UNIVERSITÉ LIBRE DE BRUXELLES
Faculty of Sciences
Department of Computer Science

Deep Reinforcement Learning for Autonomous Vehicle Control among Human Drivers

Manon Legrand

Supervisor:
Prof. Ann Nowé
Promoter:
Prof. Tom Lenaerts

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Academic year 2016-2017

Acknowledgments

First, I would like to thank my supervisor Ann Nowé for giving me the opportunity to do this work. I would like to extend this acknowledgment to the entirety of the COMO lab at the Vrije Universiteit Brussel who accomodated my presence.

I would like to thank especially my advisors, Roxana Rădulescu and Diederik Roijers, who helped, encouraged and guided me throughout the whole project and without whom I would not have been able to finish it.

Thanks to the members of the jury: Prof. Tom Lenaerts, Prof. Maarten Jansen and Prof. Hugues Bersini for evaluating this master thesis.

I would also like to thank the Université libre de Bruxelles for 5 fulfilling years; thanks for all the interesting classes and thanks for the people I got to meet.

Thanks to my family for always supporting me, especially my mother who first ignited my interest in computer science.

I would like to thank my friends, and Mégane in particular who spent a considerable amount of time helping me proofread this work – sometimes at the expense of her own sanity, and mine.

Finally, thanks to the clusters for their precarious stability and for apparently liking my jobs so much they would rather keep them to themselves. Readers be advised: many jobs were injured or killed during the making of this thesis. May they rest in peace.

Abstract

Traffic is a recurrent problem domain for multi-agent systems: much research in this area focuses on the traffic lights as learning agents. Another possibility to model such problems is to consider the cars themselves as the learning agents; thus dealing with autonomous cars. These self-driving cars are actually a popular research domain in AI. In this thesis, we consider autonomous cars as agents learning to drive safely. For this, we create a traffic simulator – including fixed agents, human drivers – that serves as the learning environment. We investigate whether deep reinforcement learning, i.e. the use of large neural networks as function approximators for reinforcement learning, can train efficient self-driving cars. We apply deep reinforcement learning in both single- and multi-agent settings to try and identify what kind of neural networks – simple feedforward networks or convolutional networks – perform well and how the single-agent models can be used to improve the multi-agent learning. Finally, our results are two-fold. First, we show that, while some models can train acceptably efficient drivers in a single-agent setting, their trained behavior does not necessarily scale up well to a multi-agent setting. Second, we show that models – initially designed for single-agent – are difficult to train in a multi-agent setting; however, using neural networks previously trained in a single-agent setting as the starting model perform better than when learning from scratch. All in all, training autonomous cars is feasible.

Contents

Chapter 1

Introduction

Artificial Intelligence (AI) is a scientific field that has gained a tremendous popularity outside the scientific community as the idea of intelligent machines ignites people's imagination. Even though the portrayal of AI in popular culture, e.g. in films such as *Her*[1], *Ex Machina*[2] or the *Terminator*[3] saga, often involves robots developing feelings or taking over the world, AI is, in its current state, more concerned with solving smaller specific tasks such as image recognition or cancer diagnosis from X-rays and MRIs; these *narrow* applications are miles away from the common idea of "human-like artificial beings" that people almost automatically think of when talking about AI.

Although imitating humans often leads, in fiction, to dramatic situations, there are useful human traits to imitate. Namely, *learning*. The *machine learning* field of AI aims to make machines learn from data in order to make predictions or decisions. As there is data everywhere, even more so now in our increasingly connected world, artificial intelligence – and machine learning – has become an industry in its own right.

A major subfield of AI focuses on creating artificial *intelligent agents*. They differ from simple predictive models in the sense that these agents interact with an environment that they can observe and in which they act, usually to reach a predefined goal. To achieve their goal, these agents can *learn* in different ways. One of these approaches is called *reinforcement learning* and addresses tasks of a sequential decision-making nature. Some examples of such tasks are board games (e.g. chess), and robot control (deciding where to go next). These problems are modeled as an agent who learns through repeated interactions with its environment and the feedback signal of these interactions; it learns through experience. This type of learning is probably the most intuitive one from a human point of view: we continuously learn from our mistakes.

The aforementioned learning framework deals with one learning agent only. However, several real-world decision problems, that are inherently composed of several key

[1] `http://www.imdb.com/title/tt1798709/?ref_=nv_sr_1`
[2] `http://www.imdb.com/title/tt0470752/?ref_=nv_sr_1`
[3] `http://www.imdb.com/title/tt0088247/?ref_=nv_sr_2`

constituents (e.g. robot teams, traffic), require models with multiple agents. These *multi-agent* models are thus more suited for these kinds of problems and can simplify problem solving by dividing knowledge among the agents.

One such problem is *traffic*. This problem domain is popular as it is both naturally suited for multi-agents (cars, traffic lights) and a growing problem in everyday life (congestion). Congestion is an important issue to be addressed; it increases travel time, and the consequent pollution and CO_2 emissions are harmful for the environment. Improving traffic signals can help solve this problem and several multi-agent works focus on it.

One particular application of artificial intelligence in the traffic domain is self-driving cars. In fact, they have received a lot of attention in the past few years and one of their advantages could be less traffic congestion because of their decreased need of a safety distance as well as a higher speed. Huge names such as Google[4], Uber[5] and Tesla[6] have been working on autonomous car technology, and prototypes are already on our roads[7]. Tesla's CEO, Elon Musk, even said that *"to do autonomous driving to the degree that is much safer than a person is much easier than people think"*[8]. Such an assertion raises important questions: **"Can a self-driving car really behave better than a human driver?"** and **"How can we achieve this?"**. The work presented in this thesis is an attempt to answer these questions. We try to see how good self-driving cars can be by applying deep reinforcement learning to train them: first in a single-agent setting, among humans only, and then in a multi-agent setting, with other autonomous cars.

1.1 Problem Statement

As stated earlier, the goal of this work is to make cars learn how to drive correctly. By correctly we mean *safely*, without crashing, and with a *purpose*, a goal (i.e. a destination).

To do that, we first create a traffic simulator that will serve as the learning environment. This traffic simulator consists of a highway composed of straight lanes. Moreover, we define fixed agents to represent human drivers. Their behavior is based on a order of preference for the possible actions they can choose. This order of preference tries to represent accurately how real-life drivers behave.

We want to see how well an autonomous car can learn to drive with deep reinforcement learning. We first consider a single-agent setting: the autonomous car is the only

[4] `https://www.google.com/selfdrivingcar/`

[5] `https://www.uber.com/en-BE/`

[6] `https://www.tesla.com/autopilot`

[7] As of now, these cars still require a person behind the wheel; fully autonomous cars are not yet permitted on public roads.

[8] `http://gizmodo.com/elon-musk-describes-the-future-of-self-driving-cars-1692076449`

learning agent on the highway, with human drivers. We use and compare different neural networks, simple ones then more complex ones with convolutional layers. We thus use deep Q-learning with the neural networks as Q-function approximators.

We then proceed to a multi-agent setting where there are multiple learning agents on the highway. For this, we use the models initially designed for the single-agent setting and extensions of these models with additional information regarding the other learning agents.

In short, we apply single-agent and multi-agent deep reinforcement learning to the problem of autonomous cars.

1.2 Structure of the Thesis

This thesis is divided in five parts. Part I introduces and explains the theoretical background on which our work is based. In Part II, we present the traffic simulator that we implemented; our agents, the autonomous cars, are trained in this environment. Part III then explains how these agents were created, as well as the different models used to train them. Part IV presents our results. Finally, Part V concludes this thesis with a discussion and ideas for future work.

Part I

Background

Chapter 2

Single-Agent Reinforcement Learning

In this chapter, we introduce the notion of agents and the single-agent learning framework. We first define agents and reinforcement learning separately then proceed to explain how reinforcement learning is applied; we present the mathematical tool used to represent these problems – the Markov decision process – and what reinforcement learning is trying to achieve: finding an optimal policy. We then present a popular RL technique: Q-learning; the exploration-exploitation trade-off problem, typical for reinforcement learning algorithms; and, finally, we explain how function approximation can be used to estimates Q-values and deal with the problems encountered when dealing with very large or continuous state-action spaces.

2.1 Agents

Establishing a formal definition for an agent is a rather controversial subject within the scientific community. This debate comes from the fact that many attributes of an agent have varying importance depending on the domain to which we are applying it. It is however possible to give *some* definition of an agent: "*it is a computer system situated in some environment, and capable of autonomous action in this environment in order to meet its design objectives*" (Wooldridge and Jennings, 1995). Even though this definition dismisses the possibility of human agents, they can still be considered if needed. These human agents can interact with the environment in the same way *non-human* agents would, but our concern with them is different. Namely, we can wonder what roles they are playing in the system, if they make decisions, if they are guiding other agents, or other questions that can arise from the system.

From the given definition, we can observe two important notions. First, an agent has to be *autonomous*; it means that it must be able to act on its own, without human

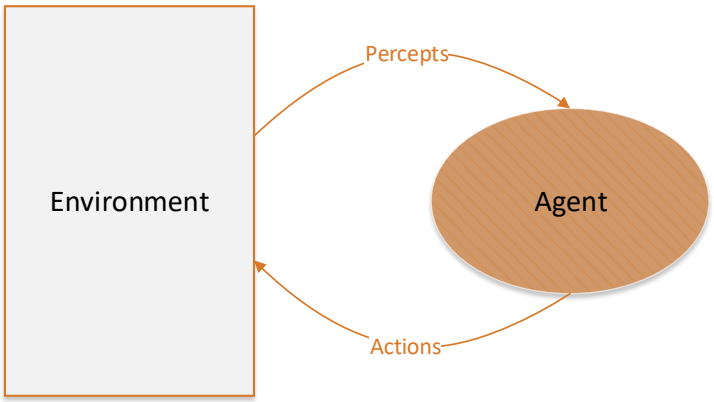


Figure 2.1: An agent and its environment

intervention. Second, an agent has *design objectives*, *goals* that it tries to reach. How these objectives can be achieved makes for different types of agents. Indeed, we can distinguish between two key characteristics of an agent: its *reactivity*, its ability to perceive the environment and to respond to changes in it, and its *proactivity*, its ability to take initiatives and act towards its goal. Agents differ by the degree of importance we give to each of these two characteristics.

Finally, we consider another attribute of agents: their *rationality*. Rationality is the assumption that an agent will try to achieve its goal and thus act accordingly.

2.2 Reinforcement Learning

In machine learning, we distinguish between three types of learning: *supervised*, *unsupervised* and *reinforcement* learning. The distinction is usually made by the feedback the agent receives.

Supervised learning is the task of learning a function from a *labeled* data set. By labeled, we mean that the data consists of training examples – the inputs – and their wanted output value. The problem can be a *classification*, if the output is a class, or a *regression*, if the output is one or multiple real values. In both cases, the feedback is the correct output that corresponds to the given input. A fully supervised trained model serves as a function that we can then use to map new inputs. The goal is to be able to label correctly unseen data: examples that were not in the training set.

Unsupervised learning is similar to supervised learning, except that the training data is "unlabeled". In other words, it must learn while not receiving any feedback. In this case, an unsupervised learning technique can be applied to identify, for example, different groups of types of data (i.e. clustering).

In between these two cases stands reinforcement learning where the feedback exists

but does not indicate whether the action taken was the right one. After acting, the agent gets a feedback, an immediate *reward* that can be either positive or negative. It is up to the agent, and thus the learning algorithm, to use and interpret this reward.

Reinforcement learning can be seen as a trial and error approach. As the agent is not explicitly told which action to take, it can only evaluate the actions with the rewards it received. For this evaluation to be efficient, the agent has to continually interact with the environment and adapt its strategy with regard to the rewards it gets. The immediate reward depends on both the action taken and the state the agent is in. This is summarized by Figure 2.2.

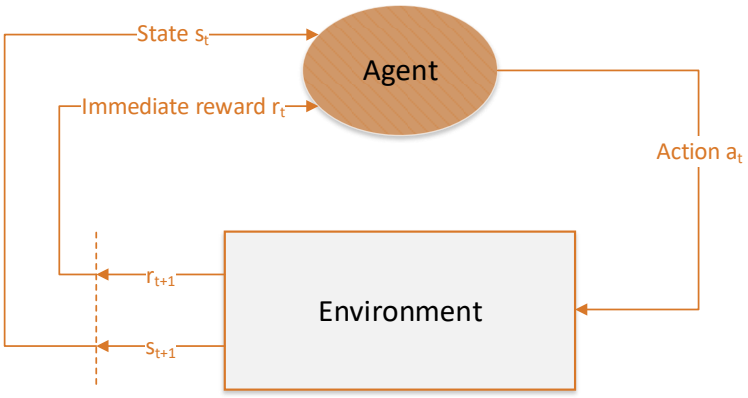


Figure 2.2: Reinforcement learning schema (adapted from Sutton and Barto (1998))

2.3 Markov Decision Process

The environment of the Single-Agent Reinforcement Learning case is formalized as a *Markov decision process* (Puterman, 1994), subsequently referred to as MDP.

Definition 2.3.1. *A finite Markov decision process is a tuple* (S, A, T, R, γ) *where*

- S *is a finite set of states,*
- A *is a finite set of actions,*
- $T : S \times A \times S \to [0, 1]$ *is the state transition probability function,*
- $R : S \times A \times S \to \mathbb{R}$ *is the reward function,*
- $\gamma \in [0, 1]$ *is the discount factor.*

At some time step k, the environment is described by the state $s_k \in S$. The agent performs an action $a_k \in A$ that causes a change in the environment such that it is in a new state $s_{k+1} \in S$, according to T; the probability to end up in s_{k+1} when performing

the action a_k while being in the state s_k is given by $T(s_k, a_k, s_{k+1})$. The reward that the agent gets is denoted by r_{k+1} and is given by $R(s_k, a_k, s_{k+1})$. It corresponds to the immediate effect of the action a_k.

The state transitions of a Markov decision process satisfy the *Markov property*: the next state s' is determined *only* by the current state s and the chosen action a. Consequently, the reward function R can be simplified as $R : S \times A \to \mathbb{R}$.

An agent behaves according to a policy, some mapping from states to actions. A policy is a function $\pi : S \to A$ that specifies the action a that the agent should take when it is in state s. When the policy only depends on the current state s, it is called *stationary*. The goal is to find a policy π that maximizes some function of the rewards. In our work, this function is the expected discounted sum of the rewards over a potentially infinite horizon.

$$\sum_{k=0}^{\infty} \gamma^k R(s_k, a_k) \tag{2.1}$$

Where a_k is given by $\pi(s_k)$; a_k is the action that the agent in the state s_k takes by following its policy π. The discount factor γ represents the difference in importance between rewards perceived in the future and immediate rewards. Thus, when γ equals 1, every reward has the same importance. Otherwise, the farther in the future a reward is, the less important it is. For infinite horizons, γ must be in $[0, 1)$ (i.e. < 1).

In other words, the agent has to maximize its long-term performance but only receives feedback (r_{k+1}) about its immediate performance.

2.4 Optimal Policy

An optimal policy π^* is a policy that maximizes the long-term performance. Hence, the agent must find such an optimal policy.

From the policy π's definition, we can define the value of a state s under that policy π, V^π (Wiering and van Otterlo, 2012):

$$V^\pi(s) = \mathbb{E}_h\Big[\sum_{j=0}^{\infty} \gamma^j R(s_j, a_j) | s_t = s\Big] \tag{2.2}$$

Where $\mathbb{E}_h$ denotes the expected return of the agent when it follows the policy π, starting from the state s.

We can define, in a similar way, the value of taking an action a in state s under a policy π, denoted $Q^\pi(s, a)$:

$$Q^\pi(s, a) = \mathbb{E}_h\Big[\sum_{j=0}^{\infty} \gamma^j R(s_j, a_j) | s_t = s, a_t = a\Big] \tag{2.3}$$

One way of computing an optimal policy π^* is by computing an optimal value function V^*; it can be a state-value function (Equation 2.2) or an action-value function (Equation 2.3). The algorithm is named *value iteration* and is shown in Algorithm 1.

Algorithm 1: Value iteration

Initialization of $V_0^*(s) = 0 \forall s$
for $i = 1, ..., \infty$ **do**
  foreach *state* $s \in S$, *given* V_i^* **do**
    /* Value update */
    $V_{i+1}^*(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_i^*(s')]$
  end
end

Where $V_i^*(s)$ is the expected sum of accumulated rewards when we start from state s and act *optimally* for i steps. V converges to the optimal value function V^*, and we can then find an optimal policy π^*.

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} T(s, a, s') V^*(s') \tag{2.4}$$

The argmax function of Equation 2.4 may return more than one value; it means that the agent can take different actions and still act optimally. We thus need to define some strategy to decide which action to take when there is more than one choice. A simple strategy – and the one we will use – is by taking the first optimal action in the set of optimal actions.

2.5 Q-Learning

This value iteration algorithm uses explicitly the state transition probability function T and the reward function R of the MDP. However, it is usually assumed that the *model*, which consists of knowledge of T and R, is unknown. In this case, we distinguish between two approaches. *Model-based* algorithms attempt to learn the model and use the estimate of the model to compute an optimal policy while *model-free* methods focus on learning the state value function and use these estimates to get an optimal policy. Such methods are generally known as *temporal difference methods* (Sutton, 1988).

The *Q-learning* algorithm (Watkins, 1989) is one of the most popular reinforcement learning techniques. It is a *model-free* value iteration algorithm.

We define the action-value function, or Q-function, $Q^\pi : S \times A \rightarrow \mathbb{R}$ as the expected return of a state-action pair given by the policy π (as defined in Equation 2.1). The

optimal Q-function is then defined as $Q^*(s,a) = \max_\pi Q^\pi(s,a)$.

$$Q^*(s,a) = \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')] \qquad \forall s \in S, a \in A. \tag{2.5}$$

The idea behind this algorithm is to transform the Q^* formula given in Equation 2.5 into an iterative approximation procedure. At each step, we update the current estimate of Q^* by using estimates of the optimal future value.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{2.6}$$

Where:

- $Q(s,a)$ is the estimate for the action-value function of the pair s, a
- $\alpha \in [0,1]$ is the learning rate that determines how "quickly" the new information will override the old information,
- $r + \gamma \max_{a'} Q(s',a')$ is the learned value, which is composed of the immediate one-step reward and the estimate of the optimal future value.

As we can see, Q-learning uses *temporal difference*; the difference between the new estimate and the current one.

The Q-learning algorithm is explained in Algorithm 2.

Algorithm 2: Q-learning

```
Arbitrary initialization of Q(s, a)
repeat
    /* For each episode */
    Initialize s
    repeat
        /* For each step of episode */
        Choose and perform action a according to some policy (see section 2.6)
        Observe reward r and new state s'
        Q(s, a) ← Q(s, a) + α[r + γ max_{a'} Q(s', a') − Q(s, a)]
        s ← s'
    until Termination condition;
until;
```

Q-learning is an *off-policy* method: the agent learns the value function for the optimal policy but acts according to another policy. *On-policy* methods mean the agent learns the value function of the policy dictating its behavior.

2.6 Exploration-Exploitation Trade-off

As stated earlier, the agent follows some policy that dictates its actions. We should then ask ourselves how the choice of this policy influence the Q-learning algorithm. In Walkins and Dayan (1992), it is proven that the Q-values converge to the optimal Q^* if two conditions are met. One of them states that *every state-action pair has to be visited infinitely often*. Consequently, the agent's policy needs to respect this condition. For this, we could simply use a policy where the actions are always chosen randomly with a uniform distribution over the action space. However, the global performance of the agent during the learning phase will be poor; we want it to maximize its return. These two opposite behaviors are called *exploration* and *exploitation*.

On the one hand, *exploitation* denotes the behavior of agents that select an action because it is the "best" – supposed to yield the highest reward – action. This action is dictated by the optimal, or *greedy* policy. On the other hand, *exploration* is when the actions all have the same probability of being chosen by the agents. They literally explore all their possibilities. It is easy to understand that using exclusively one of these behaviors is not efficient.

Only exploration means that, at some point, the agent has correct estimates for all state-action pairs but never uses this knowlege to maximize its rewards. It kind of defeats the whole point of learning. Without exploration, the agent can achieve an optimal reward only if it already has correct state-action estimates. However, such estimates can only be achieved through exploration. For example, if we initialize all the estimates to 0, a pure-exploitation agent will choose one action and stick to this action if the reward is positive. If the reward is negative, it proceeds with another action and eventually ends up in the same situation. Imagine a system where all the rewards are positive. The first (randomly) chosen action gives the smallest possible reward; when in this state, the agent always chooses this action because its estimate is greater than the others (0). In the end, the agent is very far from the maximum reward it could have obtained if its estimates were more accurate.

It is thus important to find a balance between these two extremes. Strategies that can solve this problem exist: for example, the ϵ-greedy strategy where the optimal action a^* is selected with a probability $1 - \epsilon$ and a random action with a probability ϵ.

$$a_t^* = \arg\max_a Q_t(a) \tag{2.7}$$

$$a_t = \begin{cases} a_t^* & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{2.8}$$

The problem with ϵ-greedy is that non-optimal actions are all considered the same during exploration. We would like to give an action a probability to be chosen that

translates to its current estimate. One way to do that is by using a *soft-max* action selection method. Such methods commonly use a Gibbs or Boltzmann distribution. We then choose action a at time step t with a probability

$$p(a_t) = \frac{e^{Q_t(a)/\tau}}{\sum_{a'} e^{Q_t(a')/\tau}} \tag{2.9}$$

Where τ is the computational temperature. High temperatures cause the actions to be all nearly equiprobable. The smaller the temperature is, the more the estimate Q_t impacts the probability; the greedy action will consequently be more probable.

Finally, we could consider that exploration should be less important as time passes. Indeed, exploration is more useful early on when we need to make good estimates of the Q-values. It is reasonable to think that, after some time, we will have explored enough and should start exploiting as we are fairly confident that our Q-values estimates are reliable. Defining ϵ or τ such that they become lower over time, usually with the use of a decaying rate, makes it possible.

In this work, we use ϵ-greedy as it its both efficient and easy to implement.

2.7 Function Approximation

A substantial drawback of Q-learning is that storing the Q-values of every state-action pair becomes incredibly difficult or even impossible when the state space is very large or continuous. Moreover, the convergence's condition we tackled earlier, which states that all the state-action pairs must be visited infinitely often by the agent, becomes very difficult to respect as well. In other words, there is never enough training data in these cases.

One solution is to use a *function approximation* that approximates and generalizes the Q-values across the states instead of computing a value for each state-action pair. We thus build a function $\hat{Q}(s, a)$ which approximates the real Q-value function $Q(s, a)$. This is a parametrized function whose parameters must be learned from experience.

The problem with function approximation is that the learning algorithms that use it are not theoretically guaranteed to converge. However, Q-learning used with linear regression has been proven to converge (Sutton et al., 2008) and empirically good results can be obtained with function approximation. For a comprehensive analysis of the convergence properties of Q-learning combined with function approximation, one can refer to Melo et al. (2008).

Chapter 3

Artificial Neural Networks

Possible function approximators include neural networks; hence, we present them in this chapter, starting with a little historical context. We then explain a simple, common structure for neural networks and how they can be trained. Afterwards, we present a more complex type of neural networks called convolutional neural networks. Finally, we clearly define how neural networks are used with reinforcement learning, which is known as deep reinforcement learning.

3.1 History

Neural networks are computational models consisting of a large set of connected units called *artificial neurons*. They actually date back to the 1940s where models based on neurology were developed (McCulloch and Pitts, 1943), but the research field went through a stagnation period; we had to wait until the late 1970s to see its re-emergence.

Neural networks' structure and the logic behind it are inspired from biological brains. Figure 3.1 shows how two biological neurons are connected to each other; a cell sends an information, a signal, to another cell via a specialized connection called synapse. An artificial neuron sends information to neurons to which it is connected in a similar way. As one neuron receives several signals at the same time (from different neurons), it is the combination of these signals that will determine whether the neuron gets activated; in that case, the signal will go through the neuron and travel to other neurons connected to it.

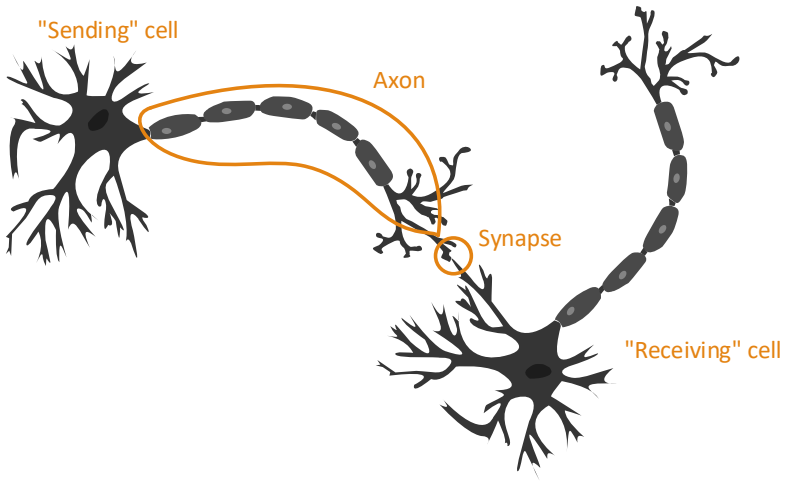


Figure 3.1: Schema of two connected neurons

3.2 Perceptron

The Perceptron (Rosenblatt, 1958) was one of the first artificial neural networks. It was initially designed for image recognition.

Nowadays, the perceptron refers to an algorithm for supervised learning of binary classifiers. Binary classifiers are functions that decides whether or not some input belongs to a specific class; the output is 1 if the input belongs to the class, and 0 otherwise. To compute this output, the perceptron has a list of weights $\mathbf{w}$ and a bias b that define a linear function.

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Where b is the bias, $\mathbf{w}$ is the weight vector, $\mathbf{x}$ is the input vector, and $\mathbf{w} \cdot \mathbf{x}$ is the dot product, $\sum_{i=1}^{n} w_i x_i$, with n the number of inputs. Thus, this function is a threshold: an input above this threshold will be classified as a positive instance. Likewise, an input below this threshold will be classified as a negative instance.

When talking about neural networks, a perceptron is an artificial neuron. Such neurons are analogous to first order logic operations. Figure 3.2 shows a perceptron when n equals 2.

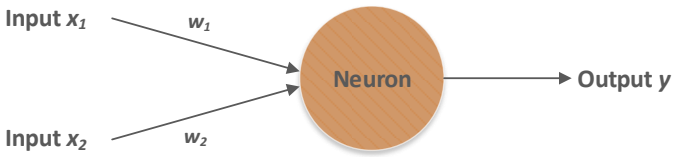


Figure 3.2: Perceptron with $n = 2$ inputs

Let us take for example the logic operator AND, whose truth table is presented in Table 3.1.

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.1: AND truth table

We can easily find values of w_1, w_2 and b such that the output of Equation 3.1 corresponds to the AND logic operator. For example, with $w_1 = 1$, $w_2 = 1$ and $b = -1$, we have:

$$f(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 + x_2 - 1 > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

Figure 3.3 shows the AND logic operation in a 2-dimensional graph with the classification's separation line defined by the perceptron.

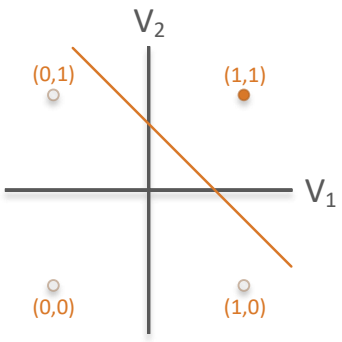


Figure 3.3: Logical AND's classification

Algorithm 3 shows the learning algorithm of the perceptron (Aizerman et al., 1964). The idea is to update the weight vector for every data point whose classification is incorrect.

Algorithm 3: Perceptron training

For a target function f

Initialize $\mathbf{w}$ as a vector of 0s

while $\exists$ *misclassified points* **do**

  Pick a misclassified point $\mathbf{x}_n$

  /* Update weight vector */

  $\mathbf{w} \leftarrow \mathbf{w} + y_n \mathbf{x}_n$

end

When it was first introduced, the perceptron seemed promising but its limits were quickly discovered. In Minsky and Papert (1969), the authors showed that the perceptron was incapable of learning the logical XOR function; Table 3.2 shows the truth table of the XOR operator and Figure 3.4 shows the 2-dimensional graph representation of this function. The reason why the perceptron cannot classify the XOR function correctly is because the data are *not linearly separable*. This constitutes the convergence condition of the algorithm presented in Algorithm 3; the perceptron learning algorithm will converge if the data points are linearly separable.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.2: XOR truth table

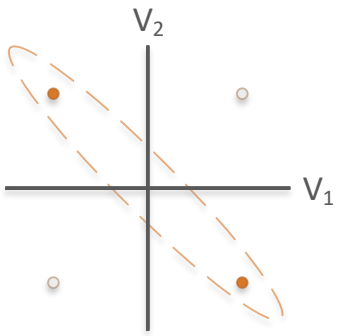


Figure 3.4: Logical XOR's classification problem

3.3 Feedforward Neural Networks

3.3.1 Structure

A perceptron is a neural network in its simplest form: a network with only one layer and one neuron. We explained earlier that this structure is limited as it is only able to learn linearly separable patterns. This *single layer* perceptron can be extended to *multilayer* perceptrons; networks composed of two or more layers of connected perceptron units, that we will refer to as simply *neurons* from now on. These networks are also called *feedforward* networks because there are no cycles; the input signals go through the network from the first (input) layer to the last (output) one directly. If there are other layers between the input and output ones, they are called *hidden layers*. Figure 3.5 shows such a structure.

Neurons from a layer l_i receive signals from neurons of the previous layer l_{i-1} and transmit these signals to neurons of the next layer l_{i+1}.

