375, 2009.

[11] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5 (2):157–166, 1994.

[12] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

[13] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[14] Steven Skiena Bryan Perozzi, Rami Al-Rfou. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014.

[15] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, And Cybernetics-Part C: Applications and Reviews, 38 (2), 2008*, 2008.

[16] Chang-Qing Cai and Zhao-Sheng Yang. Study on urban traffic management based on multi-agent system. In *2007 International Conference on Machine Learning and Cybernetics*, volume 1, pages 25–29. IEEE, 2007.

[17] Gordon DB Cameron and Gordon ID Duncan. Paramics—parallel microscopic simulation of road traffic. *The Journal of Supercomputing*, 10(1):25–53, 1996.

[18] Eduardo Camponogara and Werner Kraus Jr. Distributed learning agents in urban traffic control. In *Portuguese Conference on Artificial Intelligence*, pages 324–335. Springer, 2003.

[19] Jordi Casas, Jaime L Ferrer, David Garcia, Josep Perarnau, and Alex Torday. Traffic simulation with aimsun. In *Fundamentals of traffic simulation*, pages 173–232. Springer, 2010.

[20] Stephen Chiu and Sujeet Chand. Adaptive traffic signal control using fuzzy logic. In *Fuzzy Systems, 1993., Second IEEE International Conference on*, pages 1371–1376. IEEE, 1993.

[21] François Chollet. keras. https://github.com/fchollet/keras, 2015.

[22] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[23] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[24] Carlos F Daganzo. The cell transmission model: A dynamic representation of highway traffic consistent with the hydrodynamic theory. *Transportation Research Part B: Methodological*, 28(4):269–287, 1994.

[25] Denise de Oliveira, Ana LC Bazzan, Bruno Castro da Silva, Eduardo W Basso, Luis Nunes, Rosaldo Rossetti, Eugénio de Oliveira, Roberto da Silva, and Luis Lamb. Reinforcement learning based control of traffic lights in non-stationary environments: A case study in a microscopic simulator. In *EUMAS*, 2006.

[26] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3837–3845, 2016.

[27] Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, diogo149, Brian McFee, Hendrik Weideman, takacsg84, peterderivaz, Jon, instagibbs, Dr. Kashif Rasul, CongLiu, Britefury, and Jonas Degrave. Lasagne: First release., August 2015. URL http://dx.doi.org/10.5281/zenodo.27878.

[28] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[29] Samah El-Tantawy, Baher Abdulhai, and Hossam Abdelgawad. Multiagent reinforcement learning for integrated network of adaptive traffic signal controllers (marlin-atsc): methodology and large-scale application on downtown toronto. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1140–1150, 2013.

[30] Samah El-Tantawy, Baher Abdulhai, and Hossam Abdelgawad. Design of reinforcement learning parameters for seamless application of adaptive traffic signal control. *Journal of Intelligent Transportation Systems*, 18(3):227–245, 2014.

[31] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

[32] J Favilla, A Machion, and F Gomide. Fuzzy traffic control: adaptive strategies. In *Fuzzy Systems, 1993., Second IEEE International Conference on*, pages 506–511. IEEE, 1993.

[33] Martin Fellendorf. Vissim: A microscopic simulation tool to evaluate actuated signal control including bus priority. In *64th Institute of Transportation Engineers Annual Meeting*, pages 1–9. Springer, 1994.

[34] Yiheng Feng, K Larry Head, Shayan Khoshmagham, and Mehdi Zamanipour. A real-time adaptive signal control in a connected vehicle environment. *Transportation Research Part C: Emerging Technologies*, 55:460–473, 2015.

[35] Wade Genders and Saiedeh Razavi. Using a deep reinforcement learning agent for traffic signal control. *arXiv preprint arXiv:1611.01142*, 2016.

[36] Peter G Gipps. A behavioural car-following model for computer simulation. *Transportation Research Part B: Methodological*, 15(2):105–111, 1981.

[37] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[38] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[39] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C Courville, and Yoshua Bengio. Maxout networks. *ICML (3)*, 28:1319–1327, 2013.

[40] PDP Research Group et al. Parallel distributed processing: Explorations in the microstructure of cognition: Vol. 1. *Foundations. Cambridge, MA: MIT Press*, 1986.

[41] Carlos Guestrin, Michail Lagoudakis, and Ronald Parr. Coordinated reinforcement learning. In *ICML*, volume 2, pages 227–234, 2002.

