NEURAL NETWORKS APPLICATIONS TO TRAFFIC CONTROL

March 14, 2016 - July 17, 2016

Student: Luciano Di Palma
Tutor: Gabriel Gomes

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

Berkeley
UNIVERSITY OF CALIFORNIA

Contents

1 Introduction

Traffic control is one of the main study areas in traffic engineering. In this domain, one is concerned with controlling the flow of vehicles in order to minimize the formation of congestions and reduce the idle time. For example, in a given intersection the amount of green time given to each road should be a function of how many vehicles are expected to come through, instead of setting a fixed time interval independently of the situation. However, deciding which traffic control decision to make at any given configuration if far from a simple task. Yet, solving this problem would have a major impact in our daily lives, by optimizing the traffic flow in highways and cities and thus reducing people waiting time in traffic.

Our approach to this problem has in mind the recent advances in Machine Learning, particularly in Neural Networks and Reinforcement Learning. Briefly speaking, Reinforcement Learning is the area concerned with training and designing intelligent agents, capable of learning and improving its decision-making from interactions with an environment. Neural Networks, on the other hand, is an extremely versatile and adaptable Supervised Learning technique, capable of recognizing patterns and similarities in data and after generalize to previous unseen data points. Techniques from these areas have been successfully applied in many different situations, such as image recognition [12], stock market prediction [11], fraud detection and even problems which were previously considered to be intractable to computers, such as playing Go at professional level [10]. From these examples, we see that these areas provide powerful tools, proven to be useful and adaptable to the most diverse scenarios, in special those which require some degree of *human-thinking* behind.

In face of these results, our expectations were to make a similar application of Reinforcement Learning techniques and Neural Networks in the traffic control problem. In particular, we have studied how to train *traffic light controllers* in the case of single traffic intersections. We note that in the past decades there have been several works [7, 8, 9] which have already applied techniques from the above areas in traffic prediction and control, which stimulated us to pursue new applications in this domain.

We start our exposition on the topic by first exploring how traffic simulations are performed and the most important quantities involved when modeling this kind of problem. After we expose the basics behind both Neural Networks theories the motivation behind using those methods, and finally how they can be applied to the traffic control problem. Simulation results and model performance discussion follow, and the performance of our intelligent controller can be evaluated under a couple different optics.

2 Traffic Simulation

Here we describe the problem we worked on in more details. We start by explaining how we model traffic, followed by an overview of the different kinds of traffic simulators, and finally we show which simulator has been chosen to our proposes.

2.1 Modeling a Traffic Scenario

A traffic scenario tries to captures all the important quantities involved in modeling and simulating the traffic of vehicles. For example, before any simulation of traffic flow is performed one must specify parameters such as the velocity of vehicles, how many will be entering the road, how much time the traffic lights will remain open, the number of lanes, and many other quantities. In order to simplify our description, we separate these parameters in two major classes: the *traffic network parameters* and a *flow profile parameters*.

2.1.1 Traffic network

The *traffic network parameters* correspond to all quantities responsible for capturing how the roads connect with themselves and which constrains they impose on the traffic of vehicles.

Before explicitly defining those parameters, our first step is building a *traffic network graph*. This consists of abstracting a given network as a graph $G(V, E)$, where the V is the set of *nodes* and E is the set of *links*. In this model, the links are an abstraction of the roads, while the nodes are the points of meeting of two or more links. This representation allows us to simplify how roads connect to each other, and it is the starting point when modeling any traffic network.

In the figure below, we have one example of very simple traffic network, consisting of a single intersection with vehicles coming in two possible ways, and its corresponding network graph. Although not explicitly drawn, we consider also that a traffic light controls when each lane of vehicles may proceed.

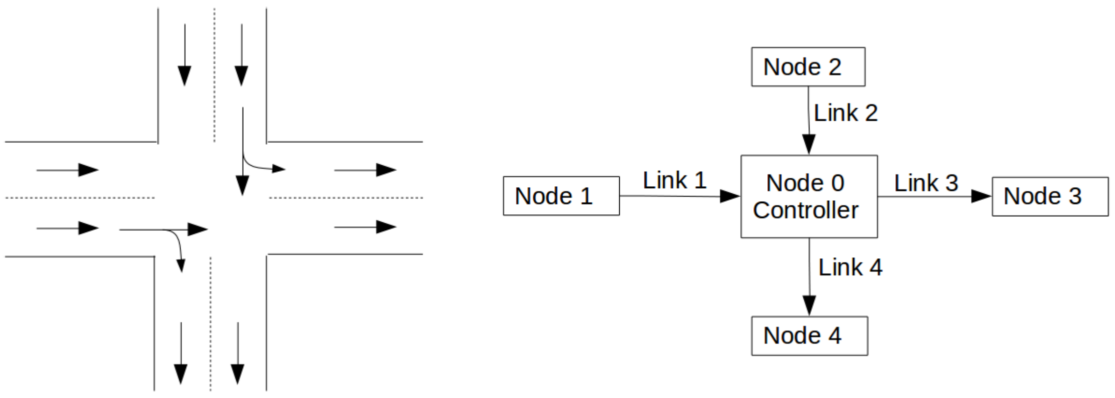


Figure 1: Simple traffic network and its Network Graph

We note that in our example nodes 1 and 2 are *source* nodes, places from where vehicles may enter our network. Similarly nodes 3 and 4 are *sink* nodes, from where vehicles can exit the network. The node 0, besides representing the point of junction between all links, also contains a Network Controller, in this case a traffic light.

After setting the main connection structure, we can start defining the traffic network parameters. First, we define the physical properties of each link, also called *road geometry*. The following parameters are specified:

- **number of lanes**: how many lanes there are in the link.
- **link type**: Every link belongs to one of two categories: *arterial* or *freeway*.

In addition, we have to set the *road parameters*, which are all properties related to the flow of vehicles. Three quantities are specified here:

- **capacity**: maximum rate of vehicles per hour the link can support.
- **speed**: free flow speed of vehicles.
- **jam density**: density of vehicles from which jams start to happen.

Once we have created the network graph, some information was lost in the procedure: we do not know anymore that, for example, cars coming from link 1 can turn right and connect to link 4. This information must be set separately, composing the *road connections*. In our example, we must include the connections:

$$link1 \to link3,\ link1 \to link4,\ link2 \to link3,\ link2 \to link4.$$

And this finishes all of the traffic network parameters. Observe that, at this point, all roads and possible paths for vehicles are set, together with all constrains to the traffic flow (velocity, number of lanes, etc).

2.1.2 Flow profile

Now we can pass to the another aspect composing a traffic scenario, the *flow profile*. These parameters determine the kind of flow profiles we will be dealing with: peak hours, congested lanes, free flow, etc. It is clear that different flow profiles can run under the same traffic network, each flow profile determining a different kind of traffic behavior to study.

Probably the most important parameter determining the flow profile is the *demand* of vehicles, that is, the rate of entering vehicles through each source node (in vehicles per hour). The entering of vehicles is stochastic in nature, and is modeled by a Poisson distribution in the traffic simulators we used. It is also possible to specify multiple means for the same source node; in this case, the mean is changed after some specified time interval. This allows for simulating more realistic scenarios, such as peak hours.