The signal that a neuron j gets is the weighted sum of the input neurons to which the neuron applies some function f, called an *activation function*. The resulting value is the output y_j of that neuron.

$$y_j = f\Big(\sum_i w_{ij} x_i\Big) \tag{3.3}$$

Where x_i is the i^{th} input neuron and w_{ij} the weight of the connection between the input neuron i and the output neuron j.

For the perceptron (Equation 3.1), the activation function is a binary threshold, but other functions are usually preferred. Namely:

- The *rectified linear unit* $ReLU(x) = \max(0, x)$
- The *sigmoid* function $\sigma(x) = \frac{1}{1+e^{-x}}$

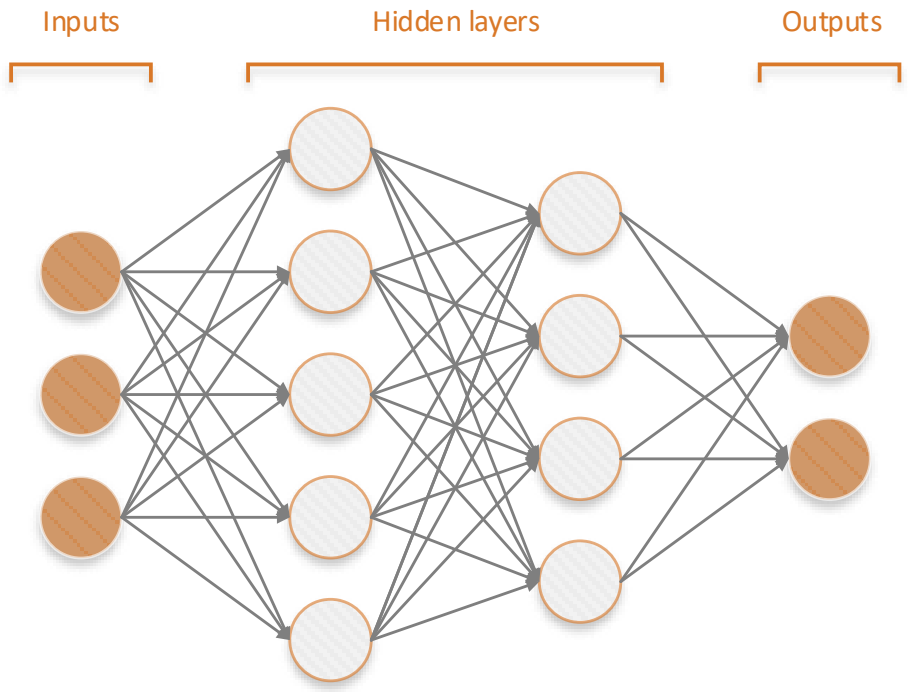


Figure 3.5: Structure of a feedforward neural network

- The *hyperbolic tangent* function $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

With this structure, any function can be approximated by some neural network (Rumelhart et al., 1986). It is thus possible to implement a network capable of solving the XOR problem we presented earlier. Indeed, the XOR logical function (see Table 3.2 for the truth table) can be achieved by using only AND ($\wedge$), OR ($\vee$) and NOT ($\neg$) logic operators:

$$A \text{ XOR } B = (\neg A \wedge B) \vee (A \wedge \neg B) \tag{3.4}$$

As a single unit perceptron can classify these three logic operators, a multilayer perceptron combining such units should be able to classify correctly the XOR function.

We still need to train this network; this is done by adjusting the weights.

3.3.2 Training

To evaluate how precise the output values $\mathbf{y}$ we get from the network are, we define the *loss* or *error* function, denoted E, as a function that represents the difference between our outputs and the target ones, denoted $\hat{\mathbf{y}}$. A common error function is the *mean*

squared error function, denoted MSE. Note that this supposes that the target outputs $\hat{\mathbf{y}}$ are known.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{3.5}$$

The goal is thus to minimize this error value. This is done by computing the gradient of E with respect to the weights and then using this value to update the weights. This optimization method is the *gradient descent*.

$$e_i = \frac{\partial E}{\partial w_i} \tag{3.6}$$

Unfortunately, for neural networks with a complex or large structure, computing the error function can be difficult. The *backpropagation* algorithm (Werbos (1974), Benvenuto and Piazza (1992)) solves this issue by computing the error function and updating the weights in a single backwards pass. This idea, shown in Algorithm 4, consists in computing an error for the output neurons then applying it to the network from the output layer to the input layer. The values that travel the network in this reverse order are used as error signals to update the weights.

Algorithm 4: Backpropagation

```
Initialize randomly all weights w_ij
repeat
    foreach training example do
        Compute all y_j   // Forward
        Compute all e_j   // Backward
        w_ij ← w_ij − αe_j   // Weight update
    end
until Termination condition;
```

The combination of the backpropagation algorithm and the gradient descent is considered a standard for training neural networks.

Online and Batch Training

Online training means that we are constantly updating the network's weights. For every input we get, it passes through the network and the backpropagation algorithm is then used to update the weights according to the gradient computed for this output.

In *batch* training, we consider subsets of the training data, called minibatches. We compute a gradient for the entirety of a subset and we then apply gradient descent through backpropagation to the network with this gradient. Thus, the difference with online training is that, since we compute the gradient of a subset, the weights are constant for all the elements of that subset.

In both cases, the network is trained iteratively and the gradient descent used at every training "step" thus becomes incremental; this is called *stochastic gradient descent*.

3.3.3 Dropout

Although large networks are powerful, they are slow to update and prone to *overfitting*. Overfitting occurs when the learning model (here, the network) fits noises and errors. Consequently, it diminishes the predictive performance of the network. A solution to this issue is the use of *dropout* (Srivastava et al., 2014).

The idea consists in dropping out neurons; with some probability p, we temporarily remove a neuron, with all its incoming and outgoing connections, from the network. Figure 3.6 shows the same network as Figure 3.5 after applying dropout to the two hidden layers.

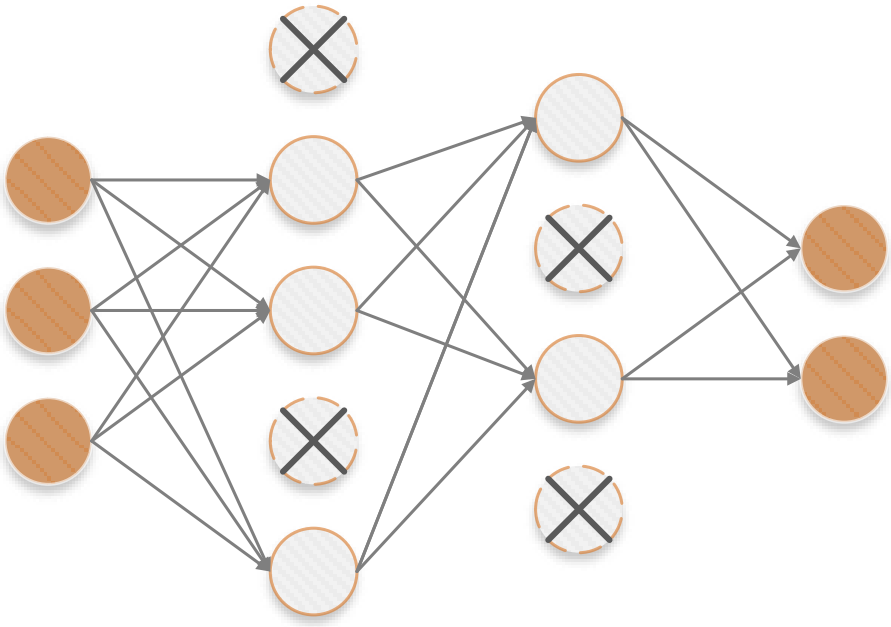


Figure 3.6: Dropout applied to the hidden layers

Dropout is only applied during training. At test time, the weights of a neuron unit whose probability of being dropped out is p are multiplied by p.

3.4 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of feedforward neural network. These networks were inspired by biology; the visual cortex is composed of cells that are sensi-

tive to small sub-regions of the visual field, called receptive fields. The receptive fields of different neurons are tiled and cover the visual field (Hubel and Wiesel, 1968). The response of a neuron to a stimulus in its receptive field is similar to a mathematical convolution operation. CNNs are designed to take advantage of the 2-dimensional structure of an input (e.g. an image).

CNNs are widely used, and effective, in fields such as classification, image recognition and natural language processing.

3.4.1 Convolution

Convolution is a form of mathematical operation on two matrices: some input matrix and a kernel, also called mask or filter. The kernel is a small (square) matrix that we "apply" on each element of the input. To do that, we first have to center the kernel on the element, then multiply each element of the kernel with the corresponding element in the input matrix. The center element's new value is the sum of these products. The concept of convolution is shown in Figure 3.7.

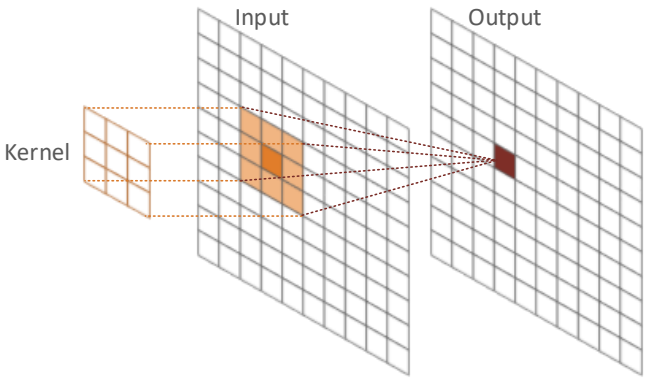


Figure 3.7: Image convolution schema

When the size of the kernel is even, it is not possible to center it exactly on the elements of the input matrix. In such cases, we usually add a new row and a new column with zero values to make the kernel's size odd. Where to add them – top or bottom for the row, left or right for the column – depends on where we want the center of the kernel to be.

Given a kernel K of size n ($n \times n$ matrix), we define k as $\lfloor \frac{n}{2} \rfloor$, the maximum distance from the center of the kernel to the edge, and we can compute the output value of the

element at (x, y) in the input matrix I with the following equation.

$$O_{x,y} = \sum_{i=-k}^{k} \sum_{j=-k}^{k} K_{i+k,j+k} \; I_{x+i,y+j} \tag{3.7}$$

This 2-dimensional definition can easily be extended to N-dimensional matrices.

$$O_{x_1,\ldots,x_N} = \sum_{i_1=-k_1}^{k_1} \sum_{i_2=-k_2}^{k_2} \ldots \sum_{i_N=-k_N}^{k_N} K_{i_1+k_1,i_2+k_2,\ldots,i_N+k_N} \; I_{x_1+i_1,x_2+i_2,\ldots,x_N+i_N} \tag{3.8}$$

3.4.2 Convolutional Layers

A CNN contains at least one *convolutional layer*: a layer whose parameters are learnable kernels, or filters. Each filter is convolved across the input (Figure 3.7) and the resulting output is called an activation or feature map. The full output of the layers is obtained by stacking all the activation maps. Figure 3.8 shows how the input neurons processed by the filter are mapped to the activation map.

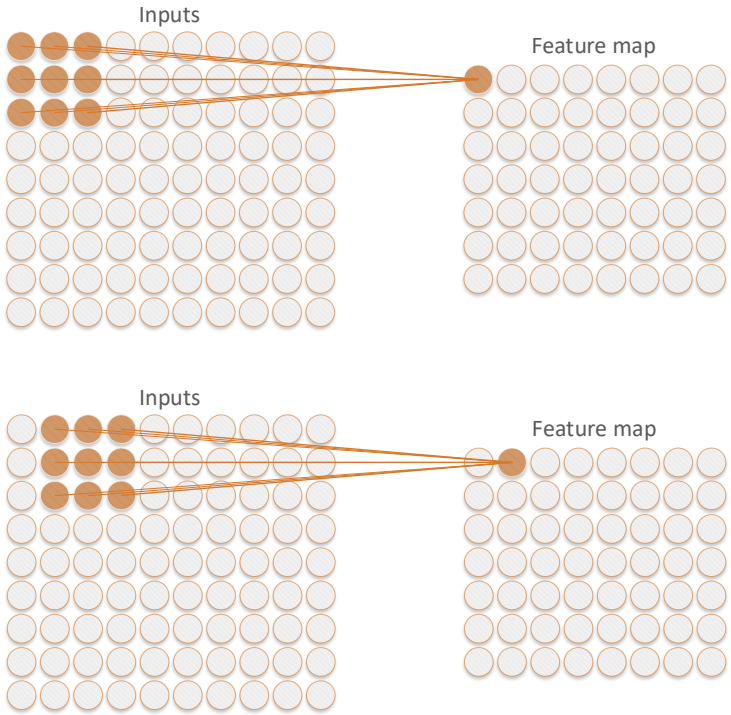


Figure 3.8: 3×3 convolution and corresponding feature (adapted from Nielsen (2013)). We slide the kernel by one neuron to the right, and so on until all input neurons have been processed.

There are important properties of a convolutional layer that influence how the convolution is done.

First, the *stride* of the layer is a value that specifies how to slide the filter along the input neurons. This is the step of the convolution operation. If each neuron is processed as the center of the filter, as suggested in Figure 3.8, then the stride is equal to 1. If, for example, the stride is equal to 2, then we would have slid the filter by 2 neurons. Consequently, the value of the stride impacts the size of the output.

Second, the way of dealing with the edges of the input also impacts the size of the output. The problem is that the filter cannot process neurons on the edge, as centering the filter to these neurons would mean leaving some of the filter's values outside the bounds of the input. One solution to this problem is to *crop* the input; we do not consider the neurons on the edges. This is the case in Figure 3.8. Another solution is to add rows/columns of 0 so that the output size of the convolution remains the same as the input one. This is called *zero-padding*.

3.4.3 Other Layers

Rectified Linear Units

We previously defined the rectified linear unit $ReLU(x) = \max(0, x)$ as an activation function for neurons. We can extend this to create a full ReLU layer, where all the neurons have this activation function. By placing an activation layer after the convolutional layer, we increase the nonlinearity. Other activation functions such as the sigmoid could be used, but it has been shown that ReLU speeds up the training of the neural network (Krizhevsky et al., 2012).

Pooling Layers

Usually placed right after the convolutional layers, pooling layers aim to decrease the size of the feature maps by condensing them – for example, summarizing a region of $n \times n$ neurons. A common procedure is the *max-pooling* one where the pooling unit outputs the maximum activation value in the region.

The concept of pooling is also called down-sampling, as it makes the network lose information. However, by reducing the volume, it also reduces the amount of computation.

3.5 Deep Reinforcement Learning

Deep learning is defined as "a class of machine learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification" (Deng and Yu, 2014). Other definitions are given but we can observe the same two key concepts across all these definitions: deep learning uses multiple layers of nonlinear processing units and

is based on the supervised or unsupervised learning of feature extractions in each layer, with a hierarchy from low-level to high-level features in the layers. As such, complex neural networks fit in this definition.

Deep Reinforcement Learning refers to the usage of deep learning in a reinforcement learning setting.

3.5.1 Deep Q-Networks

The deep reinforcement learning field's popularity started with the introduction of Deep Q-Networks (DQNs) (Mnih et al., 2015). DQNs are deep learning models that combine deep convolutional neural networks with the Q-learning algorithm. Before this, using a nonlinear function such as a neural network as a function approximator for the Q-function was known to be unstable. An important contribution made by DQNs is the use of experience replay and a target network to stabilize the training of the Q action-value function approximation with deep neural networks. Q-network thus refers to a neural network function approximator of the Q-function:

$$Q(s, a; \theta) \approx Q^*(s, a) \tag{3.9}$$

Where θ is the weights of the network.

3.5.2 Experience Replay

Catastrophic forgetting or *catastrophic interference* (Mccloskey and Cohen (1989), Ratcliff (1990)) denotes the tendency of neural networks to suddenly forget information that they previously learned. Fortunately, there are ways to overcome this issue (Kirkpatrick et al., 2016).

Experience replay (Lin, 1992) consists in maintaining a buffer of the old experiences and train the network against them.

During the training, we use a buffer where previous experiences are stored. An experience consists of an observed state and action pair, the immediate reward obtained and the next state observed. When the buffer is full, new experiences usually replace the oldest ones in the buffer.

To train the network, we sample a batch of experiences from the buffer and use them to apply the usual backpropagation algorithm.

The advantage of this approach is that, by sampling these batches of experiences, we break the correlation between data that we get when the network is trained in the usual online fashion.

3.5.3 Deep Q-Learning

The deep Q-learning algorithm uses experience replay. An agent's experience at a time step t is denoted by e_t and is a tuple (s_t, a_t, r_t, s_{t+1}) consisting of the current state s_t, the chosen action a_t, the reward r_t and the next state s_{t+1}. The experiences for all the time steps are stored in a *replay memory*, over many episodes. We then apply minibatches updates to samples of experiences. A simplified version of the full algorithm (Mnih et al., 2013) is shown in Algorithm 5.

Algorithm 5: Deep Q-learning

Initialize replay memory $\mathcal{D}$ of length N
Initialize action-value function Q with random weights
for *episode* $= 1, ..., M$ **do**
  Initialize state s_1
  for $t = 1, ..., T$ **do**
    Select action a_t according to an ϵ-greedy policy
    Perform action a_t and observe reward r_t and next state s_{t+1}
    Store experience (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch of experiences (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{if state } s_{t+1} \text{ is final} \\ r_j + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{if state } s_{t+1} \text{ is not final} \end{cases}$
    Perform a gradient descent step based on target output value y_j
  end
end

Since the network serves as an approximator for the Q-function, each output neuron of this network corresponds to one valid action, and every action is mapped to an output neuron. Thus, after a feedforward pass of that network, the outputs are the estimated Q-values of the state-action pair defined by the input and the output's corresponding action.

Chapter 4

Multi-Agent Systems

So far, we only talked about single-agent systems. However, our problem domain is traffic, which is clearly a problem composed of numerous entities (traffic lights, cars). Traffic is thus naturally modeled as a multi-agent system. In this chapter, we introduce theses systems and the mathematical tool used to formalize them: the Markov game. Finally, we briefly explain how reinforcement learning and deep reinforcement learning apply to these systems.

4.1 Introduction

A multi-agent system, subsequently referred to as MAS, is a system composed of numerous agents and the environment in which they interact (Wooldridge, 2002). Figure 4.1 represents such a system.

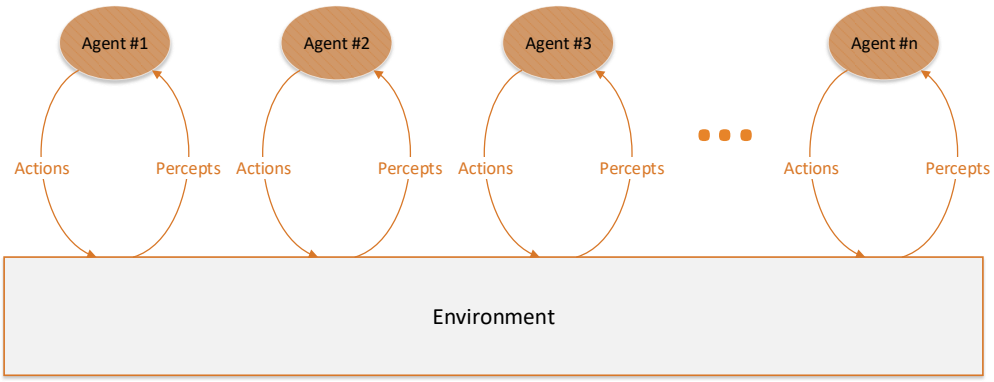


Figure 4.1: Abstract view of a Multi-Agent system (adapted from Russell and Norvig (1995))

The main advantage of multi-agent systems is that they can solve problems that single-agent systems cannot when these problems are inherently composed of numerous

entities. A MAS is essentially a distributed approach and is consequently the most evident way of looking at certain systems that are naturally distributed (e.g. robots teams, traffic light networks, electricity grids, etc.). A multi-agent system can also be an alternative for an initially centralized problem such as resource management and consequently congestion problems.

Complete surveys of multi-agent systems with a more detailed discussion of their applications and approaches to solve such problems can be found in the works of L. Busoniu (Busoniu et al., 2008) and L. Panait (Panait and Luke, 2005).

The complexity of multi-agent systems often means that we do not always know what action an agent should have taken; supervised learning is therefore not applicable often. We can, however, determine when an action was good, whether it led to an optimal situation or just a better one. This knowledge should help the learning of the agents and corresponds to the reinforcement learning approach. Moreover, single-agent reinforcement learning algorithms already exist and have good convergence and consistency properties. We are thus considering multi-agent reinforcement learning, referred to as MARL.

4.2 Markov Games

Multi-agent reinforcement learning cannot be solely captured by the same framework we use for the single-agent case. We need a more complex framework, which is the generalization of the Markov decision process to the multi-agent case; it is the *stochastic* or *Markov game* (Shapley (1953), Littman (1994)).

Definition 4.2.1. *A Markov game is a tuple* $\big(n, S, A_1, ..., A_n, T, R_1, ..., R_n\big)$ *where:*

- n *is the number of agents*
- $S = \{S^1, ..., S^N\}$ *is a finite set of environment states,*
- A_k *is the action set of player* k*,*
- $T : S \times A \times S \to [0, 1]$ *is the state transition probability function,*
- $R_k : S \times A \times S \to \mathbb{R}$ *is the reward function of player* k*.*

Where $A_k(s^i)$ denotes the set of actions available to agent k when it is in state i. Consequently, the reward function R_k and the transition probability T also depend on this state s^i, on the next state s^j, and on a joint action $\mathbf{a}^i = (a_1^i, ...a_n^i)$ from this state. The reward of agent k for their action in time step t is thus $r_{k,t+1} = R_k(s^i, \mathbf{a}^i, s^j)$. The expected return as we defined in the introduction changes in the same way; it is now also dependent on the joint action $\mathbf{a}^i$.

The main goal with Markov games is the same as with Markov decision processes: we want to optimize the future discounted reward (or the average reward over time). It can be done the same way as before, the only difference being that the criteria we used now depend on the policies h_i of the other agents. However, we are in a situation with multiple agents whose goals are not necessarily the same. As such, it is impossible to maximize the expected return of every agent when they have conflicting goals.

4.3 Multi-Agent Reinforcement Learning

The multi-agent reinforcement learning (MARL) field is quite recent, but has rapidly grown over the years, with various works presenting approaches often based on developments in single-agent reinforcement learning and game theory. MARL techniques consider three kinds of tasks: *fully cooperative* tasks when there is no conflict between the agents' goals, which is the case when they all have the same reward function, *fully competitive* tasks when the agents have completely conflicting goals (e.g. opposite reward functions), and *mixed* tasks when the setting is neither fully cooperative nor fully competitive: there is no constraint on the reward function. The type of tasks can be used to classify the different multi-agent reinforcement learning techniques.

We can also consider two main categories of MARL techniques depending on whether or not the agents observe one another. If not, they are called *independent learners* and basically ignore each other. This does not change the fact that other agents' actions could influence the environment, but it will be considered as noise. The principal advantage of this approach is that it is then really easy to apply single-agent reinforcement learning techniques. Moreover, the number of agents does not really affect the learning "deployment", making it easily scalable. Unfortunately, there are some drawbacks as well. Namely, the single-agent RL techniques can be used but lose their convergence guarantees and the agents cannot, as they ignore the others, settle for any kind of coordination, which would surely be beneficial in most systems. The other approach, where agents actually observe each other, is the *joint action learners* approach. The idea is a disadvantage in itself as observing other agents is costly and the complexity increases exponentially with the number of agents in the system. However, with this approach, coordination is possible.

A comprehensive survey of the different multi-agent reinforcement learning techniques is presented in Busoniu et al. (2008).

4.4 Deep Multi-Agent Reinforcement Learning

Although the field of deep reinforcement learning has been growing quickly in the recent years, its extension to multi-agents is still limited. One of the first works in this field

extends the deep Q-learning architecture (Mnih et al., 2015) to multi-agent environments with two agents (Tampuu et al., 2015). In the initial work (from DeepMind), they use arcade games as the learning environment for their (single) agent. However, most of these games allow multiple players.

One of the challenges faced in Egorov (2016) is the representation problem; how to represent an arbitrary number of agents without changing the structure of the deep Q-network. They propose a state reformulation that allows the system's states to be represented in an image-like fashion, effectively making convolution possible.

Other related works include the use of multi-agent deep reinforcement learning to make the agents learn communication protocols (Foerster et al., 2016) and learn co-operative policies in complex domains without explicit communication (Gupta et al., 2017).

Part II

Traffic Simulator

Chapter 5

Traffic Model

As we explained earlier, agents have to interact with an *environment*. Consequently, the first step of our project is to design and implement a traffic environment that we ideally want both *simple* and *realistic*. The setting we choose is a portion of highway; an arbitrary number of straight lanes with exits.

5.1 Level of Detail

An important criterion is the level of detail we want. We generally distinguish between three levels of detail: high, medium and low detail, classified as *microscopic*, *mesoscopic* and *macroscopic* models, respectively (Hoogendoorn and Bovy, 2001). Microscopic models focus on individual elements (e.g. cars). Macroscopic models describe traffic as an aggregation of characteristics of the system such as traffic flow, density and velocity. Finally, mesoscopic models stand in between the two other types of models and study traffic as (small) groups of traffic entities with low detail level description.

As the goal of this project is to study the behavior of *autonomous cars* – individual self-learning cars – in traffic, we opt for a microscopic simulation model. In other words, the traffic entities that our simulator consists of are drivers (and their cars) who make decisions based on their current situation on the highway.

5.2 Scale of the Variables

Another important criterion is the *scale* of the variables. A variable can either be *discrete* or *continuous*. We usually classify traffic models based on their time-scale; *discrete* models consider discrete time instants at which the environment state changes while *continuous* models describe continuous changes of the system's state in response to continuous stimuli. Other variables such as speed and position can be either discrete or continuous values. Mixed models, with both types of variables, exist.

We ultimately decided to implement a fully discrete model; even though a continuous model would be more realistic, we favored simplicity.

Chapter 6

Highway

The first key part of our environment is the highway. It is defined by its structure – that is, the lanes, the exits, the size of the lanes, etc – and by the cars driving on it. In this chapter, we thus present our simulator's highway and the dynamics in it: how the cars move and how crashes can happen.

6.1 Structure

The highway is mainly a spatial system in two dimensions: the lanes and the positions in theses lanes. As cars arrive on the highway and progress on it, we need to be able to clearly define where they are.

6.1.1 Car Position

The highway can be seen as a group of lanes and a lane can be seen as a group of cells. The position of a car on the highway is defined by the lane and the cell of that lane that it is in; this position is consequently equivalent to xy coordinates. To simplify the highway's visual representation, we define the lane index as the y coordinate and the cell index as the x coordinate. Figure 6.1 shows how the lanes and cells are indexed; the lanes are indexed from left (top in the figure) to right (bottom in the figure) and the cells are indexed from left to right.

6.1.2 Exits

The right-end lane of the highway is the exit lane, not shown in Figure 6.1. It is a lane where drivers go only to take an exit, if any. As soon as a car reaches an exit, it is considered out of the highway and consequently out of the environment/simulation. The same goes for cars that reach the end of a lane. An exit is indicated twice; right before the exit itself and once right before the previous exit. This is an attempt to mimic

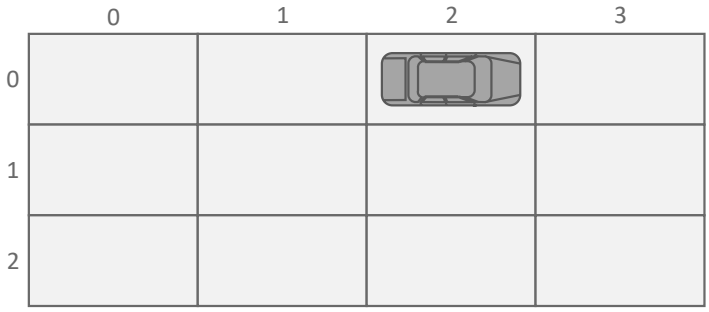


Figure 6.1: Car at $x = 2, y = 0$

real life where highway exits are indicated multiple times with the remaining distance to that exit.

The distance between two consecutive exits is always the same, as well as the distance between the start of the highway and the first exit. This is another parameter of the highway.

6.1.3 Cell Types

We distinguish between four types of cells: *car cells* where agents drive, *exit cells* where agents can leave the highway, exit *indication cells* that indicate the current and next exits, and *off-road cells*, cells on the exit lane that are not exits nor indications. Cars cannot go on these cells and movements passing by or going to these cells are never considered by the drivers.

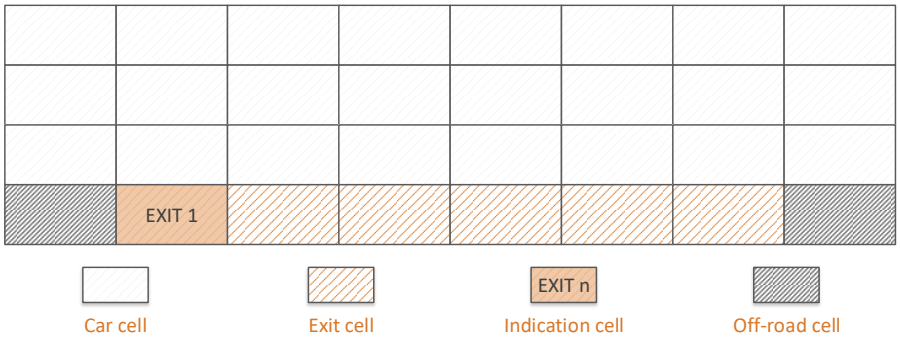


Figure 6.2: Different cell types

6.1.4 Highway Parameters

A highway H is defined by the following parameters:

- **Number of lanes,** Y: does not include the exit lane

- **Lane size,** X: number of cells in each lane
- **Number of exits,** E: in the exit lane, can be 0
- **Exit size,** S_e: number of cells of each exit
- **Space size,** S_s: number of cells between two consecutive exits

Each of these parameters can be independently modified for a simulation. However, coherency between some of them should be respected: for example, the maximum number of exits that can fit in the highway depends on the lane, exit and space sizes.