[42] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[43] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *arXiv preprint arXiv:1603.05027*, 2016.

[46] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[47] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[48] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[49] Venkatesan Kanagaraj, Gowri Asaithambi, CH Naveen Kumar, Karthik K Srinivasan, and R Sivanandan. Evaluation of different vehicle following models under mixed traffic conditions. *Procedia-Social and Behavioral Sciences*, 104:390–401, 2013.

[50] Andrej Karpathy, F Li, and J Johnson. Cs231n convolutional neural network for visual recognition,". *Online Course*, 2016.

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

[52] 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), 2012.

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

[54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[55] Lior Kuyer, Shimon Whiteson, Bram Bakker, and Nikos Vlassis. Multiagent reinforcement learning for urban traffic control using coordination graphs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 656–671. Springer, 2008.

[56] L. Li, Y. Lv, and F. Y. Wang. Traffic signal timing via deep reinforcement learning. *IEEE/CAA Journal of Automatica Sinica*, 3(3):247–254, July 2016. ISSN 2329-9266. doi: 10.1109/JAS.2016.7508798.

[57] 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.

[58] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013.

[59] Yann LeCun Mikael Henaff, Joan Bruna. Deep convolutional networks on graph-structured data. *CoRR*, abs/1506.05163, 2015.

[60] 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.

[61] 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.

[62] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[63] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[64] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. *arXiv preprint arXiv:1605.05273*, 2016.

[65] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *ICML (3)*, 28:1310–1318, 2013.

[66] Matthias Plappert. keras-rl. https://github.com/matthiasplappert/keras-rl, 2016.

[67] Jordan B Pollack and Alan D Blair. Why did td-gammon work? *Advances in Neural Information Processing Systems*, pages 10–16, 1997.

[68] LA Prashanth and Shalabh Bhatnagar. Reinforcement learning with function approximation for traffic signal control. *Intelligent Transportation Systems, IEEE Transactions on*, 12(2):412–421, 2011.

[69] Silvia Richter, Douglas Aberdeen, and Jin Yu. Natural actor-critic for road traffic optimisation. In *Advances in neural information processing systems*, pages 1169–1176, 2006.

[70] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[71] Raif Rustamov and Leonidas Guibas. Wavelets on graphs via deep learning. In *Advances in Neural Information Processing Systems*, pages 998–1006, 2013.

[72] As' ad Salkham, Raymond Cunningham, Anurag Garg, and Vinny Cahill. A collaborative reinforcement learning approach to urban traffic control optimization. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology-Volume 02*, pages 560–566. IEEE Computer Society, 2008.

[73] John Schulman, Sergey Levine, Pieter Abbeel, Michael I Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, pages 1889–1897, 2015.

[74] Zhen Shen, Kai Wang, and Fenghua Zhu. Agent-based traffic simulation and traffic signal timing optimization with gpu. In *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 145–150. IEEE, 2011.

[75] Lu Shoufeng, Liu Ximin, and Dai Shiqiang. Q-learning for adaptive traffic signal control based on delay minimization strategy. In *Networking, Sensing and Control, 2008. ICNSC 2008. IEEE International Conference on*, pages 687–691. IEEE, 2008.

[76] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395. JMLR Workshop and Conference Proceedings, 2014. URL http://jmlr.org/proceedings/papers/v32/silver14.pdf.

[77] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[78] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[79] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[80] Peter Stone, Richard S Sutton, and Gregory Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[81] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

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

[83] 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.

[84] Richard Stuart Sutton. Temporal credit assignment in reinforcement learning. 1984.

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

[86] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

[87] Thomas L Thorpe. Vehicle traffic light control using sarsa. In *Online]. Available: citeseer. ist. psu. edu/thorpe97vehicle. html*. Citeseer, 1997.

[88] Fei Tian, Bin Gao, Qing Cui, Enhong Chen, and Tie-Yan Liu. Learning deep representations for graph clustering. July 2014.

[89] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 2012.

[90] Elise van der Pol. Deep reinforcement learning for coordination in traffic light control. 2016.

[91] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR, abs/1509.06461*, 2015.

[92] Vladimir N Vapnik and A Ya Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. In *Measures of Complexity*, pages 11–30. Springer, 2015.