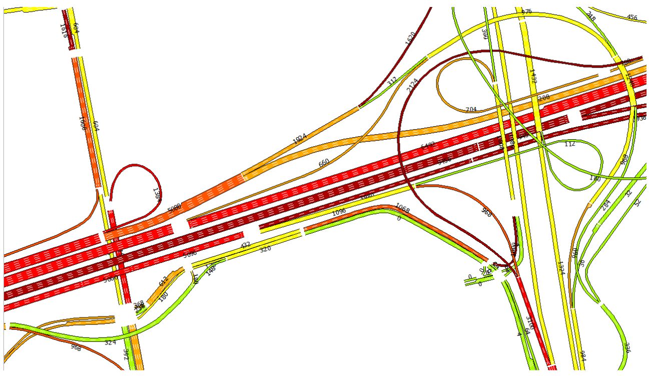
The last parameter that must be set is the *split ratio*. Whenever a vehicle has more than one possible choice of path (going straight or turning right, for example), we must determine which way it will follow. The split ration is thus the probability that the vehicle will follow each path.

These two parameters completely determine how the traffic flow will behave, and simulating a traffic flow can be done once a simulator is chosen.

2.2 Choosing a traffic simulator

Once a traffic network is chosen and its parameters are set, a traffic simulator is needed. There are basically three kinds of traffic simulators: Macroscopic, Mesoscopic and Microscopic.

The biggest difference between then is related to the level of detail when describing the individual vehicles behavior. In the Macroscopic model, we treat traffic flow as if it was a fluid, and traffic is simulated by solving a kinematic wave equation. In a mesoscopic model, vehicles still are not simulated individually, instead they are grouped in packets which accumulate in queues in the network. Finally, in microscopic models one has a model for how vehicles interact and behave individually in the network, and traffic flow is simulated in a car-by-car fashion.


(a) Macrosimulation example

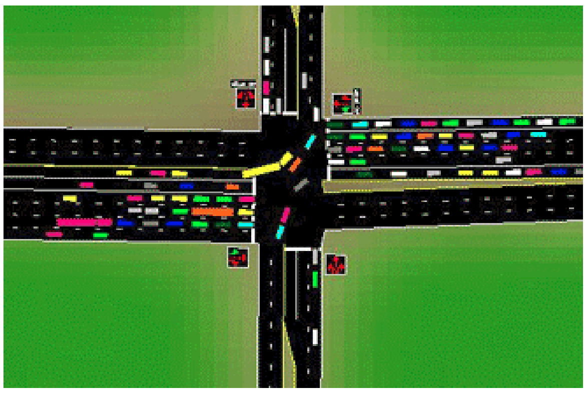

(b) Microsimulation example

Figure 2: Visual comparison of traffic simulators

It is clear that these different approaches in modeling may lead to different traffic behaviors. In general, it is known that macroscopic models works better in freeways, where traffic itself is not greatly constrained and where vehicles do not interact much. The mesoscopic model works great on arterials, and it can be simulated nearly as fast as the macroscopic model, but its performance is not as good on freeways comparatively. The microscopic model can in principle handle any situation accurately, but simulating every car is usually too costly to be handled efficiently in large scale scenarios.

In our current project, we worked exclusively with mesoscopic traffic simulators. Since we have chosen to work with in traffic lights controllers in arterials, this model seemed to be more appropriate than the macroscopic one, while being faster than microscopic simulations. Also, since we were expecting that thousands of scenarios needed to be constantly simulated for training our learning agent, speed was a critical factor to us.

2.3 Implementation details

Finally, we describe more in depth how the traffic network is implemented and the traffic simulator we have chosen to use.

All of the traffic network properties are written to a XML file. This file format allows for storing highly structured sets of data, making it easy and simple to both read and create new models. One example of such a file can be found in the appendix.

The traffic simulation software we used is currently being developed by my tutor, Gabriel Gomes. This simulator, called BeATS2, is capable of supporting macroscopic and mesoscopic simulations in the same traffic network, accounting for mode accurate predictions in both freeways and arterials. However, since it is still a work in progress, as the time of this writing the macroscopic traffic simulator is not still implemented, being still another reason for choosing mesoscopic simulators in our study. Yet, the mesoscopic simulator was enough for our developing purposes (besides being more adapted to our needs), so no drawback was accrued from using BeATS2 under this aspect.

For the purpose of using the simulator without worrying about implementation details, an API class was developed containing all methods needed for running a simulation. In addition, this API allowed us to decouple the work on training a traffic controller agent from the work on the traffic simulator itself, allowing for more efficiency on both parts. This API contained the following methods:

- **load_and_check**: reads the properties file, creating a network model and setting internally its parameters.
- **initialize**: makes the simulator ready for the next run. All links are initialized empty (0 vehicles).
- **run**: run a simulation for a given time duration.
- **get_occupations**: returns the number of vehicles in each link.
- **set_occupations**: set the number of vehicles in each link before the next run.
- **set_green_times**: interface with the traffic light controller.

In conclusion, once a traffic scenario was defined, we see that the above API allowed for easy generation of traffic simulations.

3 Neural Networks

Neural Networks is one of the most popular Supervised Learning techniques being studied and used in our days. The main reason for its popularity is its *high generalization capability*; in other words, it can approximate any reasonable regression function we want, provided we have a sufficient number of neurons (and training data). Below we describe its main features and how it integrates in the traffic model.

3.1 Perceptron

The very first model of a neural network was called Perceptron (or Single Layer Perceptron). It was developed by Frank Rosenblatt in late 1950's [1], in an attempt to model the internal functioning of a real neuron. Rosenblatt suggested the following model for how a neuron reacts when given a stimuli:

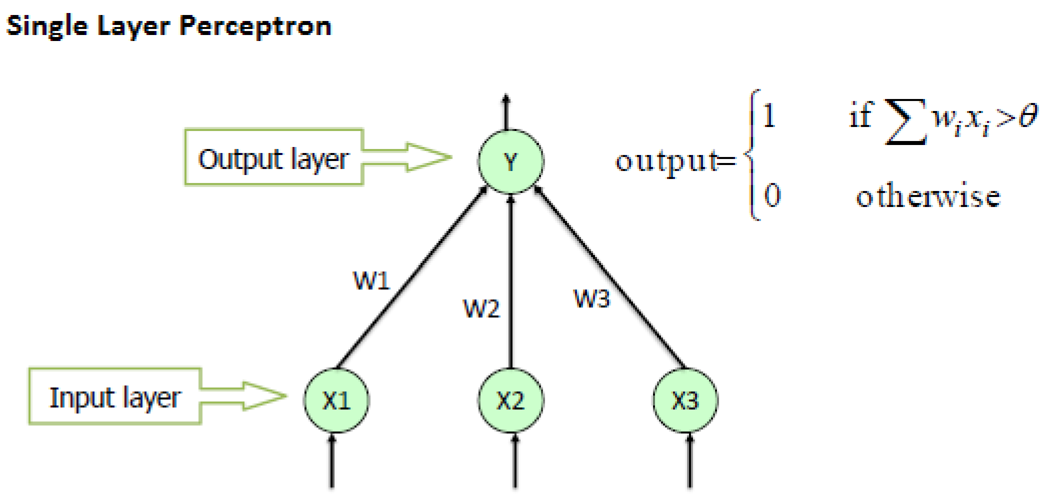

Single Layer Perceptron
Output layer
Y
$\text{output} = \begin{cases} 1 & \text{if } \sum w_i x_i > \theta \\ 0 & \text{otherwise} \end{cases}$
W1
W2
W3
Input layer
X1
X2
X3

Figure 3: Perceptron model

1. The stimuli is encoded by an **input** vector $x = (x_1, \dots, x_n)$, each value corresponding to the stimulus from a different neuron.
2. Each input value is then *weighted* differently, in a way to attribute a different importance to each stimulus.
3. Finally, if the total stimulus $\sum_{i=1}^{n} w_i x_i$ surpasses a threshold value θ, the neuron *fires* (i.e. returns 1), propagating the stimulus to other neurons. Otherwise, it does nothing (i.e. returns 0).