6.2 Cars

6.2.1 Speed and Acceleration

A car has two important attributes: its speed and its acceleration that can change at every time step t. The speed v_t and acceleration α_t are bounded; the speed cannot be negative or greater than the cars' maximum speed v^{max} and the acceleration cannot be greater, in absolute value, than the cars' maximum acceleration α^{max}. These attributes are discrete variables and can only take integer values.

$$0 \leq v_t \leq v^{max}, \quad v_t \in \mathbb{N} \tag{6.1}$$

$$-\alpha^{max} \leq \alpha_t \leq \alpha^{max}, \quad \alpha_t \in \mathbb{Z} \tag{6.2}$$

The speed at some time step t depends on both the speed and the acceleration of the previous time step $t-1$. Consequently, at each time step, the possible accelerations are values for which the updated speed will respect the speed boundaries. For example, when the speed $v_t = v^{max}$, the acceleration α_t can only be negative or equal to zero.

The speed update rule is

$$v_t = v_{t-1} + \alpha_{t-1} \tag{6.3}$$

6.2.2 Movements

At each time step, a car c's x position, denoted by $x_{c,t}$, is updated according to its speed $v_{c,t}$ and its acceleration $\alpha_{c,t}$ while its y position (i.e. the lane), denoted $y_{c,t}$, depends on their chosen direction, denoted $d_{c,t}$.

Regardless of the direction, the length $l_{c,t}$ of the car's move at the time step t is equal to the car's updated speed, $v_{c,t+1}$ of that time step and corresponds to the number of cells that it will advance.

$$l_t = v_{t+1} = v_t + \alpha_t \tag{6.4}$$

The x position is then updated as follows:

$$x_{c,t+1} = x_{c,t} + l_{c,t} \tag{6.5}$$

When the new speed v_{t+1} is equal to 0, the destination cell will be the initial position of the car at that time step.

The y position is updated according to the direction $d_{c,t}$. The car can only change lane when the movement length $l_{c,t}$ is greater than 0. Otherwise, the car is still and does not move.

$$y_{c,t+1} = \begin{cases} y_{c,t} - 1 & \text{if } d_{c,t} = \text{LEFT and } l_{c,t} > 0 \\ y_{c,t} + 1 & \text{if } d_{c,t} = \text{RIGHT and } l_{c,t} > 0 \\ y_{c,t} & \text{if } d_{c,t} = \text{FORWARD or } l_{c,t} = 0 \end{cases} \tag{6.6}$$

All directions are not always possible; when a car is in the left-end lane, it can only go forward or to the right, as turning left would lead outside the highway. Similarly, when it is on the right-end lane, it can only go forward or turn left unless it is taking an exit when going to the right.

Finally, we add an attribute to the car, the status of their blinkers, denoted $b_{c,t}$, that indicates which direction the car is taking:

$$b_{c,t} = \begin{cases} \text{LEFT} & \text{if } d_{c,t} = \text{LEFT and } l_{c,t} > 0 \\ \text{RIGHT} & \text{if } d_{c,t} = \text{RIGHT and } l_{c,t} > 0 \\ \text{null} & \text{if } d_{c,t} = \text{FORWARD or } l_{c,t} = 0 \end{cases} \tag{6.7}$$

6.3 Crashes

When considering two cars c_1 and c_2 and their respective initial position $(x_{c_1,t}, y_{c_1,t})$ and $(x_{c_2,t}, y_{c_2,t})$ and destination $(x_{c_1,t+1}, y_{c_1,t+1})$ and $(x_{c_2,t+1}, y_{c_2,t+1})$, we have to determine whether they are crossing paths at the same time step and crashing.

6.3.1 Passing Cells

When a car moves from the position (x_t, y_t) to the position (x_{t+1}, y_{t+1}) at some time step, it does not disappear from its initial position to appear in the destination afterwards. The car passes by intermediary positions or *passing cells*, denoted PC.

For the forward direction, these passing cells are easily defined as the cells located between the starting and destination cells of the move:

$$PC_{t+1} = \Big\{(x, y_t) \mid x \in [x_t + 1, x_{t+1} - 1]\Big\} \tag{6.8}$$

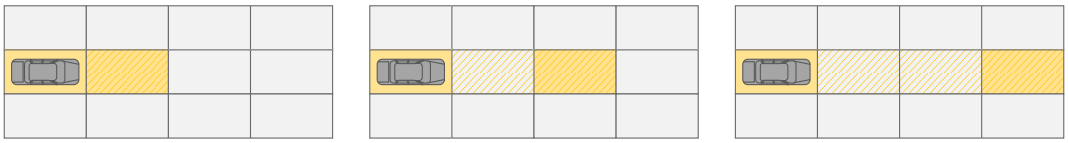
For the other directions, when the car changes lanes, we consider that the car effectively crosses the lane lines in the middle of the movement; at this moment, the car is

considered to be in both lanes. Before that, the car is still in its initial lane, and after that it is in the new lane. The passing cells are:

$$PC_{t+1} = \begin{array}{c} \left\{(x, y_t) \mid x \in [x_t + 1, x_t + \left\lfloor \frac{l_t + 1}{2} \right\rfloor\right\} \\ \bigcup \\ \left\{(x, y_{t+1}) \mid x \in [x_t + \left\lfloor \frac{l_t}{2} \right\rfloor, x_{t+1} - 1]\right\} \end{array} \tag{6.9}$$

Figure 6.3 and Figure 6.4 show the passing cells and the destination cell for all possible movements of speed 1, 2 and 3. Finally, Figure 6.5 shows the passing and destination cells of a lane change (direction right) when the speed is some unknown value; we can see the difference when this value is odd or even.

For a move of length l_t, the length of intermediary section of the highway in which the car passes is equal to $l_t - 1$. The middle of the move is considered to be the middle of that section, and we define it as the position (cell) where the number of cells to its left is equal to the number of cell to its right. This is easy when $l_t - 1$ is odd; for example, if $l_t - 1 = 5$, the middle is 3, with two values on the left (1, 2) and two values on the right (4, 5). On the other hand, when $l_t - 1$ is even, this is not possible when considering only integer values. We solve this by defining the middle as a group of cells; for example, if $l_t - 1 = 6$, the middle is the two values 3 and 4, with two values on the left (1, 2) and two values on the right (5, 6). This is formalized in Equation 6.9.



(a) $v_t = 1$, d_t = FORWARD (b) $v_t = 2$, d_t = FORWARD (c) $v_t = 3$, d_t = FORWARD

Figure 6.3: Going forward

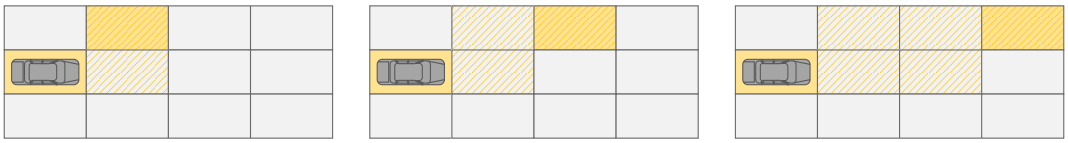
6.3.2 Detecting Crashes

Two cars crash when they are in or pass by the same cell during the same time step t. A car c_1 and a car c_2 will crash when

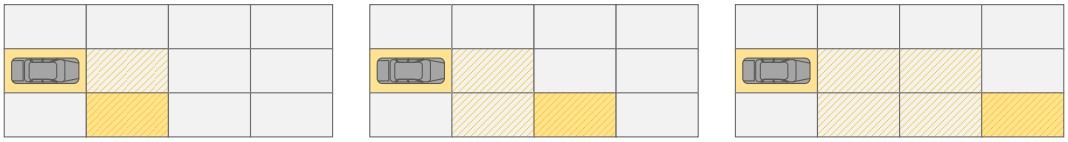
$$\left(PC_{c_1,t+1} \cup (x_{c_1,t+1}, y_{c_1,t+1})\right) \cap \left(PC_{c_2,t+1} \cup (x_{c_2,t+1}, y_{c_2,t+1})\right) \neq \emptyset \tag{6.10}$$

Figure 6.6 shows two examples of situations where cars cross paths and consequently crash.

In Figure 6.6a, the blue car is in $(1, 0)$ and is moving to $(0, 3)$; this corresponds to a move of speed 3 to the left. The green car is in $(2, 1)$ and moves to $(1, 3)$; this is

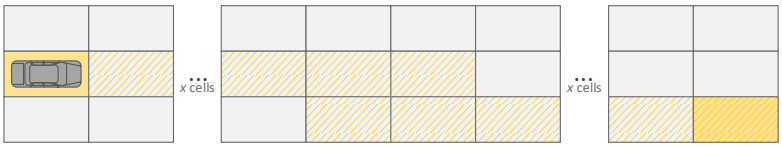


(a) $v_t = 1$, $d_t =$ LEFT (b) $v_t = 2$, $d_t =$ LEFT (c) $v_t = 3$, $d_t =$ LEFT

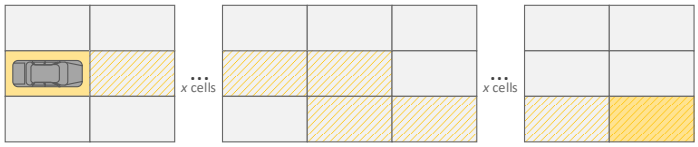


(d) $v_t = 1$, $d_t =$ RIGHT (e) $v_t = 2$, $d_t =$ RIGHT (f) $v_t = 3$, $d_t =$ RIGHT

Figure 6.4: Changing lanes



(a) Odd speed value



(b) Even speed value

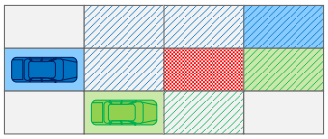
Figure 6.5: Changing lanes with an arbitrary speed

a move of speed 2 to the left. Their passing cells are $\{(1,1),(1,2),(0,1),(0,2)\}$ and $\{(2,2),(1,2)\}$ respectively. The cell at $(1,2)$ is passed by both cars. there is a crash at this position.

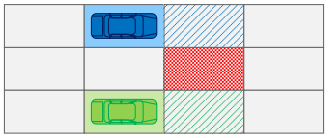
In Figure 6.6b, the blue car starts in $(0,1)$ and goes to $(1,2)$; it is a move of speed 1 to the right. The green car is in $(2,1)$ and goes to $(1,2)$; it is a move of speed 1 to the left. Their passing cells are $\{(0,2)\}$ and $\{(2,2)\}$ respectively. The cars do not pass by the same intermediary position but their destination is the same; they are going to crash.

Another example could be when a car's destination is one of another car's passing cells.

After a crash, the cars involved in the accident are removed from the simulation. To make the system more realistic, the accident is added in the cell in which it happened; it now becomes an obstacle for other cars. After a certain number of time steps, it will be removed from the highway, effectively clearing the path. This number of time steps



(a) First example



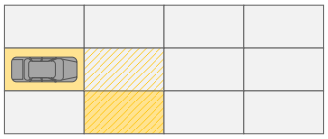
(b) Second example

Figure 6.6: Examples of crash situations

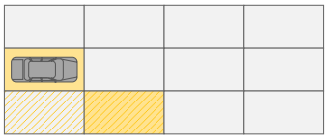
is another simulation parameter called the “crash duration”.

6.3.3 Particular Cases

As we can see in Figure 6.4a and Figure 6.4d, there is only one passing cell when the speed of a turning (left or right) move is 1. The choice of this cell seems arbitrary, as it could have as well been the one as shown in Figure 6.7b. At first glance, there is no difference between these two possibilities.



(a) First possibility



(b) Second possibility

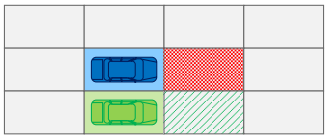
Figure 6.7: Alternative passing cell for a move of speed 1

Let us imagine a situation where there is a car in some lane, and another car at the same x position on the lane to its right. The first car wants to go to the right lane, while the second wants to go the left one. They both have a speed equal to 1. We can easily visualize this as a real-life situation; the cars would collide as they are trying to cross each other. Figure 6.8 represents this situation.

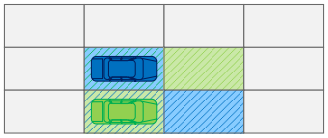
Figure 6.8a shows the passing cells of the cars when considering our choice. The crash is detected as the passing cell of a car is the destination of the other. There are actually

two positions where the crash can be; only one was shown here. The actual position of the crash in the simulator depends on the order in which the cells are updated.

Figure 6.8b shows the passing cells of the cars when considering the alternative possibility. In this case, the passing cell of a car corresponds to the starting position of the other. However, as the starting position of a move is not taken into account for the crash detection, this crash will not be detected.



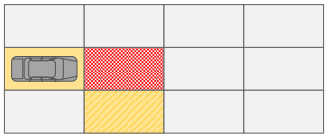
(a) Crash detected



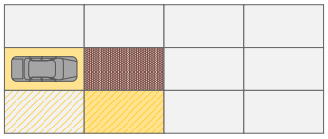
(b) Crash not detected

Figure 6.8: Cars crossing each other with a speed of 1

There are unfortunately still problems when cars change lanes with a speed of 1. Namely, when there is an obstacle – a stationary car or an accident – in the cell in front of them. Figure 6.9a and Figure 6.9b show what happens for the first and second possibility of the passing cell, respectively; in the first case, the car will collide with the obstacle and crash while it will manage to avoid it in the second case. In real life, a car should be able to easily avoid an unmoving obstacle, granted that it is not driving too fast.



(a) Crashing with the obstacle



(b) Avoiding the obstacle

Figure 6.9: Facing an obstacle

To get the desired behavior, detecting the crash and being able to avoid obstacles,

sticking to only one of these possibilities is not enough. We need to use both: by default, we use the first possibility and we switch to the second one only when facing an obstacle.

6.3.4 Limitations of the Crash Detection System

Since our environment uses only discrete variables, the time steps as we use them are the smallest possible time units. Consequently, when a car moves from a position to another and passes by intermediary positions, it goes in all these cells at the same time; it means that the car is in multiple cells at the same time. It may not seem evident and it actually yields somewhat counter-intuitive situations as shown in Figure 6.10. In this example, two cars are next to each other and both are going to the lane on their left. Their speed is equal. In real life, they would be able to do what is basically the same movement, at the same time. The reason is that, in a continuous environment, their position changes continuously and they always keep their initial y difference. However, in our discrete environment, the time step in which they move cannot be divided into smaller time units.

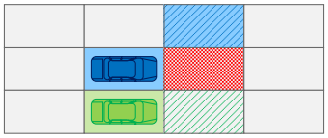


Figure 6.10: Non-evident crash

We could easily imagine a system made up of arbitrary rules deciding on the outcome of such situations. It would certainly help avoid seemingly unnatural crashes. However, we would have to define these rules for pre-estasblished scenarios whereas our system is generic for all possible movements. Namely, it works without actually knowing what the positions, speeds and directions are, and that is why we opted for this system.

6.4 Environment's States Definition

A state s of the highway is a list of all the cells of this highway, with information about what is in the cell at that moment or about what type of cell it is.

We denote $H_{x,y}$ the cell located at the x^{th} position on the y^{th} lane. For off-road and exit cells, the only information is the type of the cell. For exit indication cells, the information is the exits indicated: the number of this exit e_i and of the next one e_{i+1}. Finally, for car cells, the information can be one of three things. If there is a car, the information is the speed, acceleration, and blinkers' status of the car. If there is a crash, the information is simply an indication that there is a crash. If there is nothing, the information indicates as much. By unifying everything, we have:

Definition 6.4.1. *A highway cell $H_{x,y}$ is a tuple $(o, e, v, \alpha, b, i^1, i^2)$ where*

- *o indicates if the cell is an off-road cell (1) or not (0);*
- *e indicates if the cell is an exit cell (1) or not (0);*
- *v indicates the speed of the car currently in the cell, if any;*
- *α indicates the acceleration of the car currently in the cell, if any;*
- *b is the blinkers' status of the car currently in the cell, if any;*
- *i^1 is the number of the indicated exit if the cell is an indication cell;*
- *i^2 is the number of the next exit, if any, if the cell is an indication cell*

We can now formally define a state s at a time step t:

$$s_t = \left\{ H_{t,x,y} \mid 0 \leq x < X, 0 \leq y \leq Y \right\} \tag{6.11}$$

Where $H_{t,x,y}$ is the highway cell at the position x, y as defined in 6.4.1 for the time step t, X is the size of the lanes, and Y is the number of lanes.

Chapter 7

Human Drivers

We defined earlier the cars' attributes and how they move on the highway, but a car does not make decisions, the drivers do. We thus have to define these drivers. For now, the system we are implementing is merely a simplified, discrete, representation of an actual highway. Consequently, the drivers are humans and we define their behavior such that it mirrors as best as possible what can be observed in real life.

7.1 Attributes

As we are trying to make a realistic traffic system, we need to define some important attributes for the drivers that relate to real human characteristics.

First, human sight is undeniably one of the most important attributes; a blind person can obviously not drive. Drivers can only see for a limited distance in front of and behind them. In our environment, this limited distance translates into a number of cells. We make a distinction between the *sight*, what the drivers see in front of them, and the *backsight*, what they see behind them. As looking behind – through the rear- and side-view mirrors – while driving is less easy, we make the assumption that the backsight is smaller than the sight. The sight and backsight are denoted by ϕ^+ and ϕ^- respectively and we fix the value of ϕ^- such that $\phi^- = \phi^+ - 1$. We also consider that the drivers are able to see the entirety of the highway width; they can see all the lanes. We can now define a window of what a driver sees at each time step t. Given a driver c's current position $(x_{c,t}, y_{c,t})$ and their sight and backsight ϕ_c^+ and ϕ_c^-, the cells that are in the driver's field of view $\Phi_{c,t}$ at that time step are:

$$\Phi_{c,t} = \left\{ H_{x,y} \mid x_{c,t} - \phi_c^- \leq x \leq x_{c,t} + \phi_c^+, 0 \leq y \leq Y \right\} \tag{7.1}$$

Where Y is the number of lanes in the highway.

Figure 7.1 shows an example of a driver's field of view. The highlighted cells are the ones that the driver can see.

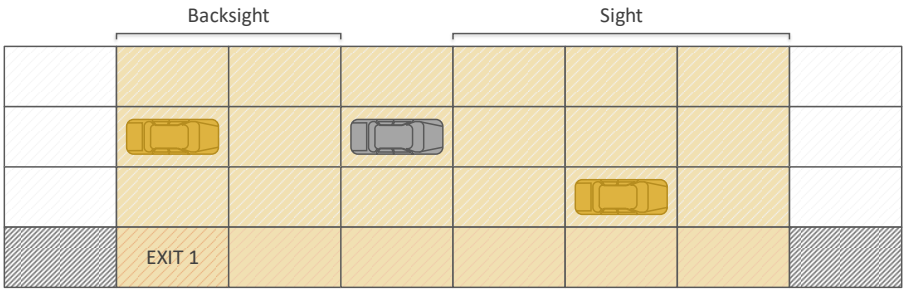


Figure 7.1: Driver's field of view

Second, the general behavior depends on what the driver wants. There are different types of people; some are careful, some are not. This mostly impacts their speed. Careful drivers will not drive too fast while reckless ones will. This characteristic is defined as a *desired speed*, the speed at which they would rather drive. It is denoted by v_c^*. The upper bound for this value is the same as for the speed, as shown in Equation 6.1. However, the lower bound cannot be equal to zero, as it would not make sense for a driver on a highway to *want* to stay still. Consequently, the lower bound is the smallest possible non-zero speed: 1.

$$1 \leq v_c^* \leq v^{max} \tag{7.2}$$

Finally, humans are rarely driving aimlessly; they are going *somewhere*. As their ultimate destination is outside the highway, they need to leave it by taking an exit. This exit is not necessarily in the system. In such cases, the goal of the drivers is to reach an exit that is beyond the highway limit of our environment; in our system, they thus need to reach that limit, the last cell of a lane, and continue on the highway. The driver's goal is denoted by ω_c and can be 0 if they want to continue on the highway, or a positive non-zero integer corresponding to the number of the exit they want to take.

$$\omega_c = \begin{cases} 0 & \text{to continue on the highway} \\ i, \quad i \in [1, E] & \text{to take the } i^{th} \text{ exit } e_i \end{cases} \tag{7.3}$$

7.2 Actions

We now have to define what the actions that the drivers can take are. We can easily see that one action should lead to one move, and vice versa; a particular move is the result of only one action. As explained in subsection 6.2.2, a move is a combination of a certain speed and a direction.

The driver can choose a direction directly as it corresponds to turning the wheel in a certain way. This direction can be left, right or forward. However, the driver cannot choose the speed they want to have; changing speed means accelerating or decelerating.

An action is consequently composed of a direction d_t and an acceleration α_t. This acceleration is bounded as stated in Equation 6.2. In conclusion, the set of possible actions A is:

$$A = \left\{ (d, \alpha) \mid d \in \{\text{LEFT}, \text{RIGHT}, \text{FORWARD}\}, -\alpha^{max} \leq \alpha \leq \alpha^{max} \right\} \tag{7.4}$$

At a given time step t, some actions can be "impossible". Namely, when an action's corresponding move is forbidden in our system, this action will not be considered by the driver. A forbidden move is a situation where the destination or one of the passing cells of that move is outside the highway; this includes the off-road cells of the exit lane. Moreover, actions whose acceleration will make the updated speed outside the speed bounds defined in Equation 6.1 are also considered impossible.

For example, when $\alpha^{max} = 1$, the set of actions is:

$$A = \left\{ \begin{array}{ll} (\text{LEFT}, & -1), \\ (\text{LEFT}, & 0), \\ (\text{LEFT}, & +1), \\ (\text{FORWARD}, & -1), \\ (\text{FORWARD}, & 0), \\ (\text{FORWARD}, & +1), \\ (\text{RIGHT}, & -1), \\ (\text{RIGHT}, & 0), \\ (\text{RIGHT}, & +1) \end{array} \right\} \tag{7.5}$$

Furthermore, the length of the action set depends on the number of directions, which is always 3, and the number of possible accelerations.

$$|A| = 3 \times (2\alpha^{max} + 1) \tag{7.6}$$

7.3 Behavior

Now that we have defined the attributes of a human driver, we can effectively create a behavior for them. Their behavior consists in a series of rules that establish an order of preference for all the possible actions. The driver will first and foremost try to avoid crashing, so they will not choose an action that will cause a crash, even though said action is the highest in their preference order. Beyond that, the driver's preferred actions should reflect their own goal as well as their attributes; a driver with a low desired speed will rather not drive too fast, and conversely, and a driver will act towards its goal, continuing on the highway or taking an exit.

We distinguish between two behaviors: a default one and the behavior of drivers that noticed the exit they want to take. In the second case, the behavior will translate

the need of the driver to get closer to their exit and to ultimately take it. The default behavior describes the usual attitude of the drivers.

In both cases, we need to define the actions' order of preference. This is done by giving to each action's direction and acceleration a preference order, then combining the two.

This way of defining and prioritizing some behaviors/actions is similar to a behavior-based approach for robots (Arkin, 1998).

7.3.1 Default Behavior

As we said previously, we want the default behavior to be some sort of usual behavior that we can observe with real-life human drivers. Namely, humans want to reach a certain speed; that is the driver's desired speed v_c^*. With this desired speed, we can define the preferred accelerations of the driver as the ones that will make their speed the closest possible to their desired speed. For example, if the driver's current speed v_c is equal to their desired speed v_c^*, their preferred acceleration will be 0 (they stay at their desired speed), then $\pm$ 1, $\pm$ 2, and so on. We can thus define their *target acceleration* α_c^* – the acceleration that will get them the closest to their desired speed – and establish the order of preference for the other accelerations based on it: $\alpha_c^*, \alpha_c^* + 1, \alpha_c^* - 1, \alpha_c^* + 2, \alpha_c^* - 2, ...$

Note that every $\alpha_c^* \pm i$ that is outside the acceleration bounds will not be in the preference order as it is not a possible acceleration. The same goes for accelerations that would make the driver's speed outside the speed bounds.

We now have to determine, for each acceleration the driver can have, what their preferred directions are. An important rule in traffic[1] is the speed difference from one lane to another; the more to the left a lane is, the faster the drivers on that lane should be. This comes from the fact that drivers overtake other cars from the left. As such, the order of preference for the directions should reflect this. To do that, we add a preferred speed attribute to the lanes; the speed that cars in this lane should preferentially be driving at. For any acceleration, the preferred directions will consequently make the driver go in the lane whose preferred speed is the closest to their own (new) speed. Note that we only consider directions whose destination lane exists and is not the exit lane. An action that leads to the exit lane when the driver's goal is not to take an exit will always be a least preferred move.

Lanes' Preferred Speed

To overtake another car, a driver must go to the lane on the left and drive faster. This leads to the observation that the average speed in a lane should be higher the farther to the left this lane is.

[1] Traffic refers to the one we are used to in Belgium.

The preferred speed of a lane between these two extremes will depend on both the number of lanes (excluding the exit lane) and the number of possible speeds (excluding 0). We denote the i^{th} lane L_i and its preferred speed v^{L_i}.

We want to guarantee a fair and balanced distribution of the preferred speeds across the lanes. The simplest case is when the number of lanes Y is a multiple of the number of possible speeds m. In such cases, each speed v^k will be the preferred speed of $n_{v^k} = \frac{Y}{m}$ lanes, ordered from highest (left-end lane) to lowest (right-end lane). For example, if $Y = 6$ and $m = 3$, the preferred speeds of the lanes, from left to right, will be 3, 3, 2, 2, 1, 1. When Y is not a multiple of m, n_{v^k} is not the same for all speeds v^k. Some speeds will be the preferred speeds of more lanes. We decided that these speeds should be the higher ones. For example, if $Y = 4$ and $m = 3$, the preferred speeds of the lanes will be 3, 3, 2, 1. The number of speeds that appear more times as a preferred speed is given by the rest of the division of Y by m.

With this construction, the preferred speeds' distribution will always respect the following conditions

$$v^{L_i} \geq v^{L_j} \quad \forall i \leq j \tag{7.7}$$

$$n_{v^k} = \begin{cases} \lfloor Y/m \rfloor & \text{if } k \leq (Y \mod m) \\ \lfloor Y/m \rfloor + 1 & \text{otherwise} \end{cases} \tag{7.8}$$

In conclusion, once we know how many times each speed should appear as a lane's preferred speed, we just need to distribute them correctly, by respecting the first condition.

Preferred Moves

Now that we know how to define the order of preference for the accelerations and directions, we can combine them to generate the preferred moves, or actions, of the drivers. We show the construction process of the list of preferred moves in Algorithm 6.

7.3.2 Taking an Exit

When the driver's goal is to take an exit whose indication is in their field of view, up to two things can happen.

If they see their exit indication, they will decrease their desired speed so that it becomes equal to the minimum speed and their behavior will change; they will now act according to a "taking an exit behavior" that we will explain shortly after.

If they only see the indication of the previous exit, the one before theirs, they will decrease their desired speed, if necessary, so that it becomes equal to the minimum speed +1 but their behavior remains the default one.

Algorithm 6: *GetPreferredMoves*

```
preferred moves ← empty list
preferred accelerations ← GetPreferredAccelerations()
forall acceleration in preferred accelerations do
    preferred directions ← GetPreferredDirections(acceleration)
    forall direction in preferred directions do
        add (direction, acceleration) in preferred moves
    end
end
if right lane is exit lane then
    forall acceleration in preferred accelerations do
        add (RIGHT, acceleration) in preferred moves
    end
end
```

In both cases, their decreased desired speed should make them preferentially go to the right and thus get closer to the exit lane.

The new behavior of the drivers that need to take an exit differs from the previous one in that the main component of the move is now the direction instead of the acceleration. Indeed, to take an exit, a driver must go to the right; this direction will consequently be their preferred one when they can take the exit. If they cannot take the exit with their next move directly, they should preferentially get closer to the exit lane – going forward if they are already next to the exit lane or going right otherwise – and avoid getting farther from it; that is, going left. They should also avoid going right if it means taking a different exit.

Once we know the driver's preference order for the directions, we need to order the accelerations. To do so, we consider the driver's preferred accelerations as defined by the default behavior and combine them with a given direction. We then give a priority to all these possible moves. The highest priority goes to moves that make the driver able to take their exit. A high priority is given to moves that make taking the exit easier; that is, getting closer to the exit or decelerating when the driver is already next to the exit. A low priority is given to moves that do the opposite. The lowest priority is given to moves that will make it impossible for the driver to take the exit in the future. Finally, a default, normal priority is given to the moves that do not meet the previous criteria. The final order of preference for the accelerations, given a certain direction, is thus obtained by re-ordering these moves from highest to lowest priority. Note that, by using the preferred accelerations of the default behavior as the base order, we make sure that if more than one acceleration has the same priority, the one that the driver

would prefer if they were following the default behavior will be considered before the other accelerations of that same priority level.

Preferred Moves

We can now, same as for the default behavior, construct the list of preferred moves. The algorithm is presented in Algorithm 7.

Algorithm 7: *GetPreferredMovesExit*

```
preferred moves ← empty list
preferred directions ← GetPreferredDirectionsExit()
forall direction in preferred directions do
    preferred accelerations ← GetPreferredAccelerationsExit(direction) forall
     acceleration in preferred accelerations do
        add (direction, acceleration) in preferred moves
    end
end
```

7.3.3 Choosing a Move

At each time step, the preference order of the moves as well as the possible moves of a driver will change, and they need to make a choice. They cannot automatically perform the highest move in their preference order as this move may cause a crash. As stated previously, the essential desire of the drivers is to not crash. They will not choose a move whose outcome is an accident. Instead, the drivers will choose the first move in their preference order that does not provoke a crash.