[93] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

[94] Vinh-An Vu, Giho Park, Gary Tan, and Moshe Ben-Akiva. A simulation-based framework for the generation and evaluation of traffic management strategies. In *Proceedings of the 2014 Annual Simulation Symposium*, ANSS '14, pages 9:1–9:10. Society for Computer Simulation International, 2014.

[95] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. 1974.

[96] R Wiedemann. Simulation des straßenverkehrsflusses. schriftenreihe heft 8. *Institute for Transportation Science, University of Karlsruhe, Germany*, 1994.

[97] Marco Wiering, Jilles Vreeken, Jelle Van Veenen, and Arne Koopman. Simulation and optimization of traffic in a city. In *Intelligent Vehicles Symposium, 2004 IEEE*, pages 453–458. IEEE, 2004.

[98] Marco Wiering et al. Multi-agent reinforcement learning for traffic light control. In *ICML*, pages 1151–1158, 2000.

[99] Luisa M Zintgraf, Taco S Cohen, Tameem Adel, and Max Welling. Visualizing deep neural network decisions: Prediction difference analysis.

Appendix A

Infrastructure

In this appendix we briefly present the infrastructure used to support the research in this work. Although this is a side aspect, we consider it to be important on its own due to the non negligible amount of time that was needed to define it.

In section A.1 we present the factors taken into account to select a traffic simulation software while in section A.2 we do the same for the deep learning framework on top of which our algorithms have been devised. Finally, in section A.3 we show the hardware setup used to perform all the experiments described in chapter 4.

A.1 Traffic simulation software

In order to test our control algorithms we need a simulation infrastructure that can mimic the traffic of a city and that allows external control of the traffic lights by our controlling agent.

Urban traffic simulation software can keep models at different levels of abstraction:

- Microscopic traffic models: simulate vehicles individually computing their positions at every few milliseconds (typically in the order of hundreds of milliseconds). The dynamics of the vehicles are governed by a simplified model that drives the behaviour of the driver under different conditions.
- Mesoscopic traffic models: simulate vehicles individually but compute their positions with larger time steps, typically at the times at which vehicles enter or leave road sections and in an event-driven manner (as opposed to a clock-driven simulation like microscopic models). This implies that the history of positions

within a road section for each vehicle is unknown. On the other hand, mesoscopic simulations require less computing resources.

- Macroscopic traffic models: estimate aggregated measures of the traffic, like the density in the different sections of the traffic network by taking into account parameters of the roads, like their capacity. Macroscopic models are less expensive computationally, and are normally used as a first approach to compute traffic densities with which individual traffic path assignment are calculated and subsequently used in microscopic simulations.

To our interests, the proper simulation level would be either the microscopic or mesoscopic, because we need information of individual vehicles and their responses to changes in the traffic lights. Nevertheless, in sake of more realistic results, we shall prefer microscopic simulations.

The following third party traffic simulators are the most widely used ones and have been considered as candidates for the infrastructure of the current work:

- **SUMO** (Simulation of Urban MObility) [52]: open source microscopic simulator created and maintained by the Institute of Transportation Systems, which is part of DLR[1]. It is actively maintained after more than 15 years of development and has a very large community, with even user conferences [2]. Lots of scientific publications rely on it [3]. It provides a programmatic interface to control the traffic lights from several different programming languages. Also, it offers the possibility to import real city maps from OpenStreetMap, a collaborative cartographic service similar in functionality to Google Maps.

- **MITsimLab** [94]: open source microscopic simulator created at the MIT Intelligent Transportation Systems (ITS) Program. Its very initial purpose was precisely to evaluate traffic control software. Like SUMO, there are lots of scientific publications relying on it [4], but it has no integration with OpenStreetMap maps and currently is has low momentum, having had its peak activity at early 2000s.

[1]DLR (Deutsches Zentrum für Luft- und Raumfahrt e.V) is the German Agency for aerospace, energy and transportation research.
[2]http://www.dlr.de/ts/desktopdefault.aspx/tabid-10490/18168_read-42578/
[3]See http://sumo.dlr.de/wiki/Publications for a list of publications
[4]See https://its.mit.edu/publications for a list of publications

- **Aimsun** [4, 19]: commercial microscopic, mesoscopic and macroscopic simulator with powerful modelling capabilities, integration with OpenStreetMap, and a Python API, and widely used, both in the private consulting sector and in traffic organization institutions.

- **Vissim** [33]: commercial microscopic simulator that is widely used traditionally by traffic organization institutions globally.