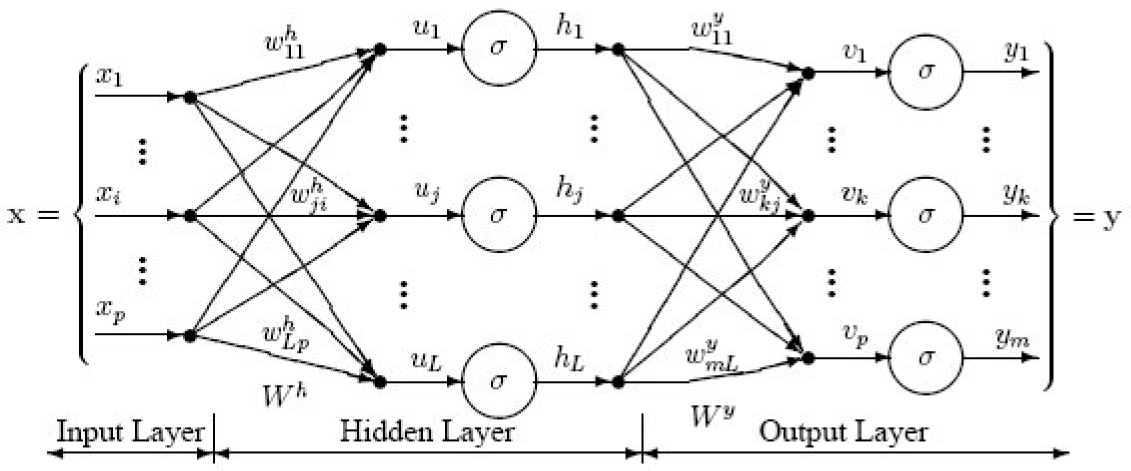
Under today's optics, we see that the Perceptron model is nothing more that a Linear Binary Classifier. First, for every point $x \in \mathbb{R}^n$ it outputs a binary class $f(x) = \sigma(w^T x - \theta)$, where the *activation function* σ is the sign function. Secondly, we observe that the equation $w^T x - \theta = 0$ represents an *hyper-plane* dividing the input space in two halves: one classified as 1 (above the plane) and other as classified as 0 (below the plane).

This observation tells us the main drawback of the Perceptron model: it cannot learn anything more complex than a hyper-plane. In order to alleviate this fundamental restriction, a variation of this model was developed: the Multi-Layer Perceptron.

3.2 Multi-Layer Perceptron

The Multi-Layer Perceptron (MLP), as the own name implies, corresponds to connect many Perceptron neurons together, mimicking a real network in our brain. One such network is represented below:

Figure 4: Multi-Layer Perceptron with 3 layers

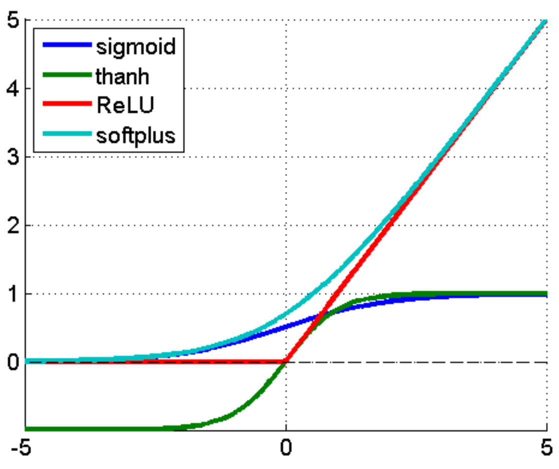


The way the MLP handles stimuli is actually quite simple. First, the input vector x is fed to every neuron in the Hidden Layer. Each one of these neurons produces its output, which in turn will be used as input to the next layer of neurons. This process repeats itself until we reach the last layer of neurons.

Besides the structural change above, another change made to the Perceptron model is the choice of activation function σ. Since the sign function constrains the output to either 0 or 1, it clearly hinders the generalization capabilities of the network. Thus, it is usual to choose other kind of activation functions, such as the *sigmoid*, *hyperbolic-tangent*, *Rectified Linear Unit* (ReLu), SoftPlus, SoftMax, etc.

We first notice that this model allows for a much more general output format, in both dimensionality and non-linearity. Also, the number of hidden layers and number of neurons-per-layer can be chosen arbitrarily, allowing for arbitrarily non-linear functions to be modeled. More precisely, it has been shown [2, 3] that given enough neurons, a 3-layer MLP can arbitrarily approximate any continuous function in a compact set.

Figure 5: Some activation functions



3.3 Back-propagation

The most difficult aspect when dealing with neural networks is the training procedure. In the Supervised Learning settings, we are furnished a collections of labeled data points $(x_i, y_i) \in X \times Y, 1 \leq i \leq n$, and we want to find a function $f : X \to Y$ that relates both values: $y = f(x)$. In particular, in a Regression problem (where y is a continuous value), we usually search the function f the minimizes the *Mean Squared Error* (MSE) over the training set:

$$R(f) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{1}$$

When approximating the regression function f by a MLP, once a network architecture has been chosen the training procedure translates into an optimization problem: find the set of weights in the net which minimize the error above.

There are many different optimization techniques which can be employed here. The standard approach is a variant of the Gradient Descent algorithm, called *Stochastic Gradient Descent*. Roughly, the difference between both methods is that while the usual Gradient Descent has to compute the gradient of the error function over the entire database, the stochastic approach notices that we can obtain a reasonable approximation by only sampling a *mini-batch* of points from the database. This allows for (much) faster iterations and more efficient use of data.

Finally, one last step is needed to use the above optimization procedure: the gradient of a neural network (with respect its weights) has to be calculated. This is done by an ingenious algorithm, called **back-propagation**. Its derivation can be found in most standard texts and tutorials on the subject, and since it involves many steps of tedious calculations, we leave a reference [4] to the topic.

3.4 Implementation details

During our project, a neural network implementation was developed in the Python language. It only contains the most primitive aspects, such as feedforwarding an input vector and making a simple Stochastic Gradient Descent optimization task. To achieve better performances we extensively made use of the Numpy library.

While our implementation was reasonable enough for most cases, there are better alternatives already implemented. In particular, we have also used two specific libraries: **Keras** and **Neupy**. Both contain very general and performing Neural Networks implementations in a Python framework called Theano [6], which is being highly used for intensive numerical computation tasks. The main advantage of using a Theano-based software is the speed proportioned: when making calculations, it first constructs an *operation graph* representing all the computations needed to be performed, and checks for possible optimizations to be done. For example, it is able to recognize inefficient patterns such as $x + x + x + x$ and substitute them for optimized equivalent versions, in this case $4x$. Another possibility of speed up is GPU integration, but we did not have the opportunity to test this feature.

Another advantage of using such libraries is that we have more freedom in choosing the optimization procedure that will be used. Besides the Stochastic Gradient Descent, there are quite a number of other possibilities. Just to name a few, we have RProp, RMSProp, Adaline, Adam, Adagrad, etc. Some of them do not even require hyperparameter optimization, because of their relative insensitivity to changes in values, which clearly speed up training procedures overall.