To make their decision, the drivers consequently have to know whether they will crash after a certain move. Their knowledge is based on what they see and is thus limited by their field of view. Moreover, they need information about the other drivers around them. We consider two things; first, they are able to evaluate the speed of the other cars and, second, they know if the cars in front of them are changing lanes (going left or right) thanks to the blinkers. The second assumption also means that the drivers consider that the cars behind them are going forward. These choices are motivated by our implementation; the drivers' observations are updated sequentially from right (end of highway) to left (start of highway). The drivers choose their move before the ones behind them. Consequently, the evaluated speed of the cars behind them does not necessarily correspond to the actual, final, speed of these cars, as the drivers can still accelerate or decelerate. In conclusion, to assess whether a move results in a crash, a driver has perfect information only about the situation in front of them, and limited information for the situation behind them.

Unfortunately, there may be situations where all the moves of a driver cause a crash and the driver can seemingly not avoid crashing. However, as a crash behind is assessed with limited, not necessarily exact, information, this crash could potentially be avoided depending on the action of the other driver. Thus, when all the moves lead to a crash, the driver should choose – when possible – a move that leads to no crash in front of them and that minimizes the number of crashed behind them. This way, they still have a chance to avoid crashing depending on the actions of the drivers behind them. If such a move does not exist, nothing can be done to avoid crashing, and the chosen action is the one that minimizes the number of crashes caused by the corresponding move.

The process of choosing an action is explained in Algorithm 8.

Algorithm 8: *ChooseAction*

```
move ← first preferred move according to the situation
best crashing move ← move
while move causes crash do
    move ← next preferred move
    if move causes fewer "front" crashes than best crashing move or fewer crashes
     in total then
        best crashing move ← move
    end
end
if move does not cause crash then
    choose move
else
    choose best crashing move
end
```

7.3.4 Irrationality

Everything we explained previously for the human drivers was based on the following assumption: human drivers are unerring. Unfortunately, this assumption is unrealistic. Humans are primarily irrational beings. To make our environment more similar to real life, we need to make our human drivers more human: irrational.

To do so, we add another attribute to the drivers, their *irrationality*. It is simply the probability that the driver chooses a random action instead of the one given by Algorithm 8. We denote it ι and it is the same for all the drivers.

Algorithm 9: *BeIrrational*

if *irrational, with probability* ι **then**
  action $\leftarrow$ random action
else
  action $\leftarrow$ *getAction*()
end

Chapter 8

Simulator

Now that we have defined all the basic components of our environment, we can use them to form a real traffic simulator. The first thing a traffic simulator needs is actual traffic, i.e., cars. Hence, we need to define how this traffic is generated. Once this is done, we can run our simulator to see how it performs.

8.1 Generating Traffic

We explained previously how the highway and the drivers are defined, as well as the dynamic on the highway. We now need to generate actual traffic. In other words, we need to define how drivers arrive in the highway and the attributes of these drivers.

How a driver arrives actually refers to when – which time step – and where – which lane – they arrive in the environment. To do so, we use a new parameter for the highway, the *traffic density*, denoted τ, that serves as the probability, at each time step and for each lane, that a new driver is arriving in this lane.

This new driver's attributes, their speed, desired speed, and goal, are chosen randomly. Their sight is also chosen randomly within a range that depends on the maximum speed. Note that the initial speed of a driver does not depend on the lane in which they arrive; we do not make assumptions about their behavior before entering the highway.

The traffic generation process is summarized in Algorithm 10.

Algorithm 10: *GenerateTraffic*

```
forall lane do
    if lane's initial position is free then
        if random < τ, with probability τ then
            driver ← randomly initialized driver
            driver enters lane
        end
    end
end
```

8.2 Highway Steps

We briefly mentioned earlier that the highway is updated sequentially from right to left. In other words, drivers that are farther on the highway make their decisions before the drivers behind them. Said decisions can then be observed by the other drivers that are behind and consequently impact their own decisions. To implement this, we divide a highway time step into two "sub-steps", that we call the *observation step* where the drivers observe the environment and choose an action, and the *update step* where we update the highway's state by executing the drivers' actions. The observation step is done sequentially from right to left. During this step, the drivers choose their action; if they are going to change lanes, their blinkers are turned on so that drivers behind them can know which direction they are going to take. After the observation step, all drivers have chosen their action, and we can now update the highway: this is the *update step*, which is done sequentially from left to right. We update the highway from left to right because we want the crashes that can happen in different cells (see section 6.3) to actually happen in the first possible cell – starting from the left. Algorithm 11 shows how a highway time step is implemented.

Algorithm 11: *HighwayTimeStep*

Let X be the number of cells per lane, Y the number of lanes, and t the current time step
/* Observation step */
forall $x = X-1, X-2, ..., 1, 0$ **do**
  forall $y = 0, ..., Y-1$ **do**
    if *car c in position* x, y **then**
      Determine driver's next action $a_{c,t}$ according to their behavior
    end
  end
end
/* Update step */
forall $y = 0, 1, ..., Y-1$ **do**
  forall $x = 0, 1, ..., X-1$ **do**
    Update cell at position x, y
  end
end

8.3 Simulation Parameters

From what we explained, we can identify the parameters, whether of the highway or of the drivers, that consitute the simulation parameters:

- Number of lanes $X > 0, \quad L \in \mathbb{N}$
- Size of the lanes $Y > 0, \quad C \in \mathbb{N}$
- Number of exits $E \geq 0, \quad E \in \mathbb{N}$
- Size of the exits $S_e > 0, \quad S_e \in \mathbb{N}$
- Size of the space between two exits $S_s > 0, \quad S_s \in \mathbb{N}$
- Crash duration $T_f > 0, \quad T_f \in \mathbb{N}$
- Traffic density $\tau \in [0, 1]$
- Cars' maximum speed $v^{max} > 0, \quad v^{max} \in \mathbb{N}$
- Cars' maximum acceleration $\sigma^{max} > 0, \quad \sigma^{max} \in \mathbb{N}$
- Drivers' irrationality $\iota \in [0, 1]$

Each of these parameters can be defined arbitrarily. The lower bounds on their values guarantee a minimal working environment. There are no upper bounds, though we expect the simulator to be used with reasonable values (e.g. a highway of 100 lanes would not really make sense).

8.4 Highway Performance

At this point, we have a fully functional traffic simulator and, since we wanted to make a realistic system, we would like to see how well it performs and mirrors real-world traffic. First, we have to define what a good performance is; we want to maximize the ratio of drivers that reach their goal and minimize the number of crashes. Second, we have to run some simulations and see what happens. It is clear that the result of a simulation depends on the chosen parameters (section 8.3), but testing how each one of them affects the behavior of the simulator (when the others are fixed) would take too much time. Hence, we focus on what we consider to be the most important ones: the irrationality of the drivers, as it impacts their behavior, and the traffic density.

We run a series of tests to analyze the percentage of each outcome and the throughput of our traffic simulator depending on the irrationality probability of the drivers. To do that, we fixed the other parameters:

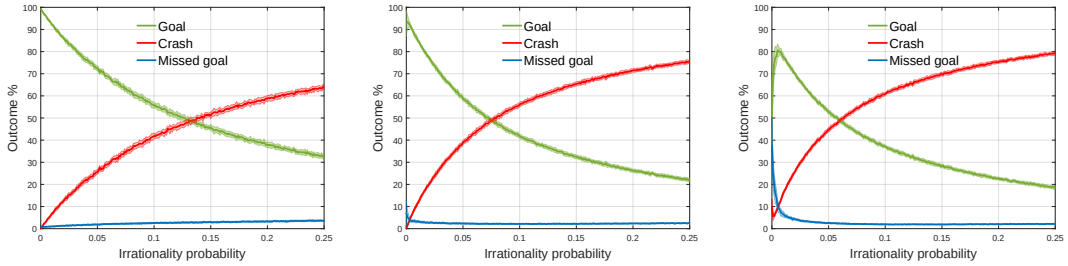
- Number of lanes = 5
- Size of the lanes = 80
- Number of exits = 2
- Size of the exits = 5
- Size of the space between two exits = 7
- Crash duration = 10
- Cars' maximum speed = 3
- Cars' maximum acceleration = 2

The possible outcomes are called *goal* if the driver reaches their goal, *crash* if they crash and *missed goal* if the driver does not reach their goal (e.g. misses their exit).

For each irrationality probability between 0 and 0.25 (with a step of 0.001, thus 251 different values), we run the simulation for 3000 time steps and repeat it 20 times to compute the average. This is done for three different traffic densities: 0.25, 0.50, and 0.75.

Figure 8.1 shows the percentage of each outcome – goal, crash or missed goal – depending on the irrationality probability for three traffic densities. We can see that when the traffic density increases, it takes a lower irrationality probability value to make the simulator behave poorly. With a lower traffic density, there should be more space between the drivers and they consequently have more room to make mistakes or, in this case, random actions.

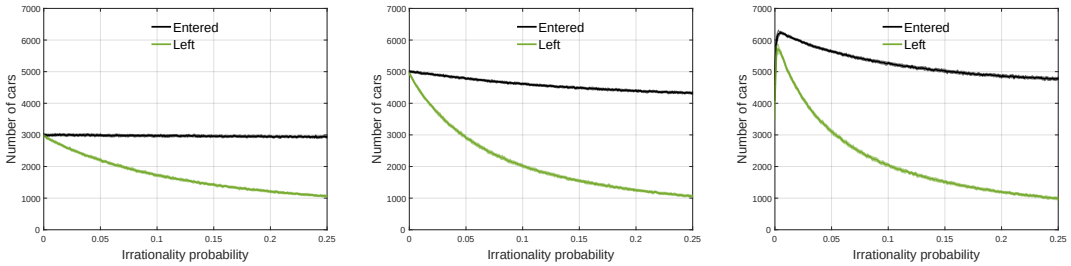
Figure 8.2 shows the number of cars that entered and left the highway depending on the irrationality probability for different values of the traffic density. The number of cars that enter the highway is obviously higher when the traffic density is high. However, we can see that when the irrationality probability increases, this number tends to decrease



(a) Traffic density $\tau = 0.25$ (b) Traffic density $\tau = 0.50$ (c) Traffic density $\tau = 0.75$

Figure 8.1: Outcome percentage depending on the drivers' irrationality

for high traffic densities while it remains somewhat constant for low ones. The reason for this is that, as there are more drivers on the highway, and more crashes, it is more likely to have congestion that prevents new drivers from entering the system.



(a) Traffic density $\tau = 0.25$ (b) Traffic density $\tau = 0.50$ (c) Traffic density $\tau = 0.75$

Figure 8.2: Throughput depending on the drivers' irrationality

8.5 Graphical Simulator

We implemented a graphical version of the simulator. The three main components of the GUI are the highway itself, a tab with statistics about the current simulation (Figure 8.3), and a tab with information about the selected car, highlighted in orange (Figure 8.4). The simulation can be run non-stop or step by step and we can switch between these two modes at any time.

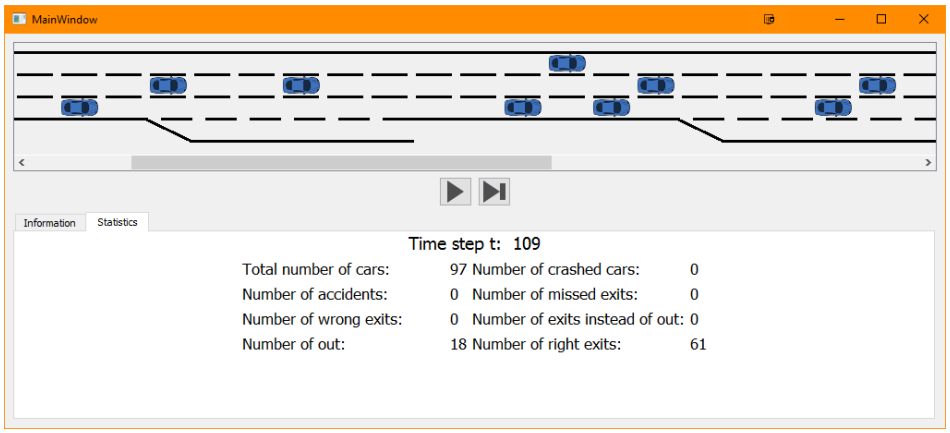


Figure 8.3: Graphical simulator with statistics tab open

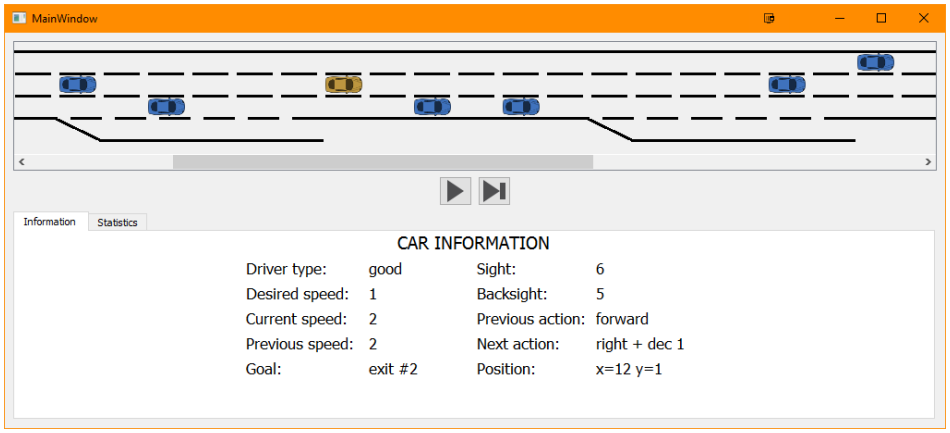


Figure 8.4: Graphical simulator with the information tab showing the selected car's information. Note that these pieces of information include the "driver type" that is actually not relevant but refers to a feature of the simulator that is not used.

8.6 Implementation Notes

The simulator has been implemented in Python 2.7, and the GUI is built with the PyQt5 module. Some parts of the code have been optimized with Cython, a Python module that optimizes the code by allowing us to annotate C types in the code.

For example, the simulations presented in section 8.4 were executed on the ULB cluster Hydra and took from 70 to 100 hours to complete before the Cython optimization; with the optimization, the executions took from 25 to 44 hours.

The graphic simulator can be run with a command line; all the different parameters of a simulation can be set through this command line directly (e.g. "--nb-lanes 4" for a highway with four lanes).

The source code of the project will be made available in due time.

Part III

Autonomous Cars

Chapter 9

Agents

In this chapter, we will explain how our learning agents, the autonomous cars, were constructed with regard to the human drivers. Thus, we will briefly explain how human drivers' attributes are adapted for the autonomous cars, and how they should behave.

9.1 Attributes

The human entities in our system are composed of both the driver and their car, and we define the self-driving cars in a similar way; we make a distinction between the car, that is defined as presented in section 6.2 for humans, and the *learning agent* that is controlling the car. We opt for a human-like definition of these agents; they have almost the same attributes (see section 7.1): the sight and the goal. While the goal is chosen randomly when an agent arrives on the highway, the sight is always fixed to the same value. The other noticeable difference is the fact that our learning agents do not have a desired speed. We define the autonomous cars as entities whose primary concern is to avoid crashing; they should consequently not exhibit any preference for a certain speed as long as they are driving safely.

Furthermore, we add an attribute ϵ to these learning agents; this is their probability of choosing a random action at each time step, as defined in Equation 2.8. We thus use the ϵ-greedy strategy to have a balance between exploration and exploitation.

Note that since the autonomous cars have a limited sight, their decisions will depend only on what they see and not on the full state of the highway. From now on, we will use s to denote the states relative to the agents and s^H for the full states of the highway.

9.2 Rewards

The actions that an autonomous car can do are the same as for the human drivers (section 7.2). Their behavior, however, depends on the learning model used to train them. We consider that impossible moves – that lead outside the highway – are never

considered by the agents; this is some sort of prior knowledge they already possess. This constitutes the only knowledge they have at the beginning, they will have to learn the rest.

We need to define the reward function R; there are three different final states the agents can be in. First, they can reach their goal, whether it is taking an exit or not. Second, they can miss their goal; they either took a wrong exit or missed their exit. Finally, they can crash. Hence, we define the following rewards:

- ρ_ω, the reward received by the agent when it reached its goal
- $\rho_{\bar{\omega}}$, the reward received by the agent when it failed to reach its goal but did not crash
- ρ_f, the reward received by the agent when it crashed

Since reaching the goal and crashing are completely opposite outcomes, we define ρ_ω = - ρ_f. Moreover, since missing the goal but not crashing is preferable to causing an accident but is a less desirable outcome than reaching the goal, the value of this reward should be smaller, such that $\rho_f = -\rho_\omega < \rho_{\bar{\omega}} < \rho_\omega$.

The agent gets these final rewards at the time step that effectively ends its run on the highway. For all the other previous time steps, it receives a default reward that is always equal to 0. However, the agent receives another reward ρ_0, a small penalty, when the agent's speed is 0; our environment is a highway and cars should not be still, unless there is congestion.

9.3 Agents as Drivers

Given that we define learning agents the same way as the human drivers, we can seamlessly add them in the simulator. The only difference is how they will choose an action: by using their learning model, a neural network. We can therefore adapt the highway's time step's algorithm (Algorithm 11) to take the learning agent into account for the observation step, as shown in Algorithm 12.

To decide what action it should take, the reinforcement learning agent uses a neural network to approximate the Q-function (see section 2.7). Thus, at every time step t, the agent c observes its state $s_{c,t}$; this state is then processed in some way – that we will explain shortly thereafter – so that it can be passed to a neural network whose outputs correspond to all the possible actions. The values of these outputs are the estimated Q-values, $Q(s_{c,t}, a)$; as it is using a neural network θ, we denote the Q-function approximated with that network by $Q(s, t; \theta)$. The agent then uses an ϵ-greedy strategy to choose the action $a_{c,t}$.

Algorithm 12: *HighwayObservationStepWithLearningAgent*

Let X be the number of cells per lane, Y the number of lanes, and t the current time step

forall $x = X-1, X-2, ..., 1, 0$ **do**
  forall $y = 0, ..., Y-1$ **do**
    if *car* c *in position* x, y **then**
      if *autonomous car* **then**

$$a_{c,t} = \begin{cases} \max_{a'} Q(s_{c,t}, a'; \theta) & \text{with probability } 1-\epsilon \\ \text{random action } a & \text{with probability } \epsilon \end{cases}$$

      else
        Determine driver's next action $a_{c,t}$ according to their behavior
      end
    end
  end
end

The neural network used by the learning agent will be trained with reinforcement learning by using the methods presented in section 3.5.

Chapter 10

Neural Network Models

In this chapter, we present the different neural network models that we will use to train our autonomous cars. These models define what information the learning agents use and how they are encoded as inputs to the neural networks.

10.1 Common Structure

Before we start to explain our different models, we need to define the building structure; how these neural networks are used by the learning agents. We use a feedforward neural network (see section 3.3) whose outputs correspond to the possible actions defined in section 7.2. Our models define different ways of using information about the agent's current state. Thus, they either encode different information or encode the same information differently to produce the inputs. This idea is summarized in Figure 10.1.

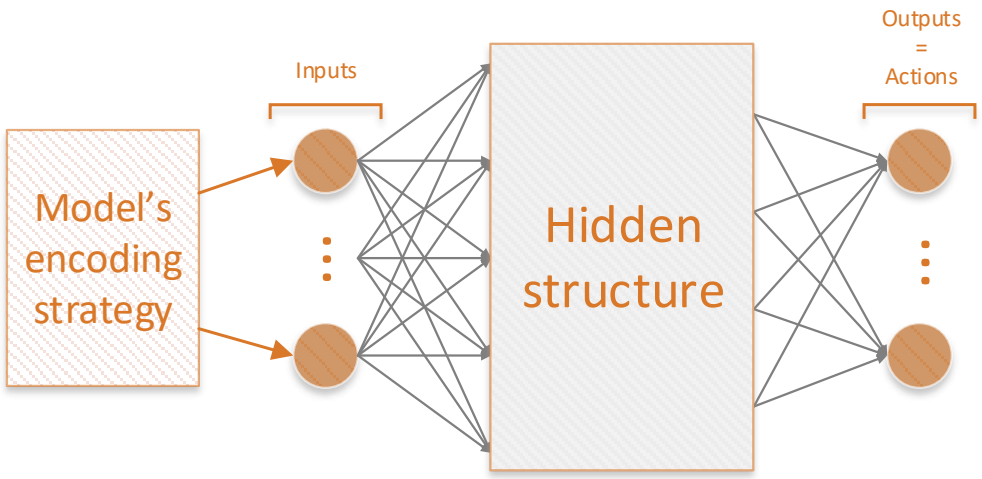


Figure 10.1: Common building structure of the different learning models

10.2 Required Information

We first start by defining the minimum amount of information that we think an autonomous car should have. Consequently, all the models that we design will possess these pieces of information. They are:

- The current lane that the agent is in
- The current speed of the agent
- The goal of the agent
- What the agent sees; cars, crashes and exit indications

What the agent sees depends on its sight ϕ_c^+ and backsight ϕ_c^-. We defined the field of view at a time step t of a driver Φ_c, t in Equation 7.1; the cars, crashes and exit indications seen are the ones present in this field of view.

10.3 *Simple Binary* Model

Our first model consists of only the basic information presented in section 10.2. Moreover, we define this model as binary; each input of the neural network can only be 0 or 1; as such, information regarding the field of view of the agent can only be 1 if there is something, or 0 if there is nothing. The agent only knows whether there is a car in some position, but has no additional information about this car.

As we only consider binary inputs, we must find a way to encode the information properly. For the current lane and speed, and the agent's goal, we must have one input neuron for each possible value that these attributes could have; the input corresponding to the current value will be set to 1. We formalize this as a vector of possible values for an attribute, with 1 for the current value and 0 elsewhere. For example, if there are 5 lanes on the highway and the agent's current lane is the left-end lane (index $y = 0$), the input vector for the lanes will be $[1, 0, 0, 0, 0]$.

To encode the observations of the agent, i.e. what it sees, we need to translate its field of view into a matrix of 1s when it sees something, and 0s otherwise. This is shown in Algorithm 13. This matrix of binary observations O is then flattened out (lane by lane) to get a 1-dimensional vector that will serve as input to the network.

Finally, as exits are not considered in what we explained previously, we need to encode information about them as well. For this, we use a vector of indications for each exit; each exit has a vector of the size of number of cells that the agent can see, if there is the indication of this exit in its field of view, we put a 1 in the vector in the position that corresponds to the position in the field of view. Algorithm 14 shows how to build these exit vectors from the agent's observations of the exit lane.

Algorithm 13: Encoding observations for the *Simple binary* model

Initialize matrix O of 0s of size $Y \times$ full sight $(\phi_c^+ + \phi_c^- + 1)$
for *lane y in field of view* $\Phi_{c,t}$ **do**
  for *cell x in lane y's observations* **do**
    if *car or crash in cell x* **then**
      $O_{y,x} \leftarrow 1$
    end
  end
end

Algorithm 14: Encoding exits for the *Simple binary* model

Initialize array E^i of 0s of size full sight $(\phi_c^+ + \phi_c^- + 1)$ for each exit
for *cell x in exit lane's observations* **do**
  if *exit indication in cell x* **then**
    $k \leftarrow$ indicated exit
    $E_x^k \leftarrow 1$
  end
end

Figure 10.2 shows an example of this model's encodings for an agent whose sight is $\phi_c^+ = 4$ (backsight $\phi_c^- = 3$), driving with a speed of 1, and whose goal is to continue on the highway. The learning agent is shown in orange while cars in its field of view are in grey.

All the input vectors are concatenated and passed to the network.

Finally, as this model is binary and consists of the minimum information required, we call it *simple binary*.

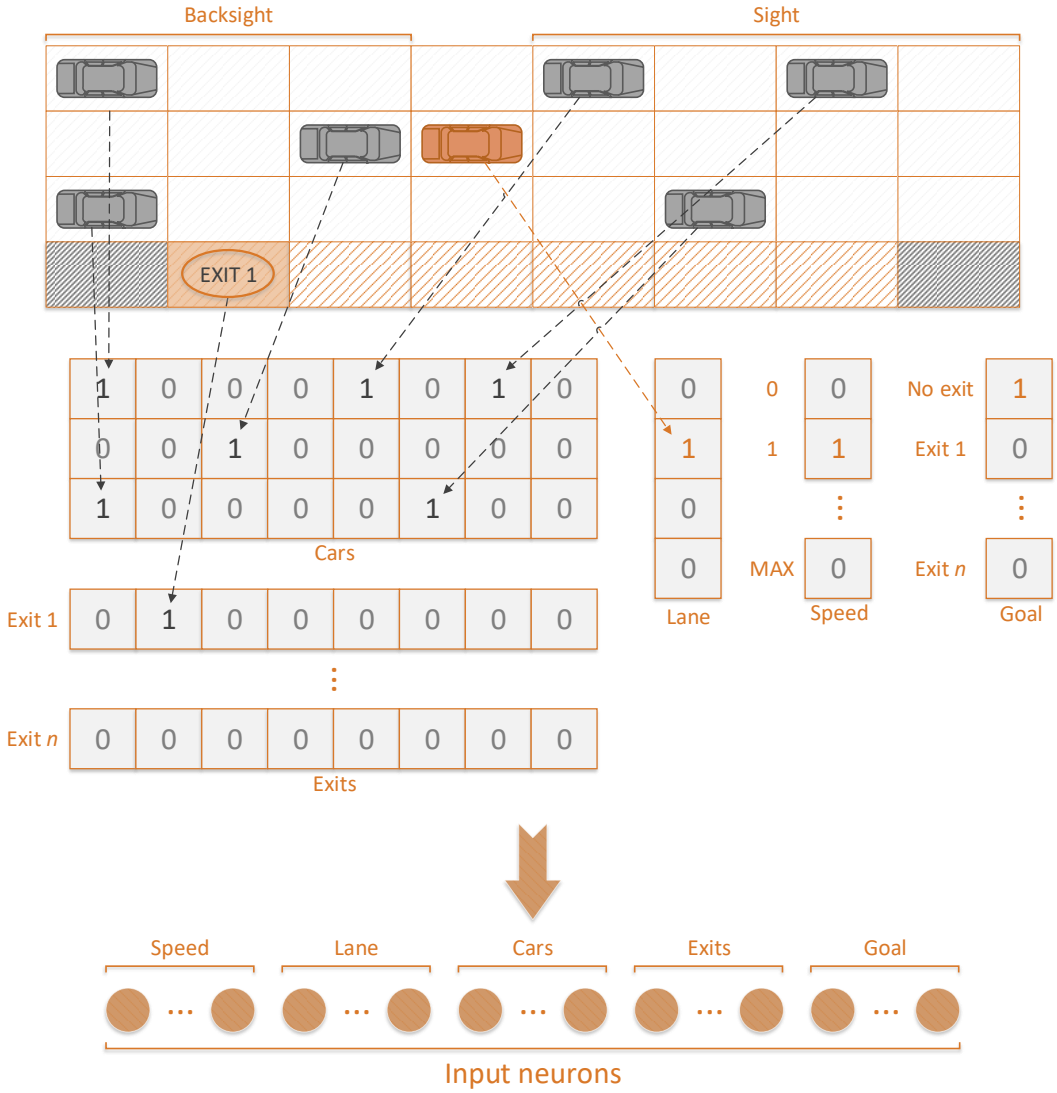


Figure 10.2: *Simple binary* model encoding

10.4 *Improved Binary* Model

There is one important information that is missing from the *simple binary* model that we just defined: the cars' speeds. Indeed, the agent only encodes the presence of cars in its field of view. We would like to have a model, still binary, that somehow encodes the speeds of the observed cars. One way of doing this could be to add, similarly to how we encode the agent's speed, a vector of speeds for each position in the observation matrix O. However, this approach would greatly increase the number of inputs of the network.

Another solution is to use the inputs of the previous time step as additional inputs. We consequently use the *simple binary* model to encode the inputs according to the

current state s_t and the previous state s_{t-1}. This approach is shown in Figure 10.3.

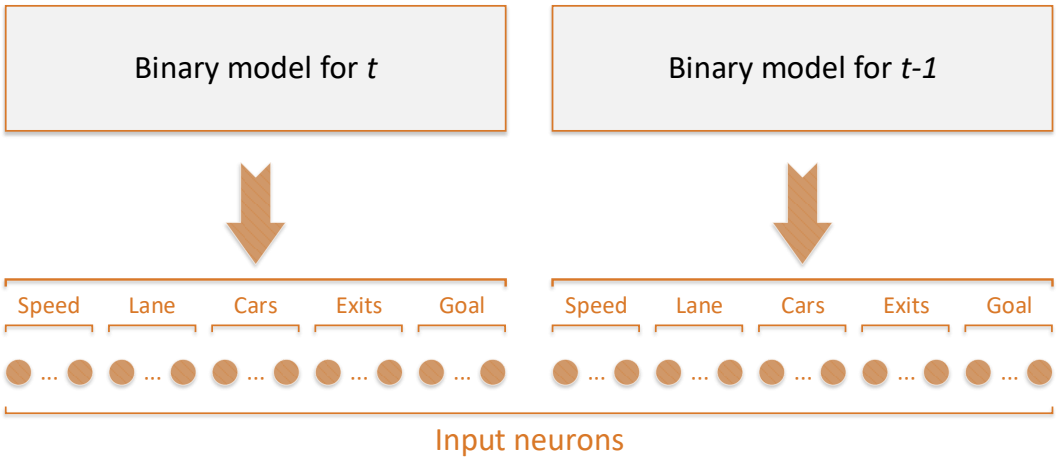


Figure 10.3: *Improved binary* model encoding

As this model is still binary but provides more information than the *simple binary* model, we call it *improved binary*.