We shall note that in traffic research at academia, a very common microscopic simulation model is the Cell Transmission Model (CTM) [24]. It divides roads into small segments called cell. The division is performed both in the longitudinal axis and in the lateral one, hence forming a *grid* along the road. The length of the cell is computed to be equal to the distance traveled in free flow traffic in one time step, while its width is that of a single lane. During the simulation, each cell is associated a density that is computed based in the inflows and outflows. The cell transmission model, despite being very popular in the academic realm, is seldom used in real-world applications and has therefore being discarded as candidate.

Among the selected candidates, in terms of behaviour of vehicles, all of them are valid. SUMO implements the Krauß model [53], Aimsun implements the Gipps model [36], Vissim offers the Wiedemann model [96] and MITsimLab implements a custom model. However, in mixed traffic conditions all of them perform in similar manner ([49]), only differing in interurban setups where the distance between consecutive vehicles are larger.

After careful evaluation, we selected **SUMO** as simulation framework, because it was easily accessible (it is open source) and it is superior to MITsimLab, in terms of community and also regarding integration with OpenStreetMap, making it more difficult to simulate already existing maps.

However, after the first experiments it was realized the difficulty of having a simulation that was close to reality, because we lacked a *demand configuration* and the tooling provided by SUMO to create random demand configurations cast very unrealistic results. This happened in a time frame were **Aimsun** was made available to the author, along with realistic demand configurations of different cities. Therefore, it was decided to switch the simulation framework from SUMO to Aimsun.

A.2 Deep learning framework

Deep Learning research has been a very active field in the last years. This fact has led to the appearance of several open source deep learning libraries, many of them backed by research labs or big companies. The most remarkable ones are:

- **Theano** [86]: a low level Python library that allows to express arbitrary computations in the form of computational graphs and to compute them either on the CPU or the GPU, supporting also automatic differentiation of the computations. It was devised at Yoshua Bengio's lab at University of Montreal. It was the first solid deep learning library and it used extensively for deep learning research at academia and industry.
- **Tensorflow** [2]: a low level Python library created by Google, similar in purpose to Theano, but more flexible in terms of distributed computing.
- **Lasagne** [27]: a Python library that works on top of Theano, making it simpler to define neural networks.
- **Keras** [21]: a Python library that works on top of Theano or Tensorflow (it is configurable), making it simpler to define neural networks and also allowing to switch the backend to Theano or Tensorflow, which is useful in case of hitting a bug or misbehaviour in either of them.
- **Torch** [23]: a deep learning C library created at Yann LeCun's lab at University of New York. It has bindings to other programming languages such as C++, Python and Lua. It is heavily used at Facebook. It was formerly used at Google DeepMind, and they used it for their landmark articles [60, 61].
- **Caffe** [48]: a C deep learning library created at Berkeley University, extensively used for research. It offers bindings to other languages like Python, R and Matlab.
- **Deeplearning4j** [1]: a Java deep learning that is popular in industry due to the possibility of using it from any JVM language.

Note that all libraries rely on *CUDA* and *cudnn*, the libraries offered by *nvidia* to access the functionality of their GPUs.

Given that all evaluated traffic simulators offer a Python API, it was desirable to choose a Python-enabled deep learning framework. We decided to use **Keras** as deep

learning framework, not only because it was accessible from Python, but also because the possibility to switch between Theano and Tensorflow as computing backend, and the strong community Keras has, which readily provides support and advice. During our research, we decided to settle with **Tensorflow** as backend in order to avoid some Theano bugs we found.

Given the availability of several open source implementations of deep reinforcement learning algorithms on top of Keras, we decided to use library **keras-rl** [66] as a starting point for our implementation.

A.3 Hardware

In order to compute efficiently train deep learning models it is necessary to run them in capable hardware. We chose the following setup: an Intel Core i7-5820K 3.3Ghz processor, with 32GB DDR4 RAM and an nvidia GeForce GTX 980 GPU with 4GB of GDDR5 RAM.

Regarding the processor, Intel i7 architecture was the most powerful in terms of computing capabilities within domestic hardware. Regarding the GPU, it was necessary to have an nvidia graphic card. The reason is that only such vendor supports the *de facto* standard GPGPU computing platform, namely CUDA. Despite the fact that CUDA is proprietary and that no other vendor supports it, deep learning software mainly offers support for it. The alternative vendor, AMD, supports the open GPGPU standard called OpenCL, whose adoption is very low. This way, currently, nvidia GPUs are the only feasible hardware option to perform deep learning research.