4 Reinforcement Learning

For the purpose of training an intelligent traffic agent, we turned ourselves to the Machine Learning subarea known as Reinforcement Learning. In this section we show which are the motivations behind this area and which solutions it offers for training intelligent, adaptable agents.

4.1 Motivation

The Machine Learning domain is usually divided in three categories: Supervised Learning, Unsupervised Learning and Reinforcement Leaning. This division is made based on the different approaches of how we can *learn* to perform a determined task. For example, in the Supervised Learning one is concerned with *learning through examples*, or in other words, ”given a set of examples (x_i, y_i), can you predict the value y given a new input x?”.

However, this approach is not capable of learning anything a human is able to. In fact, its main weakness lies in the necessity of having training examples (x_i, y_i) in order to learn. This has a strong analogy with how humans learn through a professor or a book: we receive new examples, discover patterns, and apply them elsewhere. In particular, this kind of technique is not directly applicable to the traffic control problem, since we do not have examples of the best decisions to be made. In fact, they ought to be learned through a *trial-and-error* approach instead.

What we are really looking for is a way to learn through *experience*. Consider a chess game for example. It is by playing many games, making good and bad modes repeatedly, and by evaluating our own performance that we develop strategies and improve our knowledge. In a similar way, one would expect that by running many simulations of traffic flow, and evaluating whether the control actions have improved or worsened the traffic flow, the traffic controller ought to improve its decision-making strategies over time.

It is this kind of problems that Reinforcement Learning tries to tackle. It has had astonishing achievements in the past few years: ”computers” have been successfully trained to play Atari at human-level [13], play Go at professional level [10] (thousands of times more complex than chess), and we even achieved to train robots to walk and play soccer [14].

4.2 Mathematical Formulation

Here we describe how Reinforcement Learning is mathematically modeled, and how to formulate an optimization schema for solving this kind of problem.

4.2.1 Markov Decision Process

A Markov Decision Process is a general mathematical model for the interaction of an agent with a random environment. In this model, the agent interacts with the underlying environment under the following dynamics:

Figure 6: Google AlphaGo match against world-champion Lee Sedol. AlphaGo won 4 out of 5 games.

1. At instant t, the agent makes an action A_t, changing the environment's state from S_t to S_{t+1}.

2. The agent receives an reward R_{t+1} for his action, which represents how good was the outcome of his last decision.

3. Finally, the agent observes the new state S_{t+1}, and decides which decision to make based on this new information available.

In order to clarify the above concept, we apply its modeling to a chess game. In this case, the environment is the game/board itself, an state being represented by the position of each piece on the board. The agent is one of the players, and its possible actions are comprised of all legal movements that can be made. As for the reward, we could develop a simple heuristic such as if the last movement results in a win you receive 1, if it ends in you losing you receive -1, and in other cases 0 reward is given.

In addition to the above dynamics, one further supposition is made, called **Markov Property**. It states that *given the present state, the future states are independent of the past*:

$$\mathbb{P}\left(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t, \ldots, S_0 = s_0, A_0 = a_0\right) = \mathbb{P}\left(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t\right)$$

This settles the general behavior of how an agent can interact with an environment and how he is rewarded by his actions. Now we introduce some vocabulary associated with the Markov Decision Process. It is composed by the following objects:

- **State Space (S):** This is a set of states that our system can be placed on. The current system's state will be denoted S_t, where t is the time instant. This set can be *discrete* (finite) of *continuous* (intervals in $\mathbb{R}^n$). In the chess example, the state space is composed by all possible configurations of pieces in the board (discrete space).

- **Action Space (A):** This is the set of decisions the agent can make. A_t is the action chosen by the agent at instant t. This set can be discrete or continuous also.

- **Transition Probabilities (P):** Given a initial state $s \in S$ and an action $a \in A$, the random nature of the environment makes it impossible to predict which state we are going to end after making a decision. This defines a set of transition probabilities given by:

  $$P_{ss'}^{a} = \mathbb{P}\left(S_{t+1} = s' | S_t = s, A_t = a\right)$$

  We note that the above probabilities are independent of time t; this is a consequence of the above Markov Property. In the chess example, these would represent a model of how your opponent plays, which is clearly not known a priori. There are ways to avoid this limitation.

- **Rewards (R):** At a given state $s_t \in S$, every action $a_t \in A$ incurs into an immediate reward R_{t+1} received by the agent. We note that this reward may depend on the unknown future state S_{t+1}, so in the general case it is a function $R_{t+1} = R_{t+1}(S_t, A_t, S_{t+1})$. Usually the reward function is *designed* to capture what we think that distiguishes good decision-making from poor decision-making.

- **Policy (π):** The policy is the model for the *agent's behavior* during the process, that is, how he picks an action at a given state. Given the stochastic nature of the environment, the policy is usually represented by a function $\pi : S \to \mathcal{P}(A)$, assigning to any given state a probability distribution over all possible actions. In some cases however, we might be interested in finding a *deterministic policy*, that for every state gives an specific action to be taken: $\pi : S \to A$.

In the general case, we observe that not all of the above quantities are known beforehand. In particular, the transition probabilities $P_{ss'}^{a}$ are rarely known, and some kind of workaround is needed in those cases. The usual approach is to turn to *model-free* reinforcement learning algorithms, which are able to learn an optimal policy by means of simulating several interactions with the environment, without relying on a model for the environment behavior.

4.2.2 Training a Learning Agent

For the sake of simplicity, in the underlying section we suppose to work in **discrete** state and action spaces. However, the conclusion obtained are still valid in the continuous frameworks.

One further supposition we make is to consider only the *single-step MDP*, where a simulation ends after the very first step ($t = 1$). We make this restriction because it was sufficient for our research purposes in traffic control, and also because the problem formulation stays simpler. We give more details in the next section.

In this restricted formulation, the objective of the agent is to receive the largest amount of reward in a single time-step, adapting its decisions to the random initial state S_0. In order to compare the efficiency of each policy π, we define a *reward function* $J(\pi)$ as:

$$J(\pi) = \mathbb{E}_{s,\pi}\left(R_0\right) = \sum_{s} d(s) \sum_{a} \pi(s,a)\, r(s,a) \tag{2}$$

where $d(s)$ is the probability distribution for the initial state S_0. The reward function allows us to compare different policies on the same standards: the best policy is the one that is expected to return the most reward for any given initial state s.

In order to find π, we are going to approximate it by a function approximator π_θ, for example a neural network. This makes the problem easier to solve, turning the original question into a optimization problem: find the parameter θ which maximizes the reward function:

$$J(\theta) = \mathbb{E}_{s,\pi_\theta}(R_0) = \sum_s d(s) \sum_a \pi_\theta(s,a)\, r(s,a) \tag{3}$$

There are a couple ways to approach the above maximization problem. We have in particular worked with two optimization techniques: the *cross entropy method* and the *policy gradient* technique. Both will be described below.

4.3 Algorithmic Approach

As we said before, we have employed two major techniques to approach the traffic problem: the *cross entropy method* and the *policy gradient* technique. Both have quite different approaches to the same problem, but return great results nonetheless.

4.3.1 Cross Entropy Method

In general, optimization techniques rely on computing the derivatives of the cost function and them performing a gradient descent step or similar technique. However, the cross-entropy method is a *gradient-free* optimization technique: the gradient of the cost function is never computed, and the objective function is not assumed to be differentiable.