10.5 *Simple Speed* Model

The next model is an attempt to simplify the *improved binary* one; we still want to encode the speeds of the observed cars but without using the previous state. To do this, we need to encode the speeds as real values and use another matrix, of the same size than the observation matrix O but with the speed of the observed car at the corresponding position, if any. We want real values for the speeds that are bounded by 0 and 1. To do this, we use the relative speed of the cars with regard to the maximum speed v^{max}. For example, if $v^{max} = 4$ and the car's speed is 2, the relative speed will be $\frac{2}{4} = 0.5$.

We modify the algorithm for the observations' encoding (Algorithm 13) to add the encoding of the speeds; the new algorithm is shown in Algorithm 15.

Figure 10.4 presents an example of this model's encoding.

We name this model *simple speed* as it remains somewhat simple but encodes the speeds explicitly.

Algorithm 15: Encoding observations and speeds for the *Simple speed* model

Initialize matrices O and V of 0s of size $Y \times$ full sight $(\phi_c^+ + \phi_c^- + 1)$
for *lane* y *in field of view* $\Phi_{c,t}$ **do**
  for *cell* x *in lane* y*'s observations* **do**
    if *car or crash in cell* x **then**
      $O_{y,x} \leftarrow 1$
      if *car in cell* x **then**
        $V_{y,x} \leftarrow \frac{car's\ speed}{v^{max}}$
      end
    end
  end
end

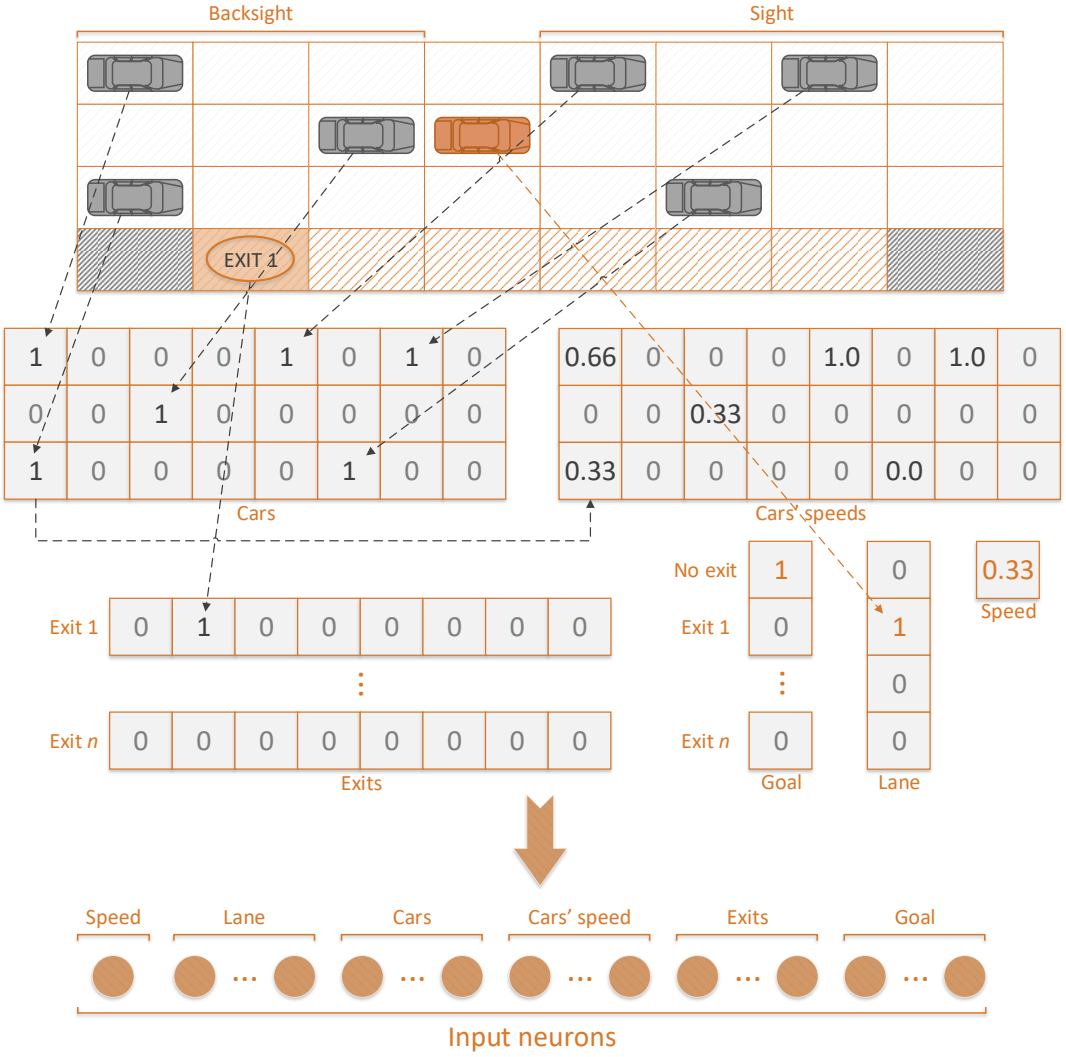


Figure 10.4: *Simple speed* model encoding

10.6 *Cars Speed* Model

We now want a model which makes use of the 2-dimensional nature of the highway and the observations of the agents. Namely, we want to use convolutional layers on the inputs representing these observations. They are indeed well suited for convolution since they are 2-dimensional, as demonstrated by the fact that we initially use a matrix to store them.

Our model is inspired by the work of Genders and Razavi (2016); starting from a continuous environment, they discretize a portion of the lanes into cells, and use them to create two vectors: a binary one for the presence of a car in the cell, and a real-valued one for the speed of these cars. These first two vectors are two-dimensional data (lane and cell) flattened out, and both go through a convolutional neural network (the same for the two). The outputs of these CNNs are then used as inputs, next to a third vector of additional system information, for a classic feedforward neural network.

This model basically uses the same information as the *simple speed* model, but the observation and speed matrices O and V go through a CNN first. Figure 10.5 shows this new model.

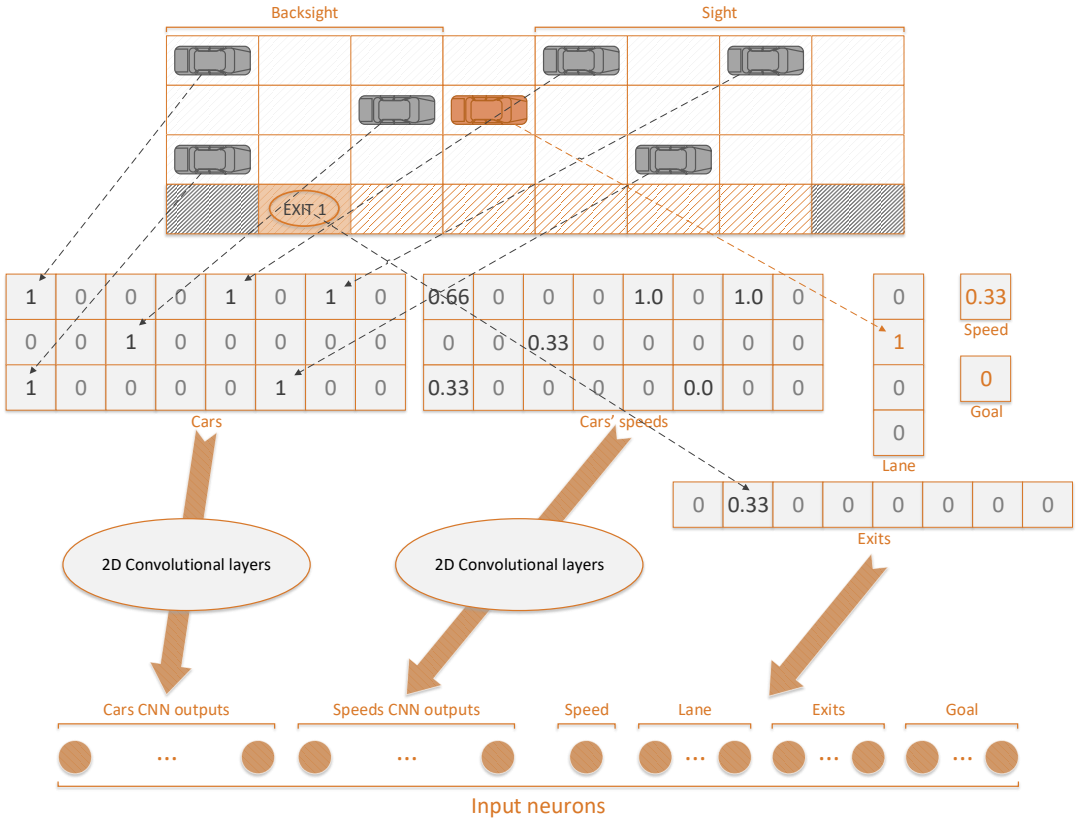


Figure 10.5: *Cars and speed* model encoding

In an attempt to decrease the number of inputs, this model also introduces a simplification of the exits' encoding. Similarly to the speed, we now encode the exits in a relative way; instead of having a vector for each exit, we have only one vector, and the value is not binary anymore. If the exit indication in the field of view is, for example, the first exit out of 3, the value will be $\frac{1}{3} = 0.33$. Likewise, the encoding of the goal of the agent is now a single real-value that is 0 if the goal is to continue on the highway, or the relative number of the exit it wants to take. These modifications are shown in Figure 10.5.

We call this model *cars speed* model.

10.7 *Cars* t Model

Finally, our last model is based on the same idea as the *improved binary* model; we use information about the previous time step (the cars' presence represented by the observation matrix O) instead of the current speed of the cars. This time, the observation matrix of the previous time step $t-1$, denoted O_{t-1}, is not additional inputs, but it forms, along with the current observation matrix, a 3-dimensional matrix with time as the third dimension. We then pass this matrix through a 3-dimensional convolutional neural network in a way that is similar to the previous model.

We also keep decreasing the number of inputs by including the learning agent itself in its observation matrix. To differentiate itself from the other cars, the value is not 1 but 0.5. This way, the only additional information required are the exits and the agent's goal.

Figure 10.6 illustrates this model that we call *cars* t, as it encodes the cars' presence at different time steps.

What we explained only considers a length of 2 (current and previous time steps) for the time dimension. The model can however be extended to longer time dimensions.

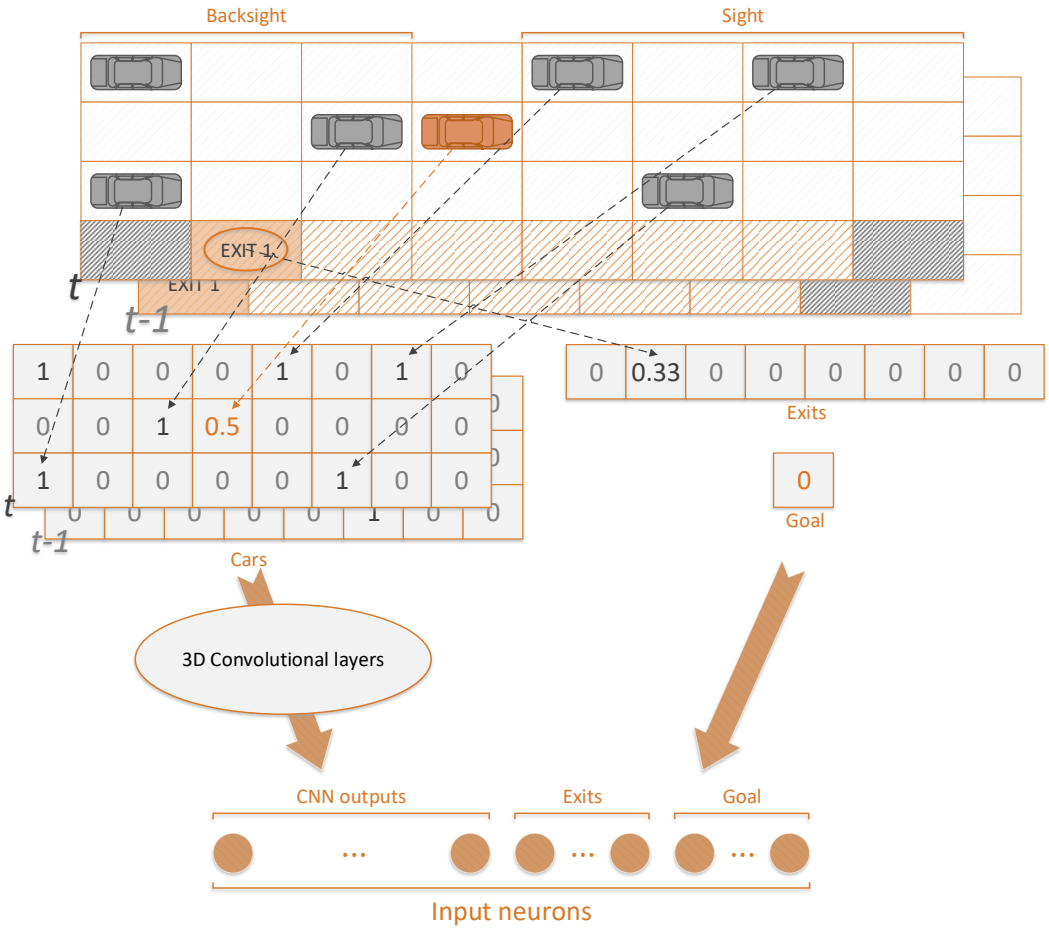


Figure 10.6: *Cars t* model encoding

Part IV

Deep Reinforcement Learning

Chapter 11

Single-Agent Setting

We first train autonomous cars in a single-agent environment; there is only one learning agent on the highway, other cars are our simulated human drivers, whose behavior is fixed (see section 7.3).

All the models (chapter 10) were implemented in Python 2.7 with the module Keras[1]. This modules uses either TensorFlow[2] or Theano[3] as back-end.

In this chapter, we will first present the basic training strategy we used, then present the different results we obtained.

11.1 Training

Now that we have defined all our neural network models and the simulator, it is time to tackle our main problem: making an autonomous car learn how to drive.

11.1.1 Training Scheme

The training environment we set up for our learning agent is divided into episodes. One episode consists of a full run, or simulation, of the agent on the highway.

First, as the agent could potentially stay still indefinitely, if it always chooses to stay at the speed zero, we need to guarantee that its run on the highway will come to an end. We thus define a maximum number of steps T^{max}; if the agent's run on the highway reaches this upper bound, we consider the run over and introduce another reward, denoted ρ_T, for this new outcome: the *overtime* outcome. Furthermore, the value of T^{max} must be chosen wisely as it should not, for example, prevent the agent from finishing its run in a reasonable number of steps. In short, the bound must not be too low nor too great. As the time required for an agent to leave the highway depends on the size of the lane X, we need to define the overtime bound based on this value.

[1] `https://keras.io/`
[2] `https://www.tensorflow.org/`
[3] `http://deeplearning.net/software/theano/`

If a car always drives at the minimum speed (1), it will need exactly X time steps to leave the highway; it progresses cell by cell. We could therefore define the bound to this value. However, such a bound would not permit the agent to stay still in some time steps – and staying still may be helpful to avoid a crash. We could then use twice that value, $2 \times X$, but it means that the agent could potential stay still half of the time and still finish its run. Finally, we define arbitrarily T^{max} as $2 \times X - 10$; we want the agent to actually drive more often than it is not moving.

Second, we want our learning agent to arrive in a highway after *some time*, to ensure that it arrives in a situation where it is not the only car on the highway. To do that, we perform an arbitrary number of highway time step updates, usually 20, before adding the agent in the system. Then, we randomly initialize the attributes – speed v and lane y – of the agent and add it on the highway. Note that if it cannot, for some reason, be placed in its lane, we perform other highway time step updates until it is possible.

Finally, we use experience replay (see subsection 3.5.2) to train the network in an online fashion; at each time step, we sample a batch of experiences from the memory and train the network with it.

The training of the neural network – that is, the weights updates – is done by the module Keras itself. All our models were compiled with the *stochastic gradient descent* optimizer and the *mean squared error* loss function (defined in Equation 3.5).

To train the network, Keras needs inputs for this network as well as the wanted, target outputs of these inputs. Thus, we need to define these target outputs according to the chosen action. To do that, we define the target outputs as the outputs that we get from the network except for the chosen action. For this particular output, we set its target value to the newly estimated Q-value. That is, if the new state is final, the immediate reward r_t received; otherwise, the immediate reward and the discounted best possible future reward as estimated by the network. In short, when an agent in state s_k performs action a_k that yields the immediate reward r_k and arrives in the next state s_{k+1}, we define the target outputs $\mathbf{y}$ as:

$$\mathbf{y_a} = \begin{cases} Q(s_k, a; \theta) & \text{if } a \neq a_k \\ r_k + \gamma \max_{a'} Q(s_{k+1}, a'; \theta) & \text{if } a = a_k \end{cases} \tag{11.1}$$

Where $Q(s_{k+1}, a; \theta)$ is equal to 0 for every action a when s_{k+1} is final.

This whole process is formalized in Algorithm 16.

Algorithm 16: Single-agent training algorithm

Initialize replay memory $\mathcal{D}$ of length M
for *episode* $= 1, ..., N$ **do**
  Initialize highway H according to fixed highway parameters
  for $t = 1, ..., 20$ **do**
    Perform one time step update of the highway H
  end
  Initialize learning agent c with a fixed sight ϕ_c^+ and ϵ
  Choose randomly lane y in which to add the agent, and speed v of the agent
  while *agent cannot enter lane* y **do**
    Perform one time step update of the highway H
  end
  Add the learning agent c on the highway
  for $t = 1, ..., T^{max}$ **do**
    Perform one time step update of the highway H (Algorithm 11 and Algorithm 12)
    /* Experience replay */
    Observe the state s_t, action a_t and reward r_t of the agent for that time step, and the next state s_{t+1}
    Store experience (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    if *replay memory* $\mathcal{D}$ *is full* **then**
      Sample minibatch of size B from experience memory $\mathcal{D}$
      Initialize batch training buffer of size B
      for *experience* (s_k, a_k, r_k, s_{k+1}) *in minibatch* **do**
        Get network outputs $\mathbf{y}$ according to $Q(s_k, a; \theta)$
        Set $y_{a_k} = \begin{cases} r_k & \text{if state } s_{k+1} \text{ is final} \\ r_k + \gamma \max_{a'} Q(s_{k+1}, a'; \theta) & \text{if state } s_{k+1} \text{ is not final} \end{cases}$
        Store $(s_k, \mathbf{y})$ in batch training buffer
      end
      Train network θ against batch training buffer
    end
  end
end

11.1.2 Training Parameters

A training experiment is defined by numerous parameters; first, the traffic simulator and all its own parameters (section 8.3), and all the parameters regarding the learning agent and the training algorithm. These parameters are:

- ϕ_c^+, the sight of the learning agent (its backsight $\phi_c^- = \phi_c^+ - 1$)

- ϵ, the agent's probability of choosing a random action
- R, the reward function; which corresponds to defining the different rewards ρ_ω, $\rho_{\bar{\omega}}$, ρ_f, ρ_0 and ρ_T
- μ_c, the neural network model used by the agent
- γ, the discount factor as defined for the MDP
- α, the learning rate for the neural network training
- N, the number of training episodes
- M, the size of the experience memory buffer
- B, the size of the batches of experiences

Moreover, the hidden structure of the neural network model μ_c can also be chosen; this includes the number of hidden layers, the number of neurons in these layers, their activation function, and the dropout rate δ_c (0 if we do not want to use dropout) to apply to each layer. Finally, if the chosen model is a CNN one, the structure of the convolutional networks can also be set: the number of convolutional layers with their stride and the number and size of the filters.

Learning – sampling batches of experiences to update the network's weights – starts once the experience memory buffer is full: after encountering M experiences.

11.2 Experiments

For our experiments, we train and compare the five different models we presented earlier. Moreover, when possible[4], we also repeat the same experiments with the two different back-ends; we trained our models with TensorFlow as back-end on the AI cluster of the VUB, and we trained them with Theano as back-end on the Hydra cluster of the ULB. We also considered different possible hidden structures for the neural networks.

11.2.1 Settings

For all the experiments, most of the parameters are fixed. We present them in Table 11.1.

The structure of the CNNs for the *cars speed* and *cars t* models are also fixed: there are two convolutional layers, the first with a stride of 1 and 16 filters of size 4×4, and the second with a stride of 1 and 32 filters of size 2×2.

Finally, the reward system of our learning agent is fixed, and the values are shown in Table 11.2.

11.2.2 Results

The training results are shown in the form of graphs presenting the evolution of the percentage of outcomes (accumulated) throughout the training process. We tested four

[4]We encountered issues with the CNN models on Hydra, with the back-end Theano.

(a) Simulation parameters

Simulation parameters	
Number of lanes Y	3
Lane size X	40
Number of exits E	0
Exit size S_e	5
Space size S_s	7
Crash duration D	10
Traffic density τ	0.25
Cars' maximum speed v^{max}	3
Cars' maximum acceleration α^{max}	2
Cars' irrationality ι	0

(b) Learning parameters

Learning parameters	
Number of episodes N	150000
Batch size B	25
Experience memory size M	50
Learning rate α	0.01
Discount factor γ	0.9
Agent's sight ϕ_c^+	6
Agent's ϵ	0.05

Table 11.1: Single-agent training's fixed parameters

	Reward ρ	**Value**
Goal	ρ_ω	+1
Missed goal	$\rho_{\bar{\omega}}$	-0.15
Crash	ρ_f	-1
No speed penalty	ρ_0	-0.01
Overtime	ρ_T	-0.4

Table 11.2: Single-agent training's reward system

different configurations for the hidden structures of the networks:

- 2 hidden layers: the first with 60 neurons and a *tanh* activation function; the second with 30 neurons and a *linear* activation fucntion. No dropout (see subsection 3.3.3).
- 2 hidden layers: the first with 150 neurons and a *tanh* activation function; the second with 75 neurons and a *linear* activation fucntion. No dropout.
- 2 hidden layers: the first with 150 neurons and a *tanh* activation function; the second with 75 neurons and a *linear* activation fucntion. Dropout rate of 0.5.
- 2 hidden layers: the first with 300 neurons and a *tanh* activation function; the second with 150 neurons and a *linear* activation fucntion. No dropout.

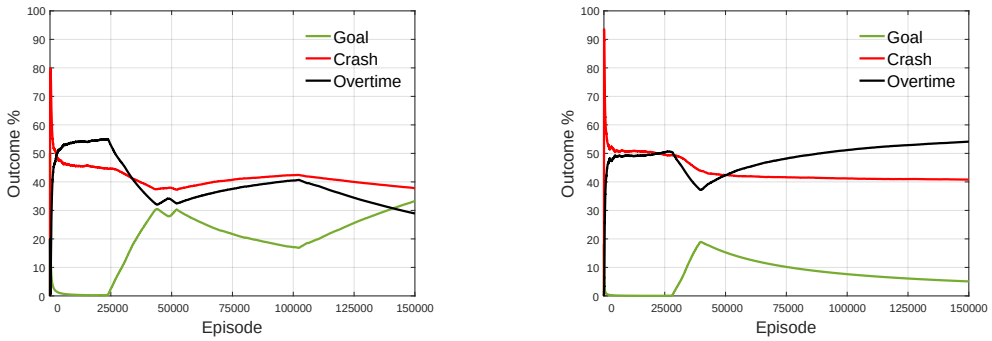
The dropout rate of 0.5 has been chosen because it seems to be optimal for a wide range of networks (Srivastava et al., 2014).

Simple Binary Model

For the *simple binary* model, we can observe multiple things. First, the results seem to change greatly depending on the back-end, Theano or TensorFlow. Second, we are mostly confronted with one of two situations: either the model starts to learn quickly and continues to perform well, as seen in Figure 11.2b and Figure 11.4b, though this situation appears to happen only with TensorFlow; or the model starts to learn to reach the goal but then stagnates and the performance starts to drop, as it is the case in Figure 11.2a, Figure 11.4a and Figure 11.1b. As we can see in Figure 11.1a, the performance increases then decreases then increases again; the learning has not yet converged. This leads us to believe that it might be the case for the others as well.

Finally, we can see how different the training is progressing when using dropout (Figure 11.3a and Figure 11.3b). There, it is roughly similar for the two back-ends, and the performance increases quickly at the start before converging to approximately 55% of goal outcome.

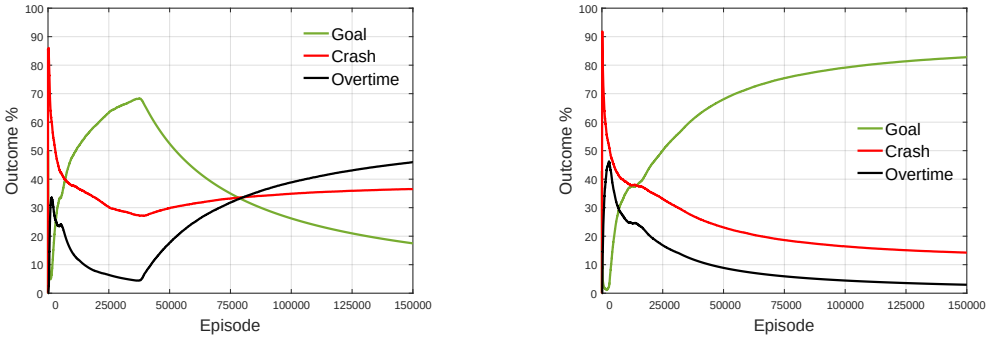
We can note that, in most cases, when the goal percentage decreases, the overtime outcome percentage increases; the drivers do learn not to crash in some cases, but they still do not reach their goal. This can be observed in Figure 11.2a.



(a) Theano

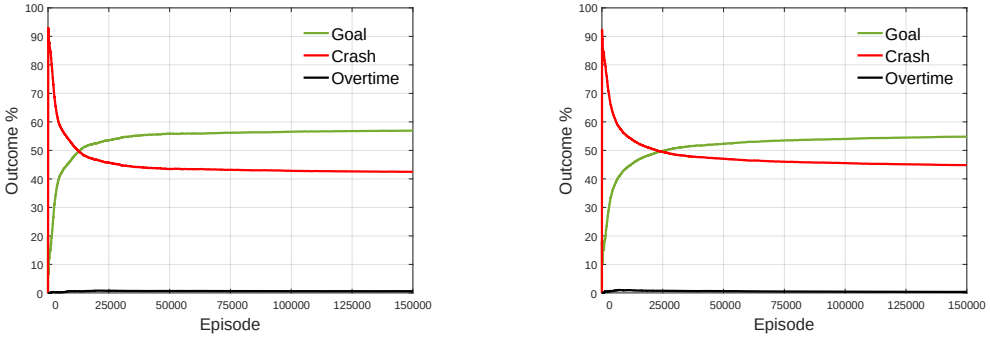
(b) TensorFlow

Figure 11.1: *Simple binary* model training with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.



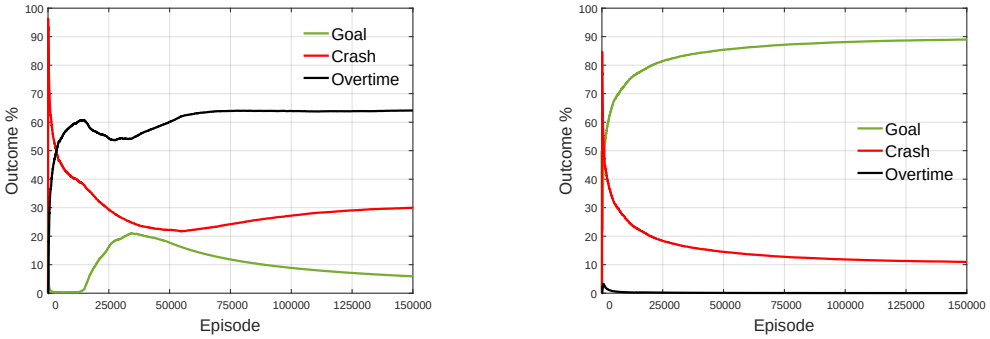
(a) Theano (b) TensorFlow

Figure 11.2: *Simple binary* model training with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.



(a) Theano (b) TensorFlow

Figure 11.3: *Simple binary* model training with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. Other parameters are fixed, see subsection 11.2.1.