Appendix B

Unsuccessful Approaches

In this chapter we briefly describe some approaches that were studied but led to no improvement over the more simple approach followed (described in chapter 3). This lines of research did not undergo full formal testing and therefore no associated results are included. Nevertheless, it was considered useful to mention other paths researched during the elaboration of this work.

The most successful cases of deep reinforcement learning ([60, 61]) receive the pixels of a video game as inputs, that is, the input state is a matrix. This enables the authors to apply convolutional neural networks in they deep reinforcement learning approach. Convolutional layers exploit the locality of the information to *identify* features, that is, they learn to detect groups of pixels that have some trait.

This way, a convolutional neural architecture may be able to profit in the same way from the *locality* of the detector information.

It is possible define the concept of "group of pixels" because the organization of pixels in a matrix gives us information of which pixels are next to other pixels, and such neighbourhood is homogeneous: one pixel always has only one pixel to the left, only one pixel to the right, one pixel above and one below. However, in our case we do not have a matrix as input, but a collection of (detector, detector info) tuples. Nevertheless, there is also the notion of locality in the detectors: there is certain travel time from one detector to another that can be used as a distance measure. However, such notion of locality is not homogeneous and does not enable us to apply convolutional networks to the detector information.

With the distance information, we could create a weighted directed graph where each node is associated to a detector and each pair of nodes is connected by an arc whose weight is the travel time between the detectors. We could define a threshold for the minimum distance for an arc to be present, so that we do not have a connect for every pair of detectors, but only for those that are below certain travel time.

There are research lines that study the possibility of applying convolutions to graphs, mostly by obtaining matrices derived from the graph, such as the graph laplacian [13, 71, 88], or by using diffusion kernels [59], or by elaborating a custom matricial receptive field [64]. Others, follow similar approaches but focus on the graph Fourier transform. There are also some research on deep learning on random walks over the network [14].

In our case, we have to take the following into account:

- It is not possible to define an embedding in the line of [14], as we need to represent the full connection graph, and not only a single node.
- Level of connectivity in our graph is arbitrarily set, that is, we define an arbitrary threshold of the travel distance among detectors under which there is connectivity. Therefore, applications that assume a formal graph structure like [13, 71, 88], may not make sense.

Therefore, it would be possible to use an *ad hoc* approach to represent our input space in a way that makes it possible to apply convolutions: for each detector d, we find a path that connects $2K + 1$ detectors so that the middle detector is d. Such path is elaborated in the following way:

- Based on macroscopic simulations, we extract paths assigned to vehicles in our simulation. Let this set of paths be referred to as P.
- For every detector d_1, we traverse P and find the probability of finding any other detector d_2 close in the same path. That is, we find a mapping between a pair of detectors and a probability: $f : DxD \leftarrow \mathbb{R}$. This mapping may not be complete, depending on the paths on P.
- For every detector d_1, we compose a path that contains d_1 in its center position and that contains K detectors to the right of d_1 and K detectors on its left. In order to compose it, we find the most probable detector to be found after d_1,

that is $\arg\max d_i f(d_1, d_i)$. We do this sequentially for K steps, and then do the same in backwards direction.

This way, for each detector d_i, now we have a path of length $2K+1$, where our detector is in the middle; let us call this path $p(d_i)$, defined as a collection of $2K+1$ elements, and where $p(d_i, j)$, refers to the j^{th} element in the path. Then, we define a transformed input state in which we have a matrix with the same number of rows as number of detectors in the traffic network, and with $2K+1$ columns, where in each row i and column j, we have the speed score (as defined in section 3.2) of detector $p(d_i, j)$. This allows us to apply 1-dimensional convolutions to the input space, capturing the network structure of our detector information.

Once a suitable input space is available, it can be fed into a convolutional architecture: a Deep Deterministic Actor-Critic Policy Gradient architecture (as described in section 2.3) with a path-based state space as described previously in this appendix. The layers used are classical ConvNets, as described in section 2.1: repeated blocks convolutional layers, followed by max pooling layers, which residual learning to improve learning while allowing greater depth.

The early results obtained did now improve over the simpler approach described throughout chapter 3, but did increase the complexity of the architecture, therefore the convolutional architecture with path-based state space was dropped.