Actually, the idea behind this method is very simple. Given a cost function $J(\theta)$, the cross-entropy method find its maximum by recursively updating the parameters of a Gaussian distribution $\mathcal{N}(\mu, \sigma)$:

1. Choose any initial condition (μ_0, σ_0).
2. Sample N parameters $\{\theta_1, \dots, \theta_N\}$ from the current distribution.
3. Calculate the cost associated with each choice of parameters: $J(\theta_1), \dots, J(\theta_N)$.
4. Select the parameters $\{\theta_i\}_{i\in I}$ corresponding to the p% largest values $J(\theta_i)$ found.
5. Fit a new Gaussian distribution from these selected values, by calculating the mean and variance of $\{\theta_i\}_{i\in I}$:
$$\mu = \frac{1}{|I|}\sum_{i\in I}\theta_i, \quad \sigma^2 = \frac{1}{|I|}\sum_{i\in I}(\theta_i - \mu)^2$$
6. Return to step 2 until convergence.
7. Return μ as the optimum value.

It is reasonably intuitive that this method converges to a local optima of the function $J(\theta)$. At each iteration it takes a step towards a region augmenting the value of J, and after it reaches a local maxima it gets stuck if the variance is too small, going back and forth this point.

One other problem with this method is that it tends to converge too quickly to a local optima, finding a solution that is not even close the global maxima desired. That is mainly because at each iteration the variance of our sample decreases, until it stays too low to result in any change. To avoid this, one technique is to add a constant factor to the variance, so it never becomes zero and exploration continues.

In terms of algorithmic complexity, most of the calculation time is usually spent on step 3, since it involves simulating the environment several times in order to estimate the reward function, and after repeat this estimation for each sampled parameter. This overhead in calculation is the main drawback of this method, but it is the price to pay for not relying on a derivative to find an ascent direction.

4.3.2 Policy Gradient

The policy gradient is a Gradient Descent based technique, and as such it relies on computing the gradient of the cost function $J(\theta)$. Considering that our policy is not deterministic, and using the relation $\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a)$, differentiating equation 3 with respect θ gives:

$$\begin{aligned} \nabla_\theta J(\theta) &= \sum_s d(s) \sum_a \pi_\theta(s,a)\, r(s,a)\, \nabla_\theta \log \pi_\theta(s,a) \\ &= \mathbb{E}_{s,\pi_\theta}\left(R_0 \nabla_\theta \log \pi_\theta(S,A)\right) \end{aligned} \tag{4}$$

The last expression is easier to deal with, since it can be approximated by means of the law of large numbers. More precisely, suppose we have independently sampled n initial states S_i, and from each of those we sample an action $A_i \sim \pi(S_i)$ (remembering that $\pi(S)$ is a probability distribution over the actions). The corresponding rewards R_i can be found by simulation, and we approximate the true gradient of J by:

$$\nabla_\theta J(\theta) \approx \frac{1}{n} \sum_i R_i \nabla_\theta \log \pi_\theta(S_i, A_i) \tag{5}$$

Thus, once computed the gradient we can repeatedly perform a gradient descent step in order to find a local maximum:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{6}$$

The advantage of this particular method is that the gradient gives the true direction of maximization, so it may converge quicker than the cross-entropy method. However, here we have to choose a new hyper-parameters, the learning rate α, which can have drastic impacts on the performance: if too big, we may never converge to a solution, while if too small we may get take too much time to converge or get trapped early (see figure 7).

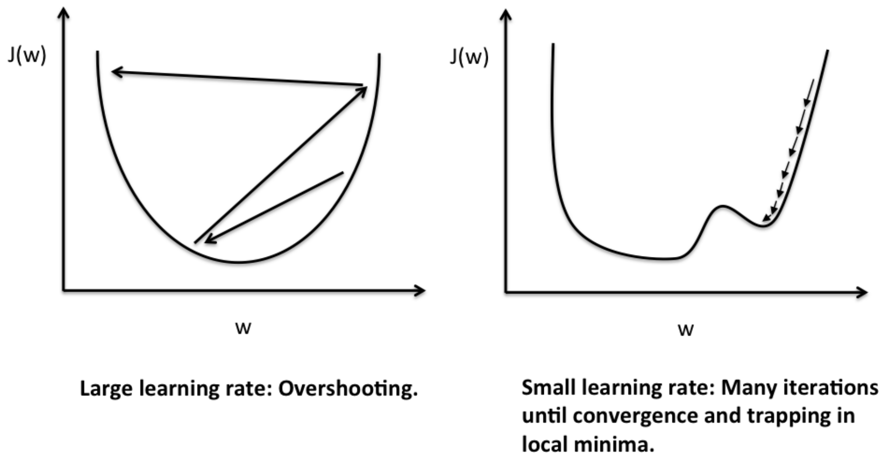

J(w)
w
Large learning rate: Overshooting.
J(w)
w
Small learning rate: Many iterations until convergence and trapping in local minima.

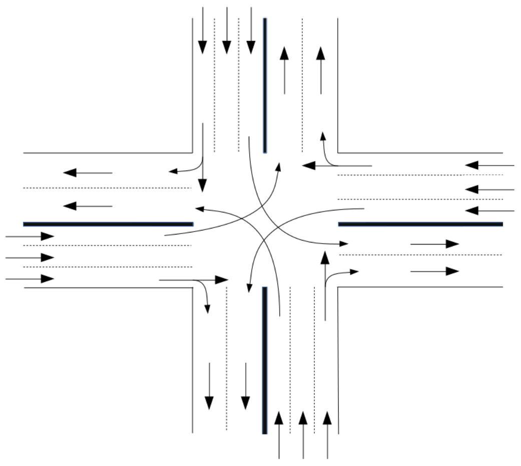
Figure 7: Effect of learning rate on Gradient Descent

5 Traffic Control

Finally, we exploit the above methods and ideas and apply them directly to the traffic control scenario.

5.1 Traffic Lights Controller

In order to evaluate more easily the performance of our algorithms, we mostly worked with the simplest traffic lights control scenario, that being the single intersection. More specifically, we have dealt with two different traffic network configurations: the one in figure 1 and the one below:

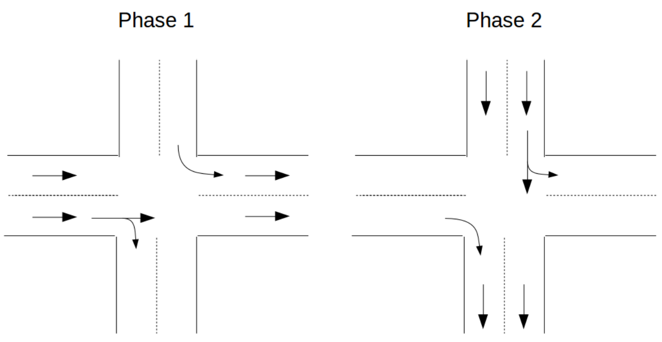


+

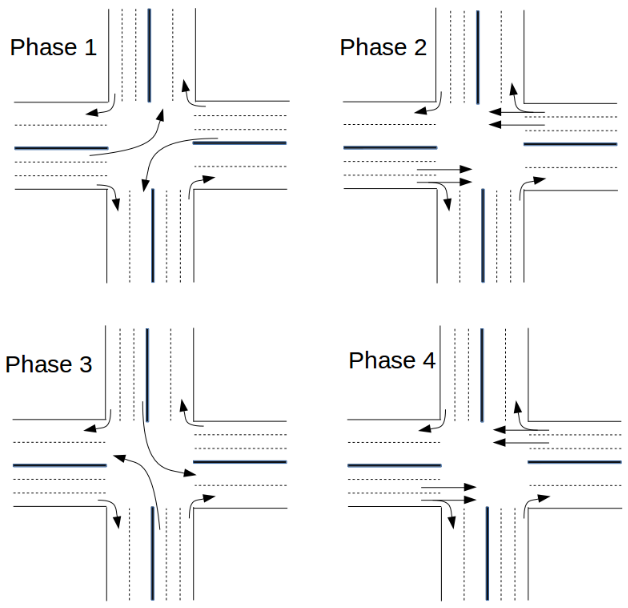
Figure 8: A more complex example of traffic intersection

We refer to both as Simple Intersection and Complex Intersection, respectively.