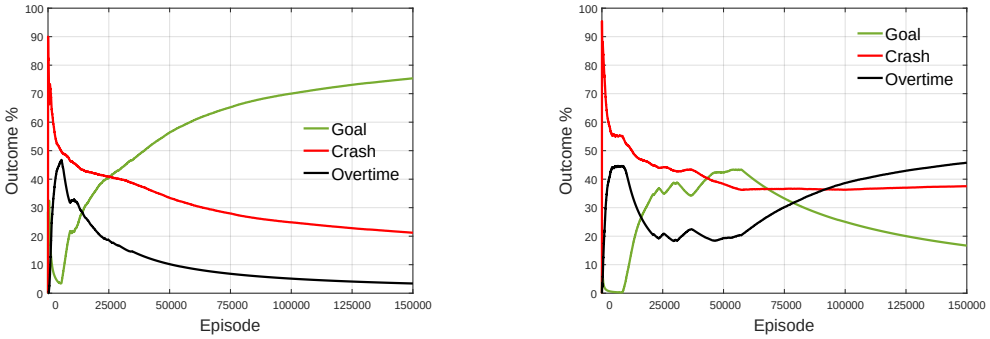
(a) Theano

(b) TensorFlow

Figure 11.4: *Simple binary* model training with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

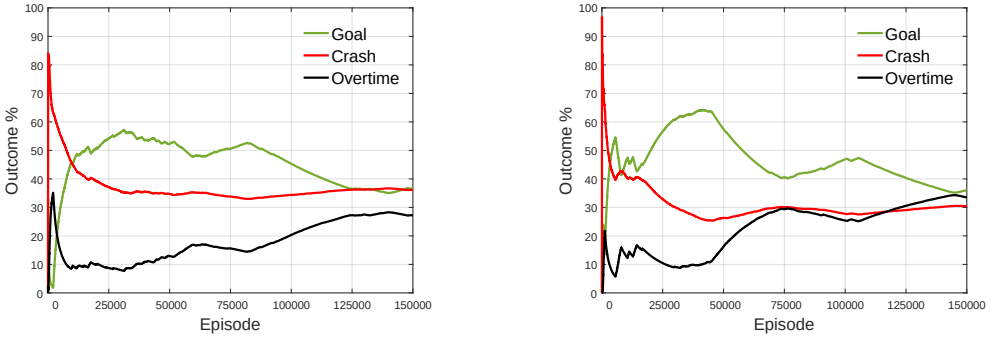
Improved Binary **Model**

The results of the *improved binary* model present less divergence between the two backends; the results of TensorFlow and Theano are similar except for the first setting with the hidden layers of 60 and 30 neurons; in Theano (Figure 11.5a), the model seems to be learning well, while the performance with TensorFlow (Figure 11.5b) decreases after an initial good learning phase. For the hidden layers structure, we can see that the results are unstable (all goal, overtime and crash outcomes are fairly similar) with 150 and 75 hidden neurons (Figure 11.6a and Figure 11.6b) while a high goal percentage is achieved for 300 and 150 hidden neurons; 90% (Figure 11.8a) and 80% (Figure 11.8b) for Theano and TensorFlow respectively. Similarly to the *simple binary* model, we can see that the networks with dropout converge fairly quickly to an average good result of approximately 50% of goal.



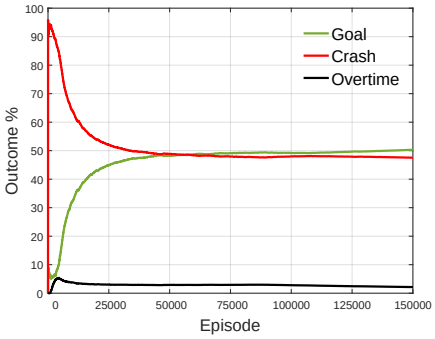
(a) Theano (b) TensorFlow

Figure 11.5: *Improved binary* model training with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

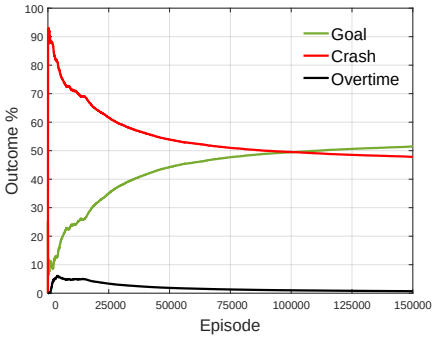


(a) Theano (b) TensorFlow

Figure 11.6: *Improved binary* model training with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

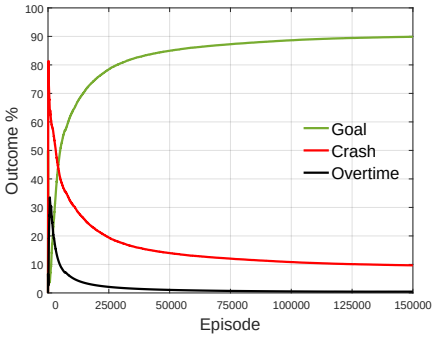


(a) Theano

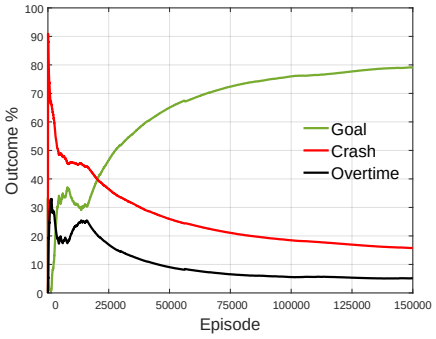


(b) TensorFlow

Figure 11.7: *Improved binary* model training with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. Other parameters are fixed, see subsection 11.2.1.



(a) Theano



(b) TensorFlow

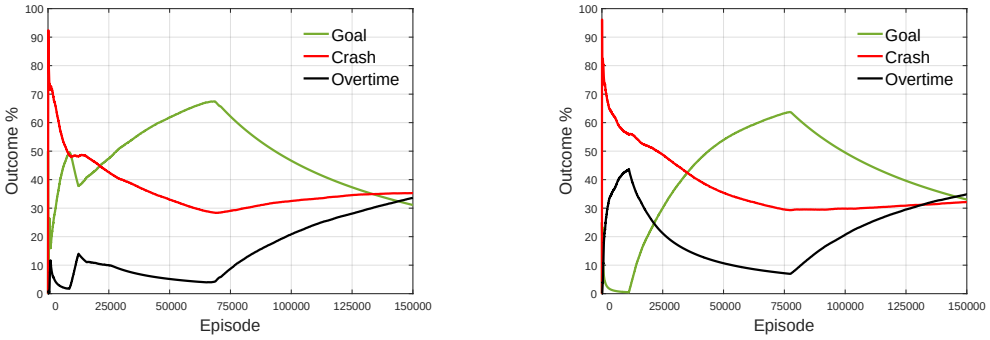
Figure 11.8: *Improved binary* model training with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

Simple Speed **Model**

Once again, the results for the *simple speed* model are almost all the same for the two back-ends. The only exception is for the structure with 150 and 75 hidden neurons without dropout: with Theano (Figure 11.10a), the model achieves a high goal percentage, but the performance with TensorFlow (Figure 11.10b) drops considerably after reaching 70% of goal.

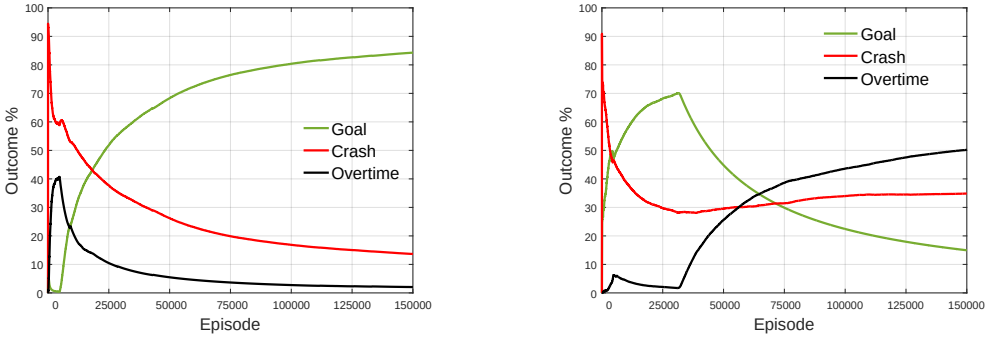
The other observations we can make are the following: first, the simplest networks with 60 and 30 hidden neurons seem to both experience some sort of catastrophic forgetting (subsection 3.5.2) as the goal percentage suddenly decreases after approximately 70000-75000 episodes; instead of reaching their goal, the drivers seem to learn to decelerate and ultimately to stay still as the overtime outcome starts to increase. Furthermore,

for the networks with dropout, the results are similar to the previous models in that the goal percentage increases rapidly then stabilizes to some average value, here around 45%. Finally, the most complex networks with 300 and 150 neurons seem to learn quite quickly to reach the goal (approximately 90% of the time) and stay stable.



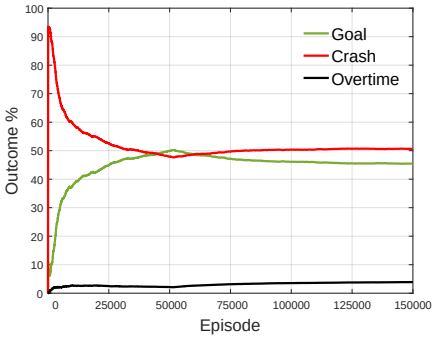
(a) Theano (b) TensorFlow

Figure 11.9: *Simple speed* model training with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

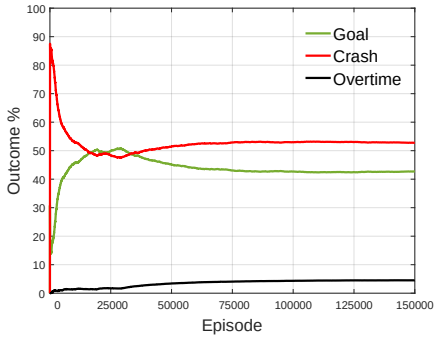


(a) Theano (b) TensorFlow

Figure 11.10: *Simple speed* model training with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

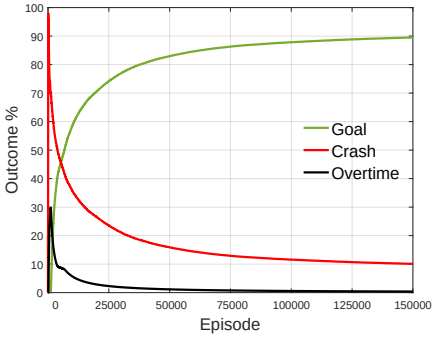


(a) Theano

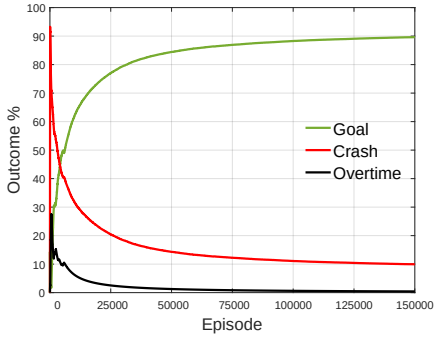


(b) TensorFlow

Figure 11.11: *Simple speed* model training with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. Other parameters are fixed, see subsection 11.2.1.



(a) Theano



(b) TensorFlow

Figure 11.12: *Simple speed* model training with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

Cars Speed Model

The results[5] for our first CNN model – the *cars speed* model – are of two types: either they learn fairly well, as shown in Figure 11.13, Figure 11.14, and Figure 11.16; or they learn before stabilizing at an average of 40% of goal reached, as shown in Figure 11.15. The network that exhibits the worst performance is the one that uses dropout, while the percentage of goal reached for the other networks (without dropout) is high and increases with the number of hidden neurons: approximately 70% for the 60/30 hidden neurons structure, 85% for the 150/75 hidden neurons structure and 90% for the 300/150 hidden neurons structure.

[5]Due to issues with the clusters, these results are obtained only with the TensorFlow back-end, and are not all 150,000 episodes long.

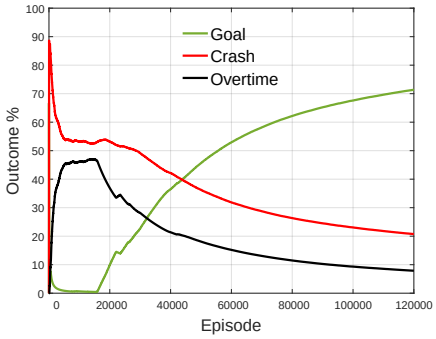


Figure 11.13: *Cars speed* model training (TensorFlow) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

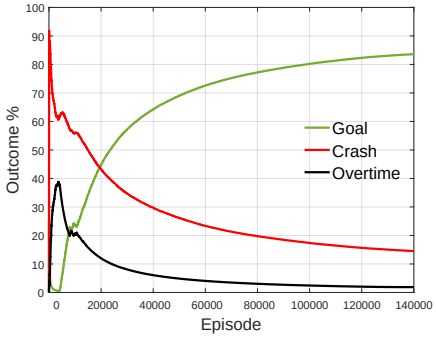


Figure 11.14: *Cars speed* model training (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

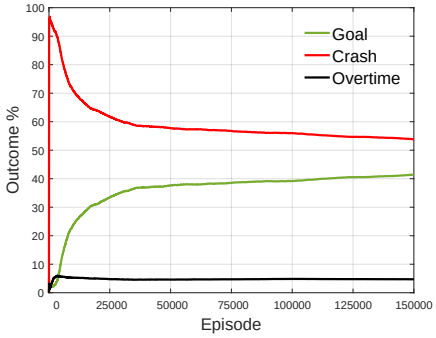


Figure 11.15: *Cars speed* model training (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. Other parameters are fixed, see subsection 11.2.1.

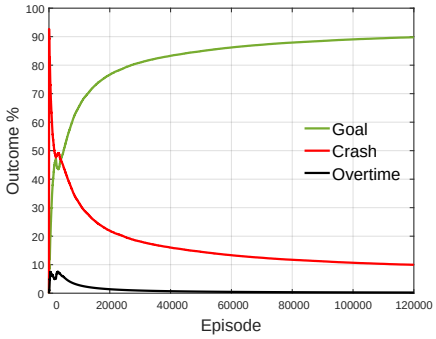


Figure 11.16: *Cars speed* model training (TensorFlow) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

Cars t Model

Although we only have partial results[6], we can make the following observations: the networks that do not use dropout (Figure 11.16, Figure 11.17, and Figure 11.18) seem to learn well, while the network using dropout (Figure 11.15) does not; it either learns very slowly or just converges to approximately 10% of goal reached.

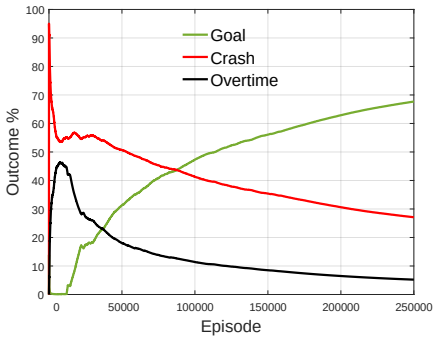


Figure 11.17: *Cars t* model training (TensorFlow) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

[6]Due to problems with the clusters for the CNN models.

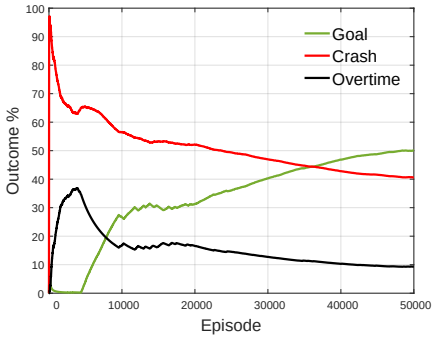


Figure 11.18: *Cars t* model training (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

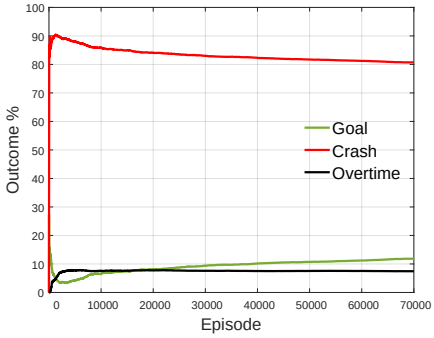


Figure 11.19: *Cars t* model training (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. Other parameters are fixed, see subsection 11.2.1.

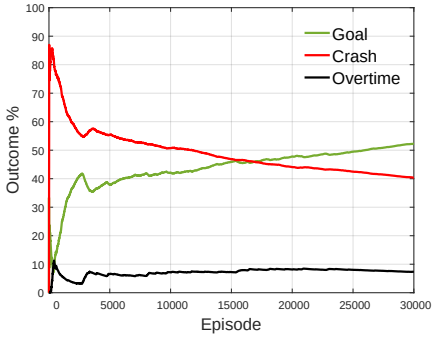


Figure 11.20: *Cars t* model training (TensorFlow) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. Other parameters are fixed, see subsection 11.2.1.

11.3 Testing the Trained Models

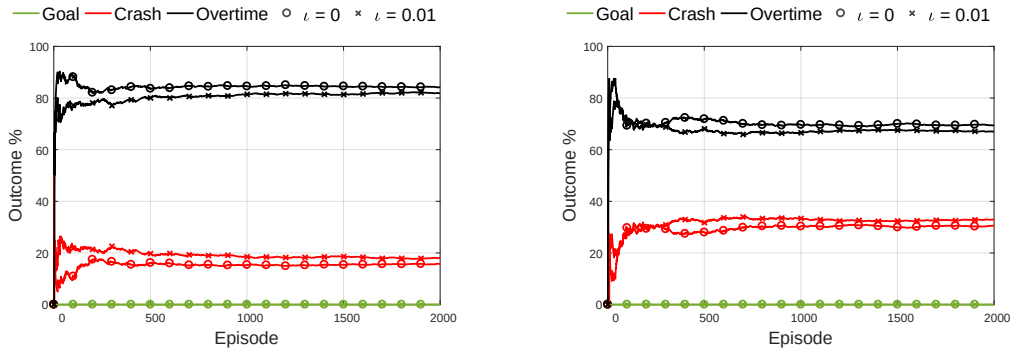
We now have trained models that we want to test: see how they perform when the agent using the model does not learn any further but just exploits what it has learned. Hence, we perform tests which consist of basically the same setting, except for the ϵ value of the agent that is now set to 0 so that the agent only exploits.

We test our newly trained agent in four different situations:

- With a traffic density $\tau = 0.25$ (same as training) and humans' irrationality $\iota = 0$ (same as training)
- With a traffic density $\tau = 0.25$ (same as training) and humans' irrationality $\iota = 0.01$
- With a traffic density $\tau = 0.75$ and humans' irrationality $\iota = 0$ (same as training)
- With a traffic density $\tau = 0.75$ and humans' irrationality $\iota = 0.01$

As there are many results, we will only show the most relevant ones; the entirety of these tests is presented in Appendix A.

First, we test a trained network whose goal percentage is low (Figure 11.5b). The results of this test are shown in Figure 11.21. We can see that the drivers never manage to reach their goal but they are not crashing in most cases; instead, they go overtime. It means that they learned to stay still. It is possible to have the driver still in the first position of a lane; in this case, other cars will not be able to enter this lane.



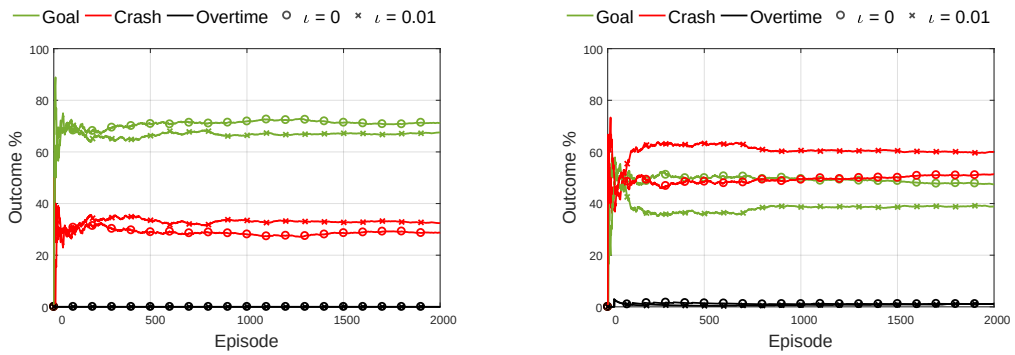
(a) Traffic density $\tau = 0.25$

(b) Traffic density $\tau = 0.75$

Figure 11.21: Testing the *improved binary* model trained on the AI cluster (TensorFlow) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.5b.

Second, we test an "average" trained model, one whose goal percentage is around 50%; it is the case for the networks using dropout. We arbitrarily test the network

using dropout for the *simple binary* model (Figure 11.3a). In the results, shown in Figure 11.22, we can observe that, in the simulator with the same parameters as the training environment, drivers achieve a goal percentage of approximately 70%, while this percentage was slightly below 60% during the training phase. Moreover, the goal percentage decreases slightly when human drivers are irrational (with probability 0.01) but stays somewhat decent. However, our autonomous cars perform worse when the traffic density increases; in this case, the difference when there is irrationality or not is also more important.



(a) Traffic density $\tau = 0.25$

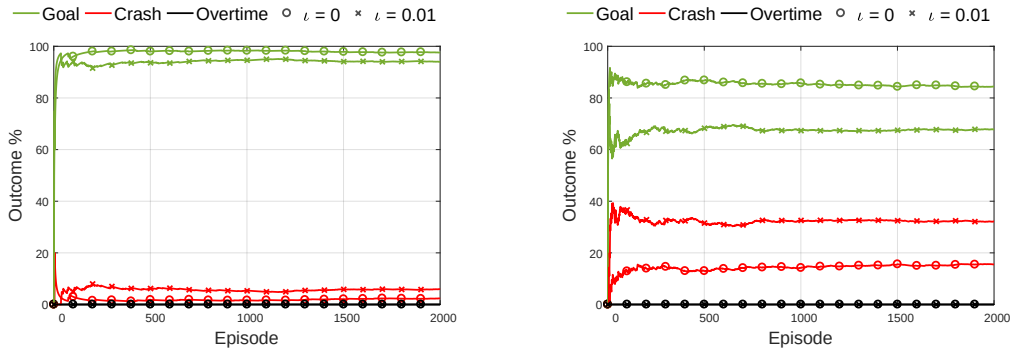
(b) Traffic density $\tau = 0.75$

Figure 11.22: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. For the training results, see Figure 11.3a.

Finally, we test a good model; namely, we test the *simple speed* model with 300 and 150 hidden neurons (Figure 11.12a). As the percentage of goal during training reached 90%, we can expect the test to show similar results: this is indeed the case, as we can see in Figure 11.23. Even with irrationality, when the traffic density is low, the agent achieves its goal more than 90% of the time. Although the performance decreases when the traffic density is higher, the agent still manages to reach its goal in most cases.

All in all, most of our tests translate well what can be observed in the training results. Moreover, the situation in which the agent performs the best is when the simulator parameters are the same as they were for the training. When slightly increasing the human drivers' irrationality, performance decreases a little. When the traffic density gets higher, the agent is less able to drive safely; this is probably due to the fact that it finds itself in situations that it never encountered during training. To convince themselves of as much, one can see the entirety of these tests in Appendix A.

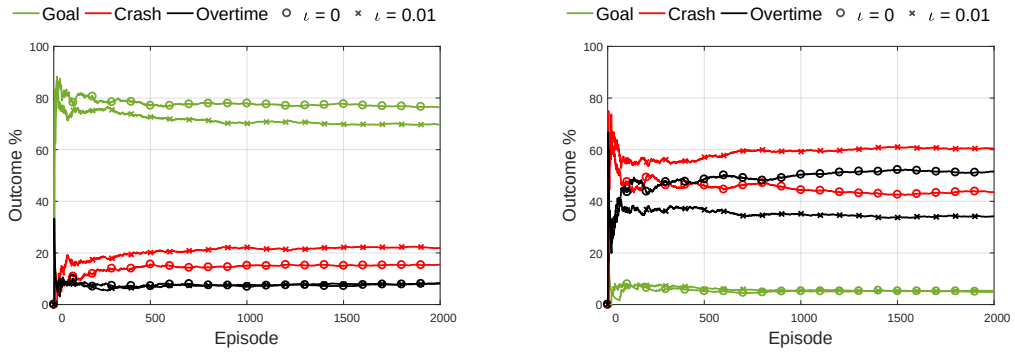
We would however like to point out one test in particular that differs from the others. In Figure 11.1a, the learning is unstable but seems to progress; the goal percentage is still increasing after the training stops. In the test results, shown in Figure 11.24, we can indeed see that when the traffic density is the same as it was for the training,



(a) Traffic density $\tau = 0.25$ (b) Traffic density $\tau = 0.75$

Figure 11.23: Testing the *simple speed* model trained on Hydra (Theano) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.12a.

the goal percentage is quite good (around 80% without irrationality and 70% with irrationality). However, for the part of the test with the higher traffic density, the performance considerably drops; the agent mostly crashes or gets overtime but almost never reaches its goal.



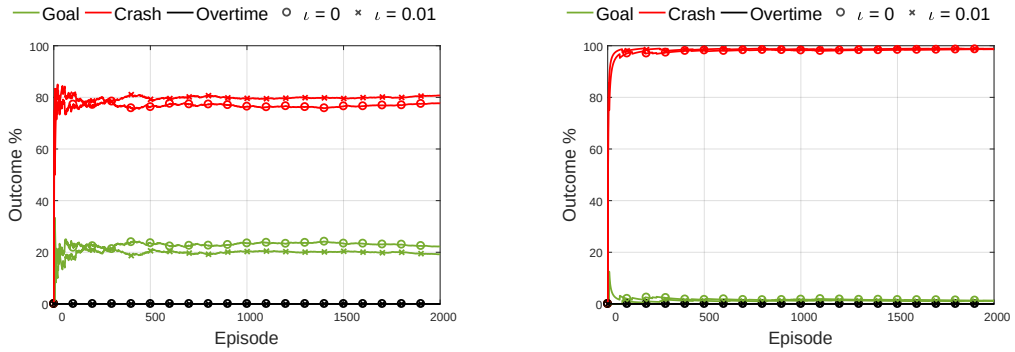
(a) Traffic density $\tau = 0.25$ (b) Traffic density $\tau = 0.75$

Figure 11.24: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.1a.

Until now, we only mentioned results regarding our three simple models. For our convolutional models, *cars speed* and *cars t*, we face two opposite situations.

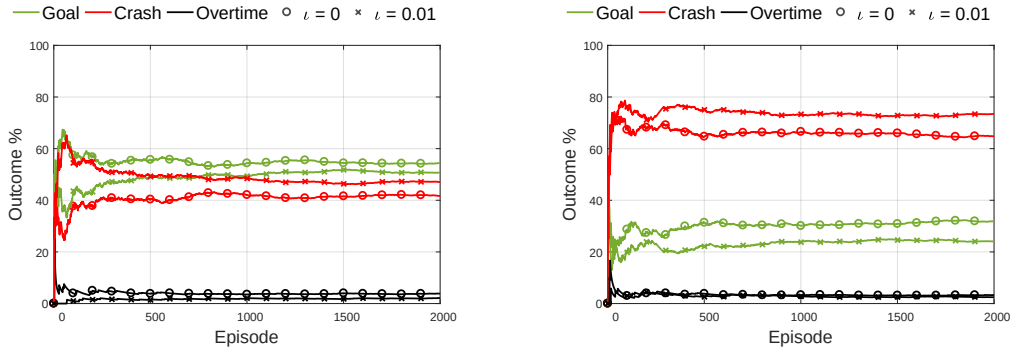
On the one hand, the test results for the *cars speed* model are completely different from what we can expect from the training results, as shown in Figure 11.25; the autonomous cars mostly crash although they performed well during the training (Figure 11.16). Moreover, the tests for the network using dropout perform better than some

of the networks whose training was better (see Figure 11.26).



(a) Traffic density $\tau = 0.25$ (b) Traffic density $\tau = 0.75$

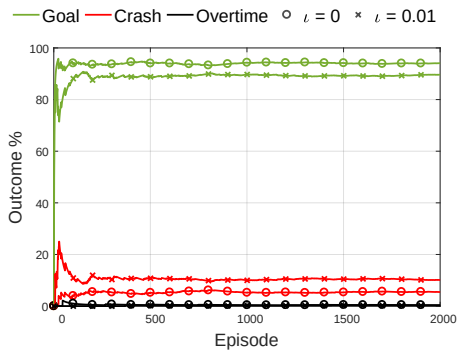
Figure 11.25: Testing the *cars speed* model with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.16.



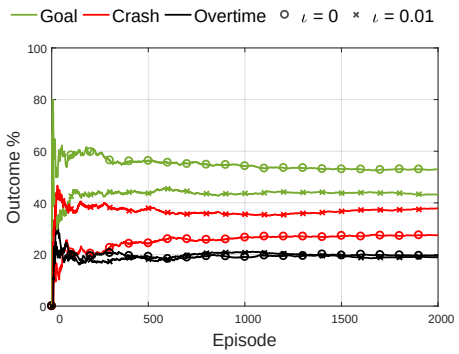
(a) Traffic density $\tau = 0.25$ (b) Traffic density $\tau = 0.75$

Figure 11.26: Testing the *cars speed* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. For the training results, see Figure 11.15.

On the other hand, the tests for the *cars t* model mostly resemble what can be expected from the training: if the training is good, it performs well when the traffic density is the same as the one used to train the model, but it does not adapt well when the traffic density increases (Figure 11.27); if it does not learn well during training, it performs poorly (Figure 11.28). The only exception is for the model with 300 and 150 hidden neurons (Figure 11.20): even though the training is promising, our tests show a bad performance, as we can see in Figure 11.29; this can be caused by the fact that this model was trained for fewer episodes than the others.

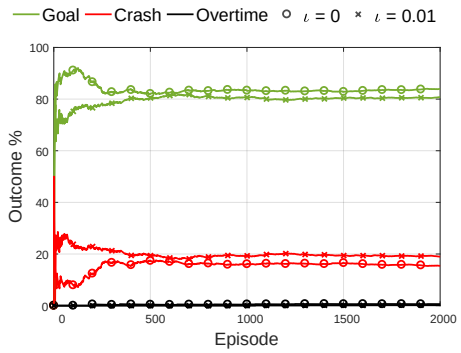


(a) Traffic density $\tau = 0.25$

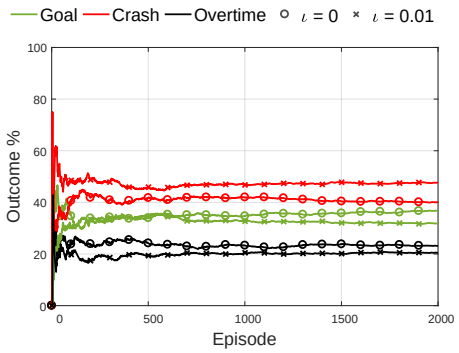


(b) Traffic density $\tau = 0.75$

Figure 11.27: Testing the *cars t* model with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.17. This model was trained and tested with an older version of Keras and Theano.

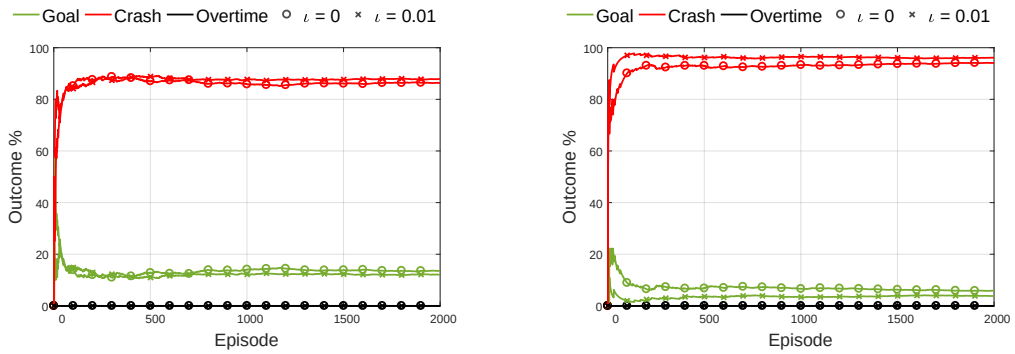


(a) Traffic density $\tau = 0.25$



(b) Traffic density $\tau = 0.75$

Figure 11.28: Testing the *cars t* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.18. This model was trained and tested with an older version of Keras and Theano.



(a) Traffic density $\tau = 0.25$

(b) Traffic density $\tau = 0.75$

Figure 11.29: Testing the *cars t* model with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.20. This model was trained and tested with an older version of Keras and Theano.

11.4 Multi-Agent Simulation

We now test our single-agent trained networks in a multi-agent environment; it means that, instead of putting only one learning agent in the system, we determine a ratio ψ of autonomous cars. Thus, we modify how we generate traffic (see Algorithm 10) so that, when a new car arrives on the highway, it is an autonomous car with a probability ψ. Algorithm 17 shows how traffic is now generated.

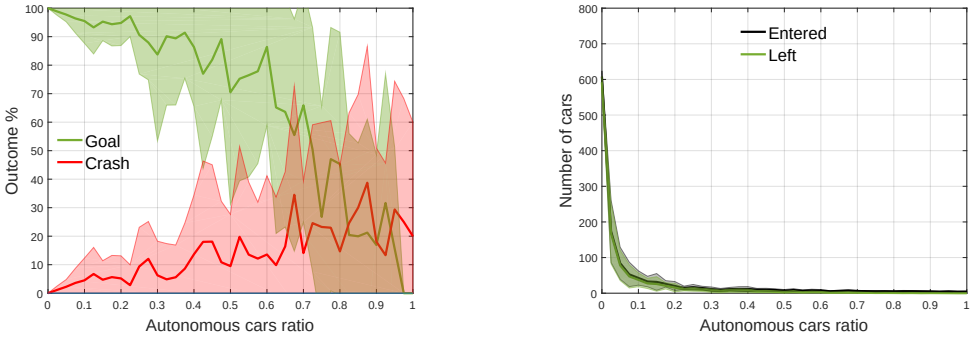
Algorithm 17: *GenerateTrafficWithAutonomousCars*

```
forall lane do
    if lane's initial position is free then
        if random < τ, with probability τ then
            if random' < ψ, with probability ψ then
                driver ← randomly initialized learning agent, autonomous car
            else
                driver ← randomly initialized human driver
            end
            driver enters lane
        end
    end
end
```

Consequently, we perform simulations with different values of this ratio and we set the neural network model of all the autonomous cars to the trained model we want to test. For each ratio value, the simulation consists of 1000 highway steps, repeated 20 times. Results thus show the average and standard deviations over these 20 repetitions. We perform these tests for all the trained models and the entirety of the results are shown in Appendix B. Note that the percentage of each outcome is computed for all the drivers in the simulation; humans and autonomous.

We can observe three main types of results.

First, there are some models that perform poorly in a multi-agent setting. This is shown in Figure 11.30. As we can see, the percentage of goal decreases as the ratio of autonomous cars increases. Moreover, we can see that the throughput almost immediately drops; this may be caused by the fact that autonomous cars crash very early in the simulation, thus creating congestion preventing new cars from entering the highway.

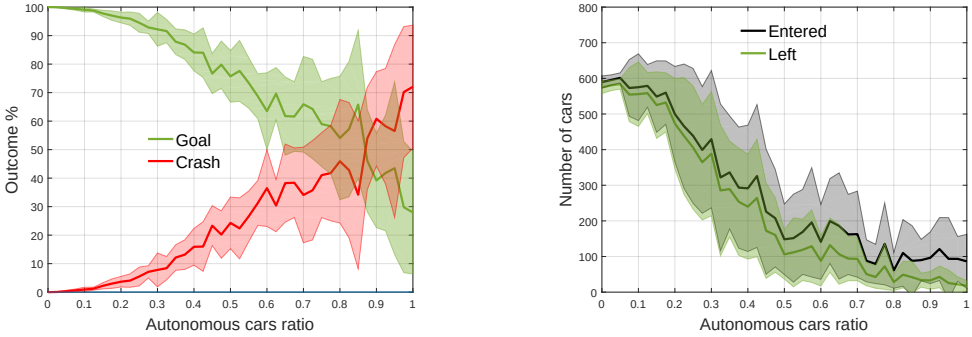


(a) Percentage of each outcome

(b) Throughput

Figure 11.30: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout in a multi-agent simulation.

Second, there are models that perform correctly in a multi-agent setting, though they clearly suffer from the increasing autonomous cars ratio, and some more than others. Such a result is shown in Figure 11.31; we can clearly see that the system has more and more crashes as the the ratio of autonomous cars increases. The throughput decreases similarly; more and more congestion means that fewer cars can enter the highway.



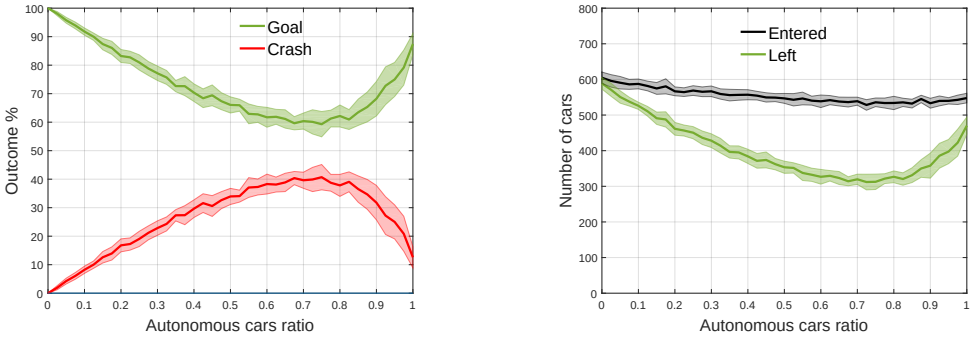
(a) Percentage of each outcome

(b) Throughput

Figure 11.31: Testing the *simple speed* model trained on the AI cluster (TensorFlow) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout in a multi-agent simulation.

Finally, some models perform well; as the autonomous cars ratio increases, the performance first decreases but starts increasing after some (high) ratio value is attained. We can observe this phenomenon in Figure 11.32. It is important to note that this happens only for the models that use dropout. A hypothesis about why this is happening is discussed later (chapter 13).

In conclusion, most of our models, even when they perform well in a single-agent environment, do not adapt correctly to a multi-agent setting. We hypothesize that it is



(a) Percentage of each outcome

(b) Throughput

Figure 11.32: Testing the *improved binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.05 in a multi-agent simulation.

because they learn to adapt to the human behavior by developing *their own different behavior*; as such, they do not evaluate correctly what the other agents do as they expect them to behave as humans do – because, as far as the learning agent making its decision is concerned, everyone else is human – even though the agents actually behave differently.

Chapter 12

Multi-Agent Setting

After training our models in a single-agent setting, we know that we can achieve a good performance. The problem now is to see whether we can achieve similar results in a multi-agent setting and, ultimately, even better results. We first train our models directly, even though they were initially designed for a single-agent environment only; afterwards, we extend these models to add information relative to the new multi-agent nature of the training environment.

12.1 Training

As we are now considering several learning agents present at the same time on the highway, we cannot arbitrarily put the (single) learning agent in the system to train it. Instead, we use a ratio of autonomous cars as explained in Algorithm 17. All the autonomous cars that enter the highway are randomly initialized in terms of speed and lane, and get the same, shared neural network model.

12.1.1 Training Scheme

We consider only two networks: the target neural network, global to all the learning agents, and the training neural network, also shared by all the agents. The training network will be updated with experiences from all the learning agents but not in an online fashion as it was the case for the single-agent training (where the network was updated at each time step according to a sampled batch of experiences). Here, experiences are stored until some number of steps I_{update}, we then use all these experiences for a batch update of the network's weights, and clear these experiences (we "forget" them). The target network is used to compute the error function for the outputs of the network and is updated every I_{target} training steps; updating the target network means that the training network actually becomes the new target network.

In the multi-agent setting, an episode cannot consist of one single learning agent's

run on the highway. For this reason, we now define a training episode as a simulation run of t^{max} highway steps. Consequently, the *overtime* outcome does not exist in the multi-agent setting. We could measure the number of time steps spent on the highway for every autonomous cars, and thus still consider the overtime outcome for them. However, as overtime means – in the single-agent setting – the end of the run, it would correspond to somehow removing the autonomous car from the highway. Since having a car disappearing all of a sudden is not realistic and could potentially affect the learning process of the other agents, we choose not to consider overtimes in this setting. The overtime was initially introduced mainly to prevent a simulation to run forever; as we define here the exact number of steps performed for each episode, we do not need such a precaution.

As we have multiple learning agents on the highway, we have, for one step of the highway, different experiences: one for each learning agent. All these experiences need to be taken into account. We consequently need to store all of them in an accumulated experience buffer $\mathcal{D}$. This process is shown in Algorithm 18.

Finally, the whole learning procedure is described by Algorithm 19. This approach is inspired by the asynchronous one-step Q-learning presented in Mnih et al. (2016).

Algorithm 18: Multi-agent learning step

// Assume existing accumulated experience buffer $\mathcal{D}$, target network θ^- and network θ

Perform one time step update of the highway H (Algorithm 12)

forall *learning agent c on the highway H* **do**

Observe the state s_t, action a_t and reward r_t of the agent for that time step, and the next state s_{t+1}

Get network outputs $\mathbf{y}$ according to $Q(s_t, a; \theta)$

Set $y_{a_t} = \begin{cases} r_k & \text{if state } s_{k+1} \text{ is final} \\ r_k + \gamma \max\limits_{a'} Q(s_{k+1}, a'; \theta^-) & \text{if state } s_{k+1} \text{ is not final} \end{cases}$

Store $(s_t, \mathbf{y})$ in accumulated experience buffer $\mathcal{D}$

end

Algorithm 19: Multi-agent learning step

```
Initialize target network θ⁻ and network θ
Initialize counter T ← 0
for episode r = 1, ..., N do
    Initialize accumulated experience buffer D
    Initialize highway H according to fixed highway parameters
    for highway step t = 1, ..., t^max do
        Perform multi-agent learning step (Algorithm 18)
        T ← T + 1
        if t mod I_update = 0 then
            Update θ with accumulated experience buffer D
            Clear accumulated experience buffer D
        end
        if T mod I_target = 0 then
            Update the target network θ⁻ ← θ
        end
    end
end
```

12.1.2 Training Parameters

The training parameters for the multi-agent setting are similar to those of the single-agent setting (see subsection 11.1.2). They are almost all the same, except for the parameters for the experience replay that are not necessary here; instead, we need the I_{update} (how often we update the training network), I_{target} (how often we update the target network) and t^{max} (the number of highway steps per episode) and the ratio of autonomous cars, as explained previously. Finally, as there is no more overtime outcome, we do not need a reward for this outcome.

12.2 Experiments

As for the single-agent experiments, we first train our five models. Since these models do not include explicit information about the other learning agents observed in the highway, we extend them to include such information and train these new models to see if they perform better. Finally, as we have fully trained (single-agent) networks, we perform a "re-training" of these trained networks.

Note that results for the three simple models – *simple binary*, *improved binary*, and *simple speed* – are only with Theano as back-end; similarly, when re-training already trained networks, said networks correspond to the Theano results presented in subsection 11.2.2. Results for the CNN models – *cars speed* and *cars t* – are only with

TensorFlow as back-end.

12.2.1 Settings

For all the experiments, most of the parameters were fixed. They are shown in Table 12.1.

(a) Simulation parameters

Simulation parameters	
Number of lanes LY	3
Lane size X	40
Number of exits E	0
Exit size S_e	5
Space size S_s	7
Crash duration D	10
Traffic density τ	0.25
Cars' maximum speed v^{max}	3
Cars' maximum acceleration α^{max}	2
Cars' irrationality ι	0

(b) Learning parameters

Learning parameters	
Number of episodes N	50000
Number of highway steps t^{max}	500
I_{update}	25
I_{target}	250
Ratio of autonomous cars ψ	0.5
Learning rate α	0.01
Discount factor γ	0.9
Agent's sight ϕ_c^+	6
Agent's ϵ	0.05

Table 12.1: Multi-agent training's fixed parameters

Among these parameters, those in common with the single-agent setting are the same as before. Thus, the same goes for the structure of the CNNs for the *cars speed* and *cars t* models: there are two convolutional layers, one with a stride of 1 and 16 filters of size 4×4, and another with a stride of 1 and 32 filters of size 2×2.

The reward system of the learning agents is also the same as before, minus the now unnecessary reward for overtime. These values are shown in Table 12.2.

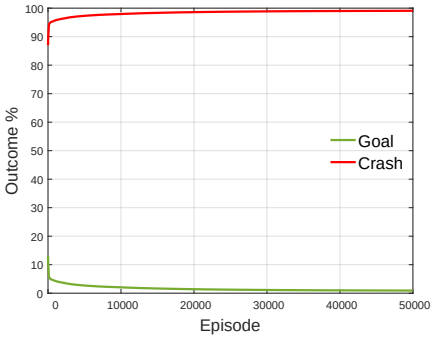
	Reward ρ	**Value**
Goal	ρ_ω	+1
Missed goal	$\rho_{\bar{\omega}}$	-0.15
Crash	ρ_f	-1
No speed penalty	ρ_0	-0.01

Table 12.2: Multi-agent training's reward system

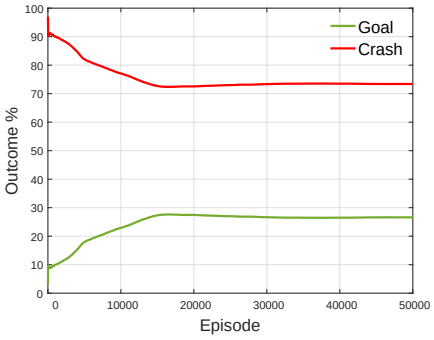
12.2.2 Training the Single-Agent Models

The first thing we can observe in our results is that the simple models with 150 and 75 hidden neurons that do not use dropout simply do not work (Figure 12.1a, Figure 12.2a, and Figure 12.3a). However, models with the same structure but that use dropout

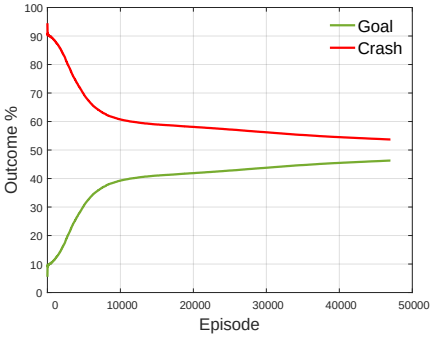
perform relatively better: they learn then converge to a low percentage of goal reached (Figure 12.1b, Figure 12.2b, and Figure 12.3b). Finally, with the 300 and 150 hidden neurons structure, only the *simple binary* model is learning (Figure 12.1c). This model is in fact the one that seems to work slightly better than the others.



(a) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

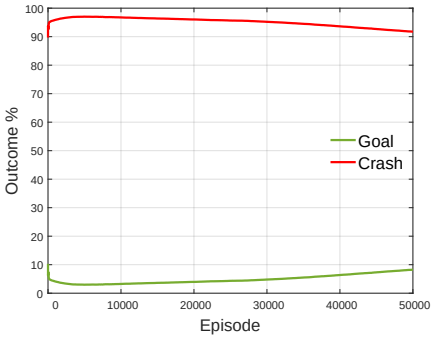


(b) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. Dropout rate of 0.5.

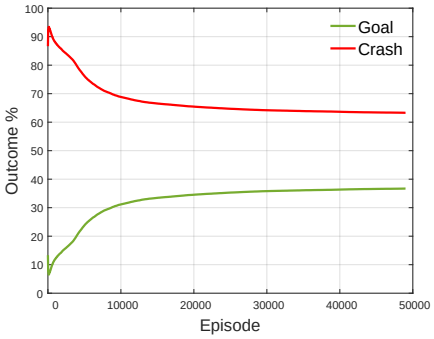


(c) Hidden structure of 300 and 150 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

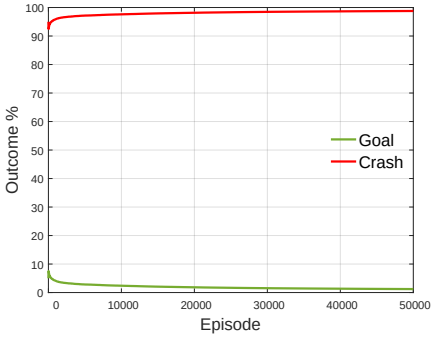
Figure 12.1: Training the single-agent model *simple binary*



(a) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

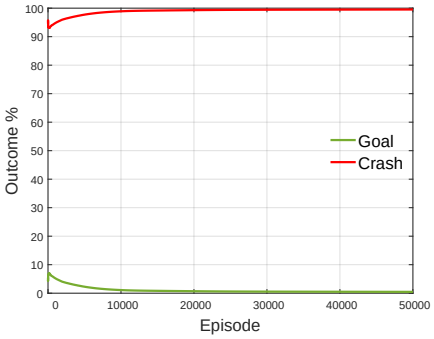


(b) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. Dropout rate of 0.5.

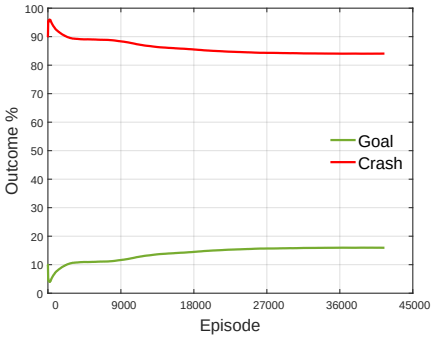


(c) Hidden structure of 300 and 150 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

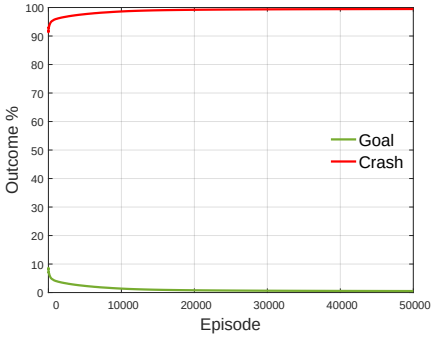
Figure 12.2: Training the single-agent model *improved binary*



(a) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. No dropout.



(b) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. Dropout rate of 0.5.

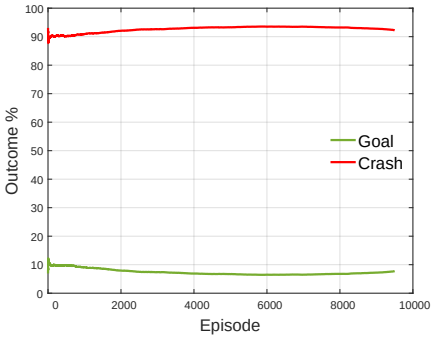


(c) Hidden structure of 300 and 150 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

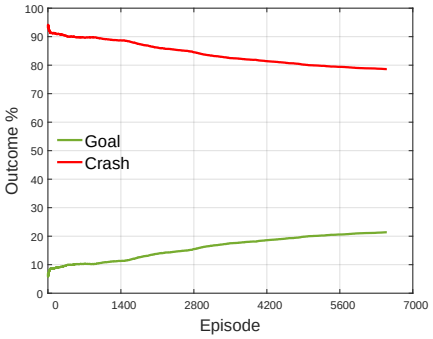
Figure 12.3: Training the single-agent model *simple speed*

For our CNN models, *cars speed* and *cars t*, our results[1] are inconclusive as they were trained for a short number of episodes, and we do not want to assume anything from the results, even though the *cars t* model with a hidden structure of 150 and 75 neurons without dropout starts to learn (Figure 12.4b).

[1] Due to problems with the clusters, these results are limited.



(a) *Cars speed* model with a hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. No dropout.



(b) *Cars t* model with a hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

Figure 12.4: Training the CNN single-agent models

12.2.3 Training the Multi-Agent Extension of the Models

We stated earlier that the models lack information regarding the multi-agent nature of this environment; indeed, according to the inputs of the networks, every car is considered the same way. However, it is logical to assume that autonomous cars would be able to recognize one another. We thus extend our five models by adding information that differentiate the human drivers from the autonomous cars.

For the *simple binary* model, this is done by separating the observation matrix O into two similar matrices; one for the human drivers and one for the autonomous cars.

Since the *improved binary* model uses the *simple binary* one, the multi-agent extension is the same.

For the *simple speed* model, as we already use real values for the speed, we decide to also use real values in the observation matrix; we now represent human drivers by 1 and autonomous cars by 0.5. This way, we do not increase the number of inputs of the network.

A similar extension is made to the *cars speed* model; the observation matrix is now filled with 1 for human drivers and 0.5 for the autonomous cars.

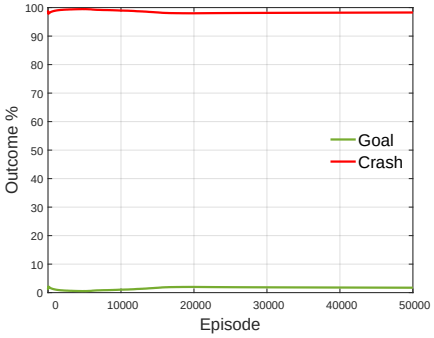
Finally, the *cars t* model is extended in the same way. The only difference is that the value 0.5 is already used in the observation matrix to represent the learning agent itself. We thus define new values to represent humans, autonomous cars and the learning agent; 1, 0.66 and 0.33 respectively.

For these last three models, the idea behind the new encoding is that we consider n possible situations observed at some position. These " situations" can be, for example, autonomous cars. We use a predefined order of all the situations so that we can encode the situation as its number (in the order) divided by the total number of situations. For example, for the *simple speed* model, the car can either see autonomous cars (situation

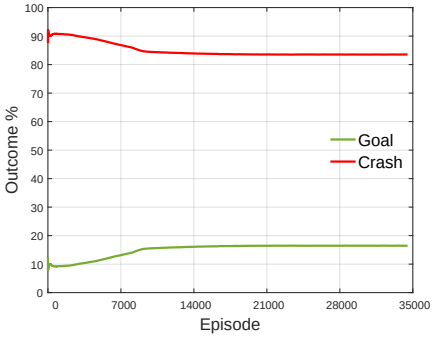
1) or human drivers (situation 2). There are thus 2 situations: autonomous cars are represented by $\frac{1}{2} = 0.5$ and human drivers by $\frac{2}{2} = 1$.

Results

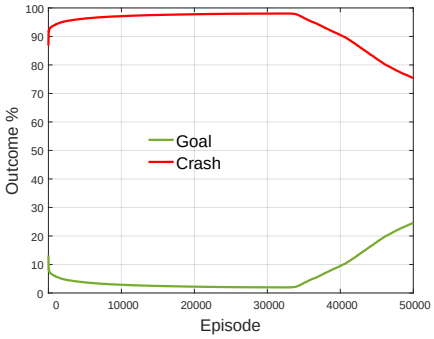
For our three simple models, we can see that using their multi-agent version does not really change how the training goes for the hidden structure of 150 and 75 neurons with dropout (Figure 12.5b, Figure 12.6b, and Figure 12.7b). Other results are quite different depending on the model: the hidden structure of 150 and 75 neurons without dropout does not learn for the *simple binary* and *simple speed* models while it works quite well for the *improved binary* model with the percentage of goal surpassing the percentage of crash (Figure 12.6a). Similarly, the hidden structure of 300 and 150 neurons without dropout does not work for the *improved binary* model while it does for the *simple binary* one (Figure 12.5c)[2].



(a) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. No dropout.



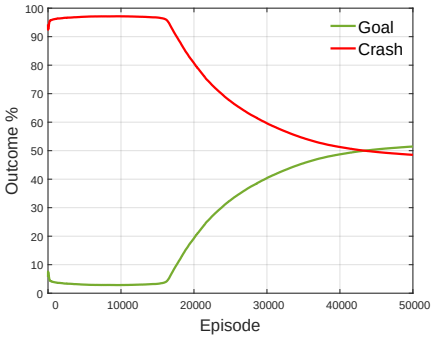
(b) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. Dropout rate of 0.5.



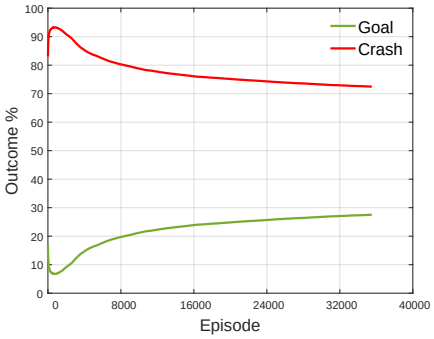
(c) Hidden structure of 300 and 150 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

Figure 12.5: Training the multi-agent extended model *simple binary*

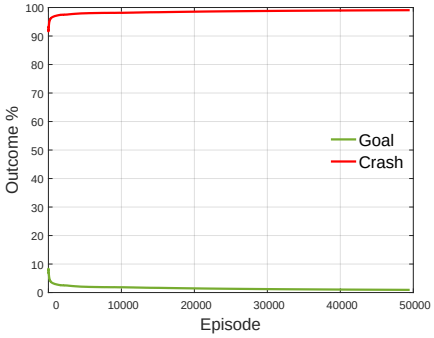
[2]Results for the *simple speed* model with this structure are missing due to a mishap on the clusters.



(a) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

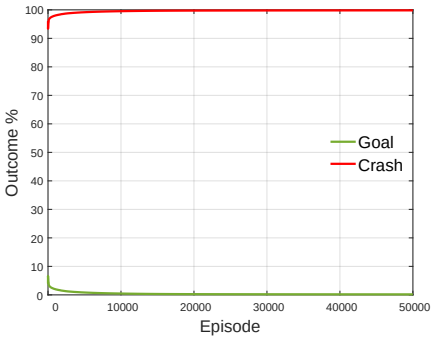


(b) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. Dropout rate of 0.5.

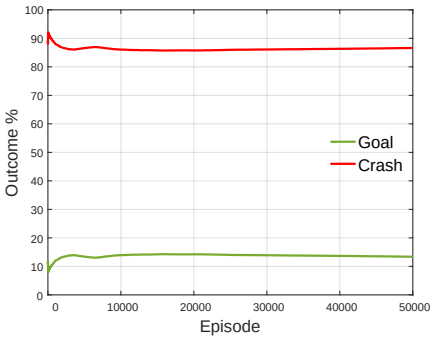


(c) Hidden structure of 300 and 150 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

Figure 12.6: Training the multi-agent extended model *improved binary*



(a) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. No dropout.



(b) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. Dropout rate of 0.5.

Figure 12.7: Training the multi-agent extended model *simple speed*

For the *cars t* model with 150 and 75 hidden neurons without dropout[3], we can see that the model starts to learn and apparently keeps learning (Figure 12.8).

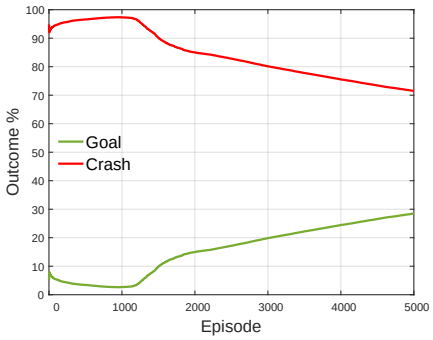
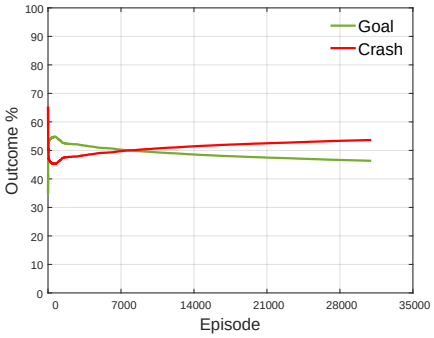


Figure 12.8: Training the multi-agent extended model *cars t* with 150 and 75 hidden neurons. No dropout.

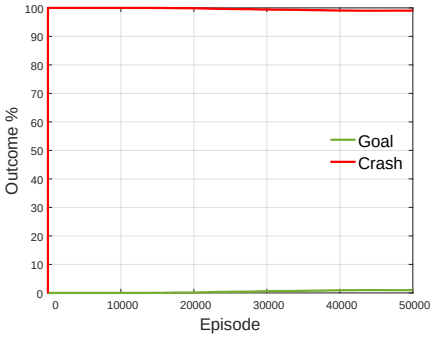
12.2.4 Re-Training the Single-Agent Trained Models

When re-training single-agent trained networks, we can see that the multi-agent performance globally corresponds to the initial, single-agent one of the initial model; for example, we have a poor performance for the single-agent *simple binary* model with 300 and 150 hidden neurons (Figure 11.4a) and a similarly poor performance for the multi-agent model that re-trained it (Figure 12.9b).