At any given moment, not all vehicles can be moving at the same time, because there would surely be crashes. Only some of all possible movements can be done at each time: those permitted movements being called *phases*. The phases for both Intersections are represented in figure 9.



(a) Phases for Simple Intersection



(b) Phases for Complex Intersection

Figure 9: Phase diagram for Traffic Intersections

We note that the phase order remains unchanged once fixed, because real drivers have an ”expectation” of when is his turn to go, an if constantly changed it could cause confusions and in worst case accidents.

The role of the traffic light controller is to assign how much time each phase gets to itself. Another constrain imposed is that the total time of each *cycle* (sequence of all phases) is constant, usually a couple minutes. The reason for this is to avoid unreasonable amounts of time being taken to complete a cycle. This also allows us to model the traffic lights actions as *what is the proportion of total time given to each phase?* So, if T_{cycle} is the cycle duration, and p_i is the proportion assigned to phase i, then each phase receives $g_i = p_i T_{cycle}$ seconds of green time. Obviously, $\sum_i p_i = 1$ and $\sum_i g_i = T_{cycle}$.

5.2 Modeling

Here we show how our previous models and algorithms fit in this the context described above.

5.2.1 General settings

Now we know how the agent (traffic lights controller) interacts with our environment (traffic scenario), we can formulate our problem in the Reinforcement Learning framework:

- **State Space (S):** In our model, the state S_t of the traffic network is the demand of vehicles for each source link at the current instant t. We remember that the demand represents how many cars are expected to enter a link during the next hour. We also note that the state space is a discrete set.
- **Action Space (A):** As we saw before, an action A_t corresponds to the proportion of green time $(p_1, p_2, \ldots)$, $p_i \geq 0$ and $\sum_i p_i = 1$, given to each phase by the controller.
- **Transition Probabilities (P):** Our environment is stochastic in nature, since the vehicles enter in the network in a random fashion. We did not try to model the transition probabilities since our algorithms are *model-free*.
- **Rewards (R):** The rewards are a bit more difficult to model, since no perfect traffic performance metrics are known. In our case, we set $-R_{t+1}$ as the *mean largest queue of vehicles in the last cycle of simulation*. The idea behind is that poor time distributions lead to cars accumulating in certain links, which we expect to avoid at all costs. The mean value is used to reduce variance of results, since vehicles enter the network randomly.

As for the policy π_θ, we have chosen as function approximator a Neural Network, mainly because of its high adaptive capabilities. One drawback of recurring to them is the large number of parameters/weights it has, in the order of thousands, but its generalization capacities outweighted the cost associated with the larger dimensionality.

One approximation we made is to consider only single-step scenarios, which means that each episode in the above MDP terminates at $t = 1$. This assumption simplifies both the formulation of the problem (as we saw in the previous section) and the simulation costs. Furthermore, it is not a bad supposition to make in traffic control, because it is rarely the case where the controller must make the traffic worse in order to make better improvements, so local optimizations are enough to find very reasonable solutions.

Another thing that must be taken into account is how much simulation time a time-step $t \to t+1$ corresponds to. If we simulate for any duration, then the states S_t and S_{t+1} may correspond to different phases and situations in the cycle. Thus, we chose run our simulations for a *multiple of the cycle duration* $T_{simul} = k\, T_{cycle}$, $k \in \mathbb{N}^*$, since in this case in both beginning and end of simulation we will always find ourselves in the beginning of a cycle.

5.2.2 Adapting the Cross-Entropy Method

The CEM can be adapted fairly straightforward to our settings, since the algorithm depends only on our capacity to efficiently evaluate the reward function $J(\theta)$ and choosing a function approximator for the policy π, which had already been set to a Neural Network.

We evaluate the function J by means of the Law of Large Numbers: many states $S_1, \ldots, S_N$ are sampled independently, the corresponding green times $A_i = \pi_\theta(S_i)$ (we use a deterministic policy) are then computed from the current network's weights θ, and finally traffic simulation are run in order to calculate the rewards R_i. One complication in this process is the stochastic nature of our environment, which makes the variable R_i noisy as a result. In order to avoid this problem, a few more simulations are run for each pair (S_i, A_i) and the final reward R_i is the mean of the computed intermediate rewards.

The above process ends up being rather costly computationally, and as such we searched a way of reducing the number of simulations performed. One way we discovered to be reasonably faster is to fix a (small) set of well-chosen initial states $\{s_1, \dots, s_n\}$, and use the CEM to separately compute the optimal green-times g_i only for each chosen state s_i. In this case, we simplify our policy to use *softmax* function:

$$\pi_\theta(s) = softmax(\theta(s)) = \frac{e^{\theta_i(s)}}{\sum_j e^{\theta_j(s)}}, \forall i \tag{7}$$

Once the g_i's are computed, we fit a regression function to those points using a Neural Network, extending our solutions to all possible states. Since training a Neural Network is fairly quick in reasonably sized training sets, most of the computation time will be in computing the solutions for each chosen state, which we have found to be made quicker than the previous settings.

In order to distinguish the two approaches, we call the first the *pure CEM* method, while the second we call *discrete CEM* approach.

5.2.3 Adapting the Policy Gradient Algorithm

To apply the policy gradient method, we must first define a stochastic policy π. We chose to model our policy as a von Mises-Fisher distribution $\pi(s) \sim \mathcal{M}_p(\mu(s), \kappa)$:

$$\pi(s, a) = C_p(\kappa) \exp(\kappa\, \mu(s)^T a), \tag{8}$$

where $C_p(\kappa)$ is a normalization constant. The von Mises-Fisher distribution is the analogue of a Gaussian distribution over the *sphere* $\mathcal{S}^{p-1}$. Points sampled from $\mathcal{M}_p(\mu, \kappa)$ are centered around the *mean direction* $\mu \in \mathcal{S}^{p-1}$. The parameters κ is called the *concentration parameter*, because the larger its value the more concentrated the samples are to the mean μ. In the limit where $\kappa \to 0$, $\mathcal{M}_p(\mu, \kappa)$ converges to an uniform distribution over the sphere.

Under this model, the mean direction varies with the state we are considering, letting the policy adapt to the situation encountered. We choose to represent the mean by a neural network $\mu(s; \theta)$ (θ is the network's weights). More specifically, the neural network receives as input the state s and outputs the *green time proportions* $p_i(s)$. The mean direction of the distribution is then calculated by taking a square root $\mu_i(s) = \sqrt{p_i(s)}$ (so it is a point in the sphere).

Another advantage of choosing the von-Mises Fisher distribution is that the log-policy gradient from equation 5 can be easily computed:

$$\nabla_\theta \log \pi_\theta(s, a) = \kappa \sum_i a_i \nabla_\theta \mu_i(s; \theta) \tag{9}$$

In the above computation, we have at some point calculate the gradient of the neural network, which can be made by backpropagation. We chose to use the already implemented gradient function in the Theano framework in Python, since it is a reliable, fast and automated method.

6 Simulation Results and Discussion

With the theory and algorithms sorted out, we turn to the results obtained thorough simulations. We intend to show that our models are able to train intelligent traffic lights controllers, capable of adapting to different scenarios and demands.

As we said before, we worked with two kinds of traffic intersections, the Simple and the Complex ones. In both cases, we have used the following set of traffic network parameters:

Table 1: Parameters used through-out simulations

Capacity (veh/h)	Velocity (km/h)	Jam density	Simulation Time (s)	Cycle Time (s)	Yellow Time (s)	All-red Time (s)
1800	70	100	600	120	3	2

These parameters where chosen by having in mind a high-demand traffic intersection. As for the demands, we have fixed a 1200 vehicles/hour throughput (sum of all demands). In the high-demand traffic intersection scenario, small changes in the green time distribution can have large impact in the performance of the traffic controller, being easier to evaluate if sensible decisions are being made.

When calculating the rewards R_{t+1}, we have run 25 simulations and averaged the results. When estimating the value of the reward function for a given policy, 400 state samples were used.

For each scenario, we are going to describe the results of both CEM and Policy Gradient methods. We also show the results for the *discrete CEM* approach, and its advantages comparatively to the other two approaches.

6.1 Simple Intersection Scenario

In this traffic network, we chose the split ratios in a way to avoid right turns, so all vehicles would drive straight. This makes easier to assess the quality of our results.

6.1.1 Simulation parameters

We first describe the simulations conditions. For the CEM method, 50 parameters θ were sampled per iteration and only the $p = 10\%$ best ones were chosen. The initial parameters for the Gaussian distribution where $\mu_0 = 0$ and $\sigma_0^2 = Id$. When estimating the reward, we found that 25 samples were sufficient (small variance and sufficiently fast computation). Simulation was stopped once the variance σ^2 dropped below $\epsilon = 0.01$.

In the discrete CEM technique, we chose our sample states by ranging the demand on link 1 from 0 to 1200 in intervals of 50, the demand on link 2 being the complement to 1200. When running each individual state optimization, all the parameters where kept the same as in the above simulations. Training the Neural Network was done through 1500 epochs, using the Adagrad optimization technique. Other optimization techniques were also tried, but no significant improvement was observed.

In the policy gradient method, our simulation consisted of 50 iterations only. The parameter κ was set to 20 in the beginning, and at each simulation it was multiplied by 1.1 until it reaches

500. This accounts for the fact that in the beginning of the simulation our learning agent is better off exploring more the action space (small κ) in order to improve, becoming increasingly more accurate (large κ) as time passes. We sampled 500 points when estimating the reward function's gradient, and used a learning rate $\alpha = 0.005$.

We have chosen to use the same neural network architecture through all cases, a 3-layer network with a 100-neurons hidden layer. All activations are hyperbolic tangent functions, expect for the output layer which is a softmax activation. The softmax returns normalized outputs (positive numbers that sum to 1), which is the case of the green time proportions we searched to fit.

6.1.2 Discussion of Results

Since there are only two source links in this scenario, and the total demand is fixed to 1200 cars/hour, the flow dynamics is dependent on the link 1's demand/green time proportion only. By applying the Cross-Entropy Method and the Policy Gradient to this traffic scenario, we obtained the following final policy:

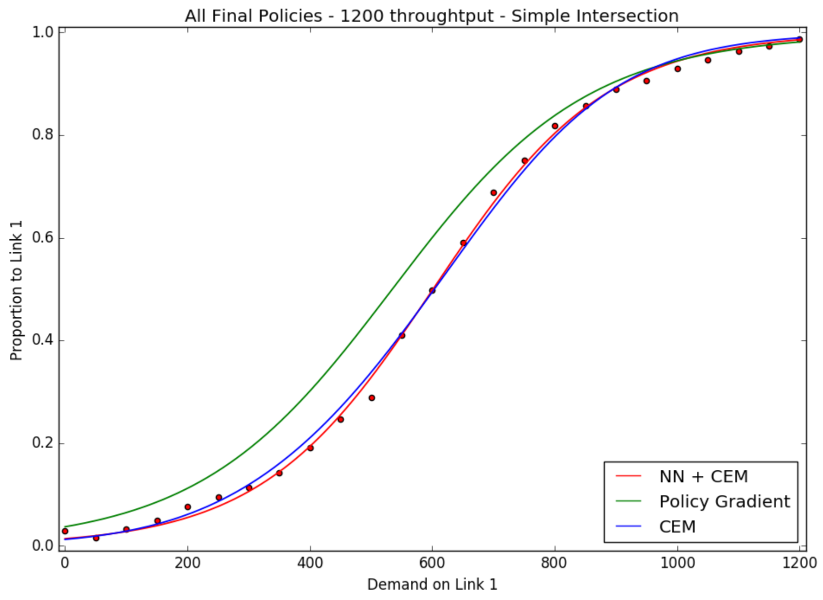


Figure 10: Results of our methods in the Simple Intersection scenario. Red points are the discrete CEM estimations for each particular value of demand.

By observing the figure 10, we can infer some common properties to all policies:

- It gives more green time to the link having the largest demand.
- When the demand in one link is zero, (almost) the entire cycle time is given to the other link.

- The curve is (approximately) symmetric with respect the middle point of 600 cars demand. This is expected since the traffic scenario itself is symmetric with respect the two links (all the traffic conditions are the same for both links).

We can conclude that the solution found has some interesting properties, however not unexpected, which make a case that a reasonable solution has been found in all cases.

Comparing the solutions, we observe that both CEM-based methods return almost identical policies, which is expected since they are based on the same method, while the Policy Gradient method returns a slightly different policy (although conserving the same shape). The difference can be attributed to being different local minima of the reward function, or the learning rate was too big around the optimal value. Quantitatively, each policy has corresponds to the following reward function:

Table 2: Performance Comparison of Policies - Simple Intersection

	Pure CEM	**Discrete CEM**	**Policy Gradient**	**Proportional-timed**	**Fixed-timed**
Reward Function	10.1	10.2	10.6	11.3	15.0

In the above table, results have a standard deviation of ± 0.1 approximately. We also computed the performance the simple controllers such as the Fixed-timed (equal division of times) and the Proportional-timed (green times divided proportionally to demands). In both cases, our models have performed better, although the improvement is only slight. This is probably because of the simplicity of the case we are dealing with.

Finally, we compare the running time of the algorithms. The CEM method is the most expensive method, taking between 200 and 300 second per iteration and between 20 to 30 iterations to converge. When using the discrete CEM approach, each state optimization only took from 2 to 10 seconds per iteration, and less than 10 iterations for each state estimation to converge. As for the Policy Gradient Method, each iteration consumed 14.5 seconds (less than 0.1 second of variance across iterations), and around 20 iterations to converge.

6.2 Complex Intersection Scenario

In this traffic network, we set the split ratios as the following: 60% of the vehicles drive straight, 20% make a left turn and the remaining vehicles make a right turn.

6.2.1 Simulation Parameters

For the CEM method, we have sampled 100 parameters per iteration, choosing the top 20% results. The initial mean μ_0 was chosen randomly, each coordinate picked uniformly between -1 and 1, and σ_0^2 was the identity matrix. When estimating the reward, 25 samples where averaged, and the reward function was evaluated with 100 state samples.

In the discrete CEM method, demands ranged from 0 to 1200 in intervals of 120, while maintaining a constant throughput of 1200. For each individual simulation we have used the same set of parameters as in the single intersection scenario. As for the Neural Network training, we used the same architecture as before, but we trained for less epochs, only 100, to avoid over-fitting problems.

In the policy gradient case, 50 iteration were run, with the concentration parameter being fixed to $\kappa = 100$. 500 samples where used when estimating the reward function's gradient, by using a learning rate of $\alpha = 0.005$.

In all cases we have used the same Neural Network architecture (hidden layers, activation functions, number of neurons) as in the Single Intersection case.

6.2.2 Discussion of Results

Due to the high dimensionality of this situation, we cannot plot the final policy as before. We restrict ourselves to comparing the reward function of each final policy and their computation time.

The following table compares the performance of each policy we obtained:

Table 3: Performance Comparison of Policies - Complex Intersection

	Pure CEM	**Discrete CEM**	**Policy Gradient**	**Proportional-timed**	**Fixed-timed**
Reward Function	10.0	5.1	5.8	5.2	7.9

The pure CEM method ran for 41 iterations, taking more than 10 hours this process. We decided to finish the simulation because the computation time has vastly higher that the other methods (but a better result could have been found if we let it run for more hours).

All the above results are exact up to a variation of ± 0.1. We see that the Policy Gradient and the Discrete Cross-Entropy methods have similar performances, and the results are better than the fixed-timed controller. However, in this case the Proportional-timed controller performs as good as those methods. One possible explanation for this is that since we have more phases, the cars end up distributing themselves more through the lanes, making accumulation more difficult to happen. Also, we have set 20% of the vehicles to make right turns, so they are not constrained to the phases movement and leave the network as soon as they can, accentuating this effect.

In terms of computational time, we have similar observations as before. Again, the pure CEM method is the most expensive one, taking around 500s per iteration. However, we are also taking the double of samples as before per iteration, which can explain this higher increase in time. In the discrete CEM method, each optimal green time computation took between 5 and 20 second, but the number of such computations to be done grows exponentially with the dimension of the state space. As for the Policy Gradient, each iteration took only 27 seconds in average, being the quickest one to train (only 50 iterations to reach a good solution).

7 Conclusion and Future Work

As far as our experiments are concerned, we showed that Neural Networks and Reinforcement Learning techniques can be successfully employed in the training of intelligent traffic controllers, at least in the case of single intersections. The results obtained by both methods are consistent with each other and both show that an adaptive controller can be better suited (or at least equally performing) than fixed-time and proportional-timed ones. We also verified that the Policy Gradient method is a quicker method to train comparatively to the CEM methods.

In future works, many working paths can be sought. One path would be to integrate more parameters such as turning rates and vehicle occupations in the state variable, making the agent more adaptive. Another possibility is to study how our methods perform in the case of multiple intersections, maybe using the single intersection as starting point and them improving the results with policy gradient. Yet another point of study is how our algorithm can be improved by substituting the gradient descent step by other methods of stochastic optimization, such as Adagrad, RMSProp and Adam, which are considered to be insensitive to change in its hyper-parameters (no learning rate to tune in principle). Finally, one could see obtain real traffic data of intersections and develop prediction models for the demand/turning rate of vehicles in a daily basis, which can after be integrated with our model to come up with real, intelligent traffic lights controllers.

References

[1] Rosenblatt, F. *The Perceptron–a perceiving and recognizing automaton.* Report 85-460-1, Cornell Aeronautical Laboratory, 1957.

[2] Cybenko, G. *Approximations by superpositions of sigmoidal functions.* Mathematics of Control, Signals, and Systems, 2 (4), 303-314, 1989.

[3] Hornik, K. *Approximation Capabilities of Multilayer Feedforward Networks* Neural Networks, 4(2), 251–257, 1991.

[4] Nielsen, M. *Neural Networks and Deep Leaning* `http://neuralnetworksanddeeplearning.com/`, Online book

[5] Hornik, K. Grun, B. *movMF: An R Package for Fitting Mixtures of von Mises-Fisher Distributions* `https://cran.r-project.org/web/packages/movMF/vignettes/movMF.pdf`

[6] *Theano: A Python framework for fast computation of mathematical expressions* arXiv:1605.02688 [cs.SC]

[7] Srinivasan, D. Chee Choy, M. *Neural Networks for Real-Time Traffic Signal Control* IEEE Transactions On Intelligent Transportation Systems, Vol. 7, No. 3, September 2006

[8] Denise de Oliveira et al. *Reinforcement Learning-based Control of Traffic Lights in Non-stationary Environments: A Case Study in a Microscopic Simulator*

[9] Yit Kwong Chin et al. *Q-Learning Based Traffic Optimization in Management of Signal Timing Plan*

[10] *Mastering the game of Go with deep neural networks and tree search* Nature, No. 529, Pages 484–489, January 2016

[11] Kar, A. *Stock Prediction using Artificial Neural Networks*

[12] Krizhevsky, A. Sutskever, I. Hinton, I. *ImageNet Classification with Deep Convolutional Neural Network*

[13] David Silver et al. *Human-level control through deep reinforcement learning* Nature, No. 518, Pages 529–533, February 2015

[14] Riedmiller, M. Gabel, T. Hafner, R. *Reinforcement learning for robot soccer* Springer Science+Business Media, LLC 2009

A Traffic model XML example

Here we show an example of the XML model format for the traffic network in figure 1.

Listing 1: Simple traffic intersection example

```
<scenario>
  <network>
    <nodes>
      <node id="0" /> <!-- Control node  -->
      <node id="1" />
      <node id="2" />
      <node id="3" />
      <node id="4" />
    </nodes>

    <links>
      <link id="1"  length="1" start_node_id="1"  end_node_id="0"  roadgeom="
          1" roadparam="1" />
      <link id="2"  length="1" start_node_id="2"  end_node_id="0"  roadgeom="
          1" roadparam="1" />
      <link id="3"  length="1" start_node_id="0"  end_node_id="3"  roadgeom="
          1" roadparam="1" />
      <link id="4"  length="1" start_node_id="0"  end_node_id="4"  roadgeom="
          1" roadparam="1" />
    </links>
  </network>

  <roadgeoms>
    <roadgeom id="1" type="arterial" lanes="2"/>
  </roadgeoms>

  <roadparams>
    <roadparam id="1" capacity="1800" speed="80" jam_density="100" />
  </roadparams>

  <roadconnections>
    <roadconnection in_link="1" in_link_lanes="1-2" out_link="3"
        out_link_lanes="1-2" />
    <roadconnection in_link="1" in_link_lanes="2"   out_link="4"
        out_link_lanes="2"   />
    <roadconnection in_link="2" in_link_lanes="2"   out_link="3"
        out_link_lanes="2"   />
    <roadconnection in_link="2" in_link_lanes="1-2" out_link="4"
        out_link_lanes="1-2" />
  </roadconnections>

  <splits>
    <split_node node_id="0" link_in="1" >
      <split link_out="3">0.8</split>
      <split link_out="4">0.2</split>
    </split_node>
    <split_node node_id="0" link_in="2" >
```

```
      <split link_out="3">0.2</split>
      <split link_out="4">0.8</split>
    </split_node>
  </splits>

  <demands>
    <demand link_id="1" start_time="0" dt="30">100,200,300,400</demand>
    <demand link_id="2" start_time="0" dt="30">100</demand>
  </demands>
</scenario>
```