One noticeable result is the re-training of the *improved binary* model with a hidden structure of 300 and 150 neurons without dropout (Figure 12.10b): starting from the trained network, the percentage of goal is around 30% and increases to more than 95%. This is the only model that actually learns and increases its performance.



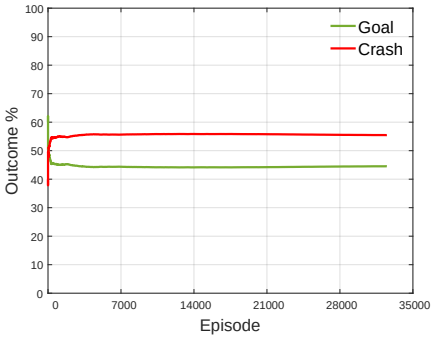
(a) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. Dropout rate of 0.5.



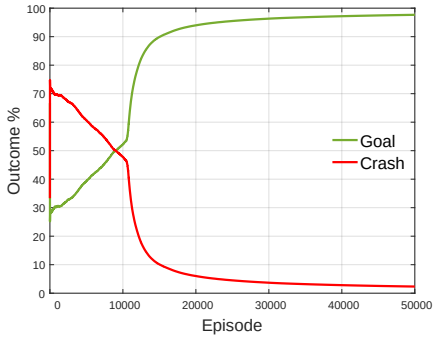
(b) Hidden structure of 300 and 150 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

Figure 12.9: Re-training a fully trained single-agent *simple binary* model

[3]Problems with the cluster prevented us from obtaining more results.

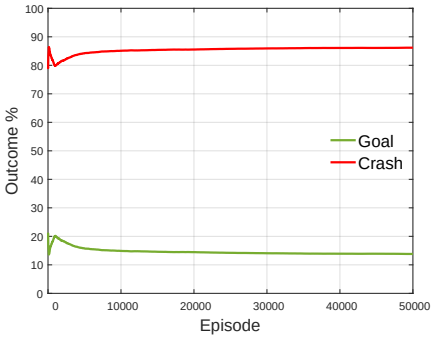


(a) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. Dropout rate of 0.5.

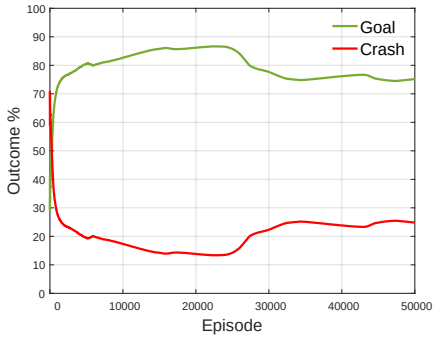


(b) Hidden structure of 300 and 150 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

Figure 12.10: Re-training a fully trained single-agent *improved binary* model



(a) Hidden structure of 150 and 75 neurons, with *tanh* and *linear* activation functions respectively. Dropout rate of 0.5.



(b) Hidden structure of 300 and 150 neurons, with *tanh* and *linear* activation functions respectively. No dropout.

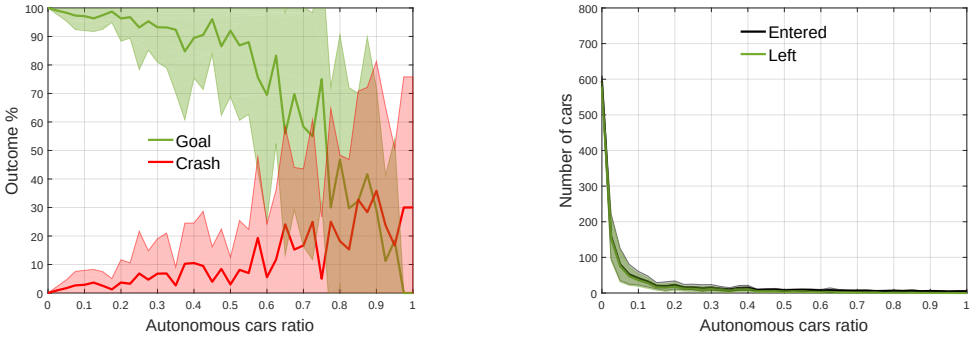
Figure 12.11: Re-training a fully trained single-agent *simple speed* model

12.3 Tests

For all the trained models obtained, we perform a test similar to the one defined in section 11.4. We thus test how the models perform for different autonomous cars ratio; for each ratio value, we do 20 runs of 1000 highway steps.

For models that perform poorly, such as the one shown in Figure 12.1a, we can see that the trained self-driving cars actually exhibit a good performance when the ratio of autonomous cars is low (Figure 12.12). However, this performance decreases as the ratio increases. Unfortunately, as we can see in the throughput, there are almost no cars on the highway as soon as the autonomous cars ratio goes up; as such, this undermines the percentage of each outcome results as they may represent the outcomes of the first few cars that arrived on the highway before some important congestion completely blocked

the highway's access.

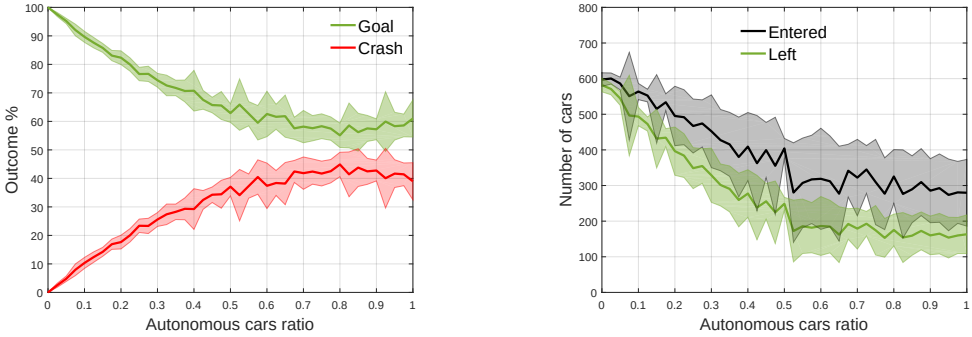


(a) Percentage of each outcome

(b) Throughput

Figure 12.12: Testing the *simple binary* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout.

For models that perform somewhat correctly, as is shown in Figure 12.2a, the performance is good although it decreases a little when the autonomous cars ratio goes up (Figure 12.13). We can see that the performance is really good for low autonomous cars ratios, and, although it decreases, stays relatively good for higher ratios.



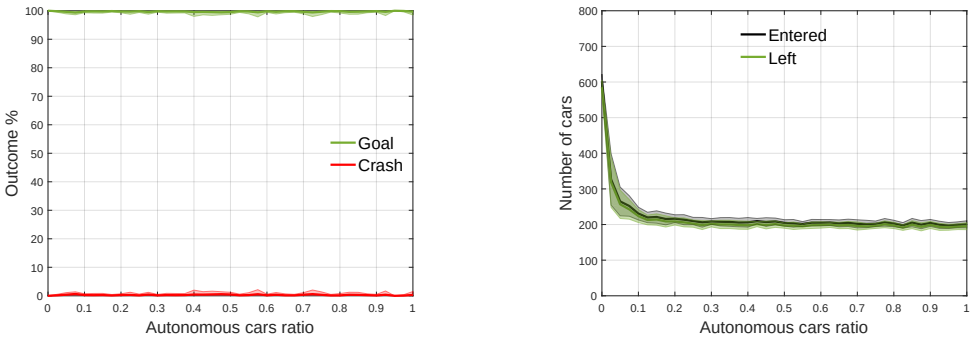
(a) Percentage of each outcome

(b) Throughput

Figure 12.13: Testing the *improved binary* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5.

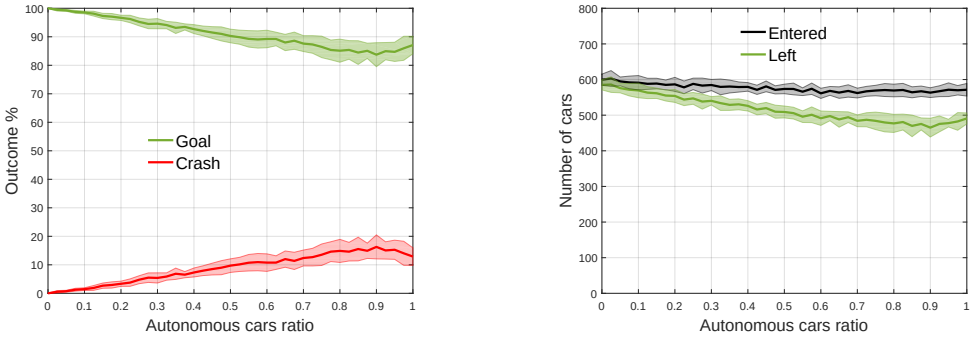
Finally, the models with a good training performance – mostly just the re-training of single-agent networks – show a good and consistent performance regardless of the ratio of self-driving cars. Such results are shown in Figure 12.14 and Figure 12.15. The difference of throughput is, however, surprising.

All the results can be found in Appendix C.



(a) Percentage of each outcome (b) Throughput

Figure 12.14: Testing the model using a fully trained *improved binary* model network with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout.



(a) Percentage of each outcome (b) Throughput

Figure 12.15: Testing the model using a fully trained *simple speed* model network with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout.

Part V

Conclusion

Chapter 13

Discussion

First, we designed and implemented a traffic simulator that we wanted both simple and realistic. Unfortunately, these two characteristics are not always compatible. Finding a good balance is thus rather difficult and requires us to make some arbitrary choices that may be different from what other people would have naturally chosen. Creating a traffic simulator also relies on our own observations of real-life traffic; observations that can vary depending on the people and their own traffic experience. Whether the final model corresponds to what the reader would have implemented or not, our traffic simulator remains a useful, fully parameterizable learning environment.

In section 11.2, we showed how our models perform when trained in a single-agent setting. One important thing we can observe in these results is that there is, for the same model, sometimes a great divergence in the training depending on the back-end used (Theano or TensorFlow). We do not know exactly why this can happen. Maybe this is simply due to the fact that the weights of the models are randomly initialized and the training sometimes leads to a local minimum or sometimes experiences catastrophic forgetting. As the only type of models where there is no sensible difference between Theano and TensorFlow is the type of models that use dropout, the problem may be due to overfitting, as dropout helps to prevent it.

In most cases, the models with the greater number of hidden neurons perform better although they do not adapt very well when we use them in a multi-agent setting. We mentioned earlier why we think this happens: self-driving cars learn some behavior that allows them to deal well with the behavior of the human drivers but this learned behavior is probably different from the humans' behavior and artificial agents do not learn how to respond to their own behavior in a single-agent setting.

There are, however, models for which the performance increases when the ratio of autonomous cars becomes sufficiently high. This result is interesting as these models were not necessarily the ones that performed the best in the single-agent setting, but they seem to exhibit better skills to deal with other drivers around them, regardless of

the actual behavior of said drivers. This could be because these agents mostly learn to be safe (i.e. staying far from the other cars).

For the multi-agent training, we can see that most models are simply not working; they probably lack important information or the way of encoding the information is simply not good. However, when using as a starting model a fully trained (single-agent) network, agents perform correctly if that single-agent network showed good performance in its own training.

As our initial models lack information relevant to the multi-agent setting, we extended these models in order to add some information; namely, we extended the models in such a way that the learning agents could see other autonomous cars by clearly identifying them in the inputs of the network. As it is fair to assume that real-life autonomous cars could possess some sort of transmitter to send and receive messages from other autonomous cars, making the agents able to identify the other autonomous cars seems only natural. Some of these new models do perform better than their single-agent counterpart.

Overall, we noticed no real difference between our models in the single-agent setting. They all seem to perform well with a sufficient number of hidden neurons even though this structure does not adapt too well to a multi-agent setting. However, the networks that use dropout seem to be more capable of adapting to multiple learning agents on the highway, although their single-agent performance was not the best. For the multi-agent environment, the model that apparently works best for all the training strategies – single-agent model, multi-agent extension, and re-training – is the *improved binary* model. The difference with this model is that it encodes information about sequential time steps – thus allowing the agents to extract information and learn from the underlying dynamics of the highway.

Chapter 14

Conclusion

We wondered if it was possible to train self-driving cars capable of driving safely and consequently designed different neural network models; we designed three simple networks – without convolutions – and two convolutional networks in order to try to solve this problem with deep reinforcement learning.

We first trained our models for a single-agent environment to see if a learning agent was able to learn the human drivers' behavior and develop a good driving strategy. We showed that it was indeed possible and even achieved high performances. However, as the training is done in one fixed setting of the highway, the models' aptitude to adapt to a new setting of the environment – higher traffic density, human drivers' irrationality – is not always good.

Afterwards, we tested our single-agent trained models in a multi-agent environment where all the autonomous cars use this trained model. We saw that, for most models, these results were unsatisfactory, which is probably caused by the fact that the autonomous cars' learned behavior differs from the behavior of human drivers – but they only learned how to deal with human drivers.

Finally, we trained models in a multi-agent setting in three different ways. We first considered the exact same models that we used for the single-agent environment and showed that most of them, even the ones that had a good performance in a single-agent setting, did not learn efficiently. We then modified these models to add information regarding the multi-agent nature of the environment: a distinction between the human drivers and the autonomous cars. We showed that these models performed better than their single-agent version, although we did not achieve the same level of performance as we did for the single-agent environment. Lastly, we used networks that were already trained in a single-agent setting as the starting models for the multi-agent training. The best performances over all the multi-agent training experiments we did were obtained with these "re-trainings" of single-agent models.

In conclusion, making self-driving cars learn to drive safely is possible. However, it

is easier to do so in a single-agent setting. Indeed, in a multi-agent setting, where all the learning agents start to learn from scratch, they must do so in situations where some of the drivers in their vicinity – drivers that influence their action and their next state – are also learning and consequently potentially doing random actions.

Chapter 15

Future work

First, we only considered one possible setting of the highway, a simple one without exits; consequently, we would like to reproduce the same experiments for a version of the highway with exits, to see how the agents perform when they have different possible goals.

The next logical step of this work would be to do more trainings by varying all the possible parameters in order to find optimal values for these parameters. A tool such as *irace*[1] could be used to optimize the training configuration.

All our models were trained with a fixed traffic density, thus limiting the number of situations the learning agents can encounter. An interesting improvement would be to use a varying traffic density during the training; furthermore, the traffic density could follow some probability distribution representing something similar to the rush hours in traffic.

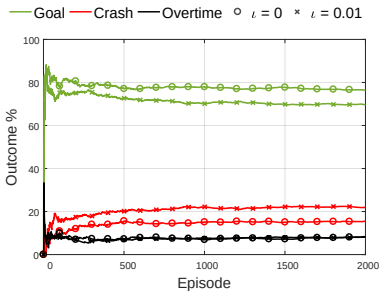
As far as our traffic simulator is concerned, we can think of multiple possible improvements that would make the whole system more realistic. For example, introducing the notion of good or bad drivers; a driver's aptitude to drive could impact their irrationality or their sight. Another improvement would be the addition of other types of cars in the simulator: namely, trucks, which would block the field of view of the cars in their vicinity. Finally, making a continuous version of the highway could also be interesting.

[1] `http://iridia.ulb.ac.be/irace/`

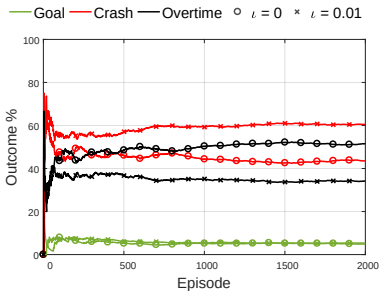
Appendix A

Single-Agent Testing Results

A.1 *Simple Binary* Model

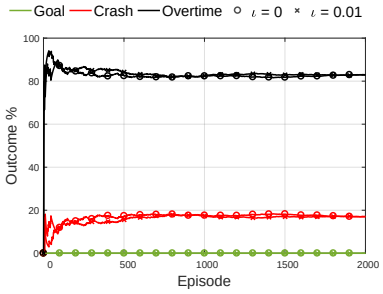


(a) Traffic density $\tau = 0.25$

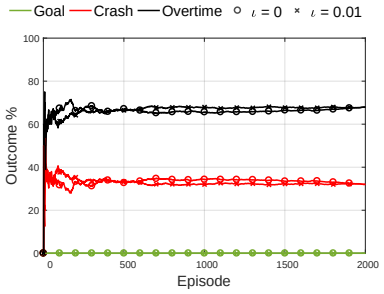


(b) Traffic density $\tau = 0.75$

Figure A.1: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.1a.

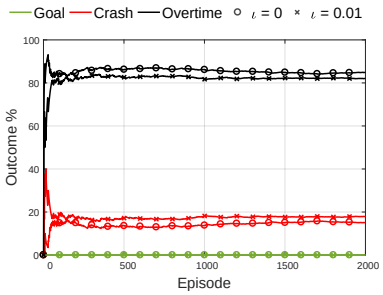


(a) Traffic density $\tau = 0.25$

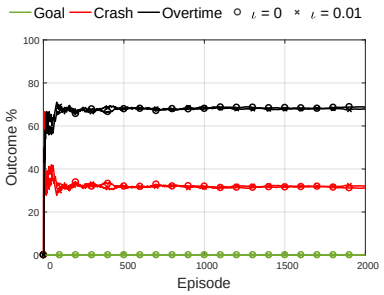


(b) Traffic density $\tau = 0.75$

Figure A.2: Testing the *simple binary* model trained on the AI cluster (TensorFlow) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.1b.

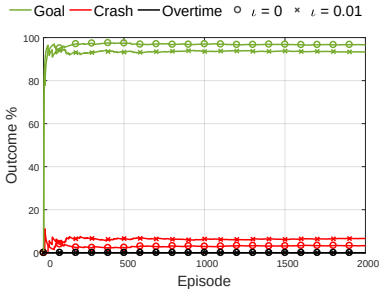


(a) Traffic density $\tau = 0.25$

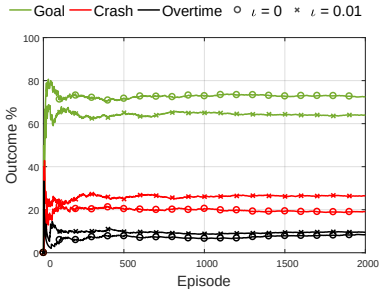


(b) Traffic density $\tau = 0.75$

Figure A.3: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.2a.

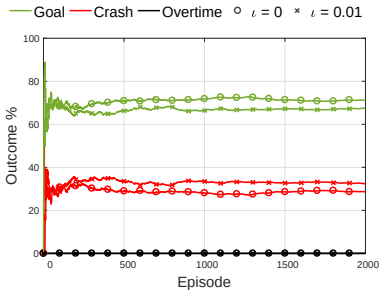


(a) Traffic density $\tau = 0.25$

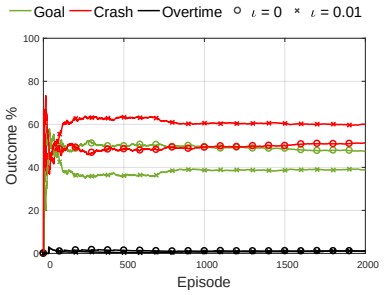


(b) Traffic density $\tau = 0.75$

Figure A.4: Testing the *simple binary* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.2b.

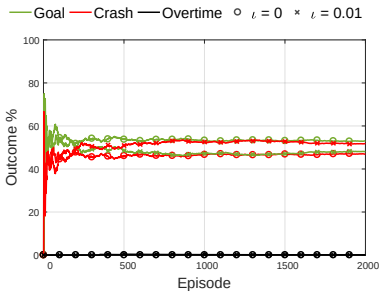


(a) Traffic density $\tau = 0.25$

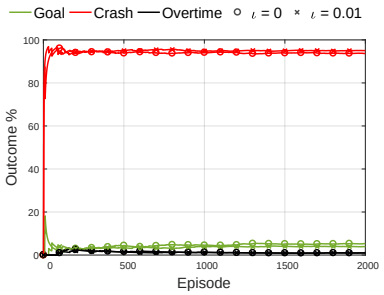


(b) Traffic density $\tau = 0.75$

Figure A.5: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. For the training results, see Figure 11.3a.

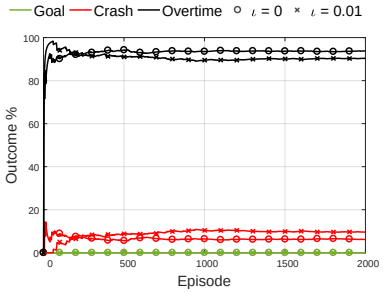


(a) Traffic density $\tau = 0.25$

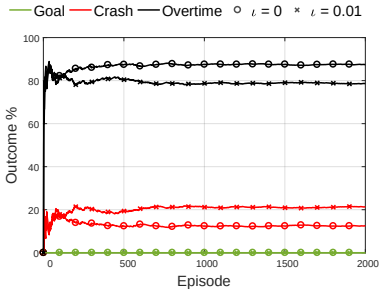


(b) Traffic density $\tau = 0.75$

Figure A.6: Testing the *simple binary* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. For the training results, see Figure 11.3b.

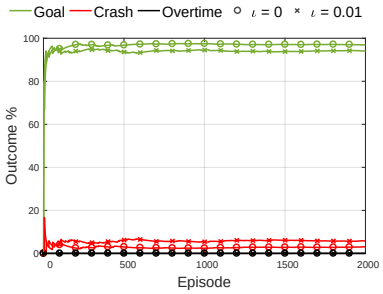


(a) Traffic density $\tau = 0.25$

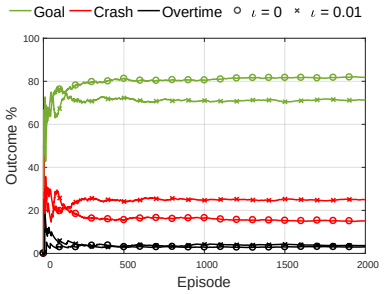


(b) Traffic density $\tau = 0.75$

Figure A.7: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.4a.



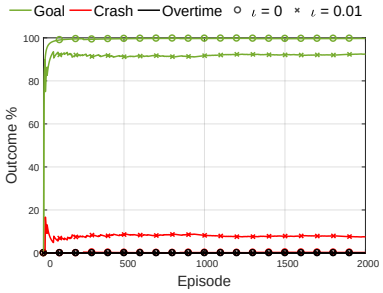
(a) Traffic density $\tau = 0.25$



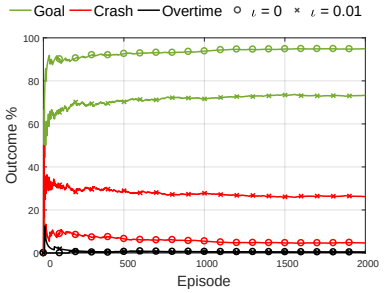
(b) Traffic density $\tau = 0.75$

Figure A.8: Testing the *simple binary* model trained on the AI cluster (TensorFlow) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.4b.

A.2 *Improved Binary* Model

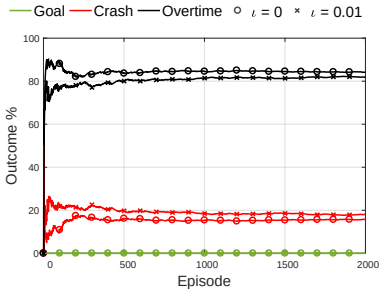


(a) Traffic density $\tau = 0.25$

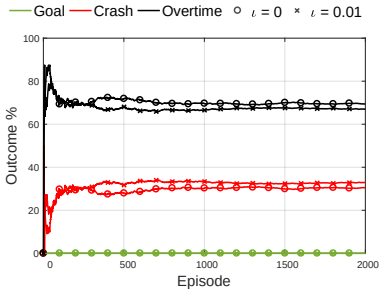


(b) Traffic density $\tau = 0.75$

Figure A.9: Testing the *improved binary* model trained on Hydra (Theano) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.5a.

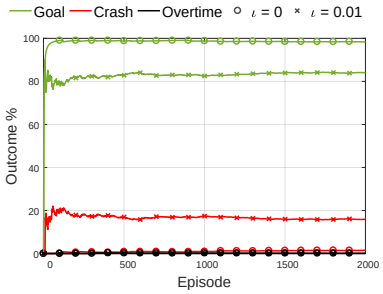


(a) Traffic density $\tau = 0.25$

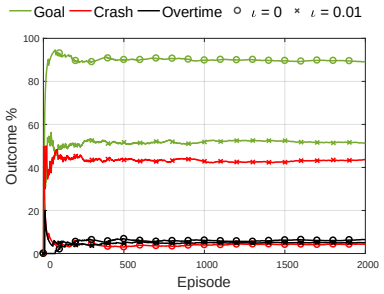


(b) Traffic density $\tau = 0.75$

Figure A.10: Testing the *improved binary* model trained on the AI cluster (TensorFlow) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.5b.

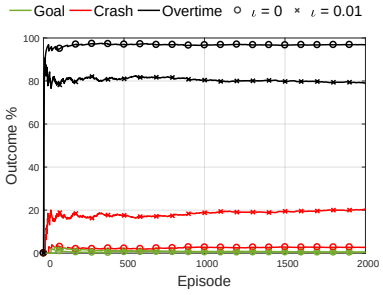


(a) Traffic density $\tau = 0.25$

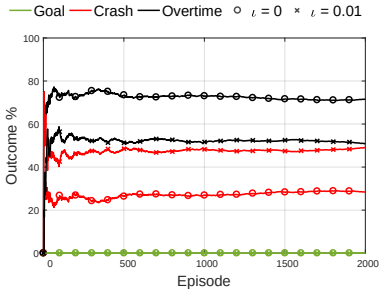


(b) Traffic density $\tau = 0.75$

Figure A.11: Testing the *improved binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.6a.

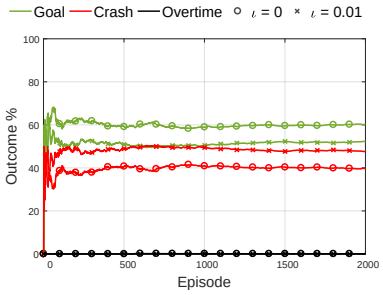


(a) Traffic density $\tau = 0.25$

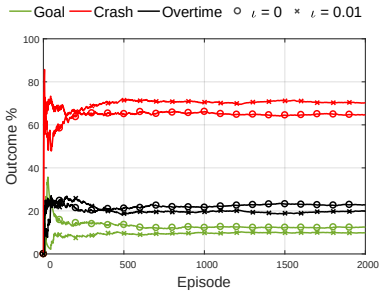


(b) Traffic density $\tau = 0.75$

Figure A.12: Testing the *improved binary* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.6b.

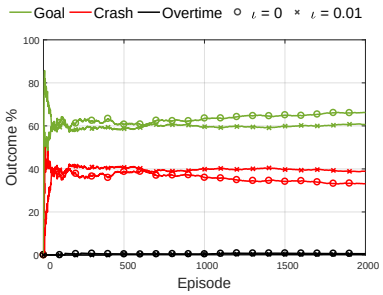


(a) Traffic density $\tau = 0.25$

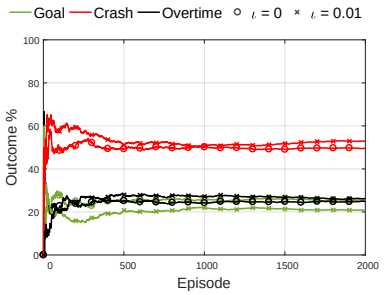


(b) Traffic density $\tau = 0.75$

Figure A.13: Testing the *improved binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. For the training results, see Figure 11.7a.

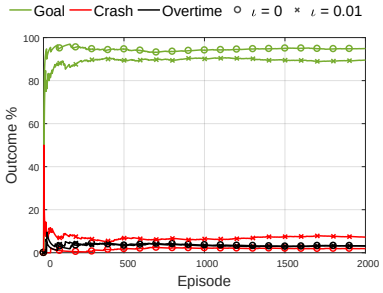


(a) Traffic density $\tau = 0.25$

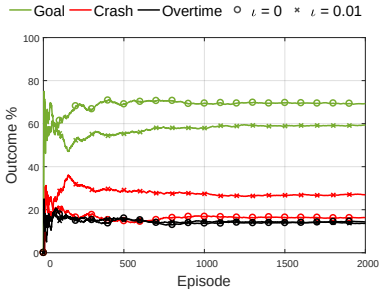


(b) Traffic density $\tau = 0.75$

Figure A.14: Testing the *improved binary* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. For the training results, see Figure 11.7b.

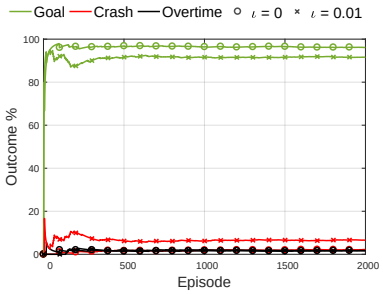


(a) Traffic density $\tau = 0.25$

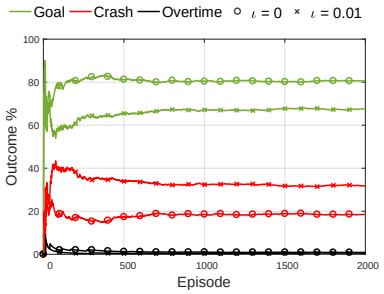


(b) Traffic density $\tau = 0.75$

Figure A.15: Testing the *improved binary* model trained on Hydra (Theano) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.8a.



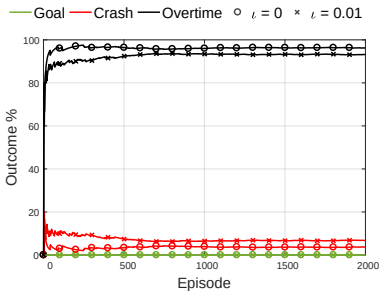
(a) Traffic density $\tau = 0.25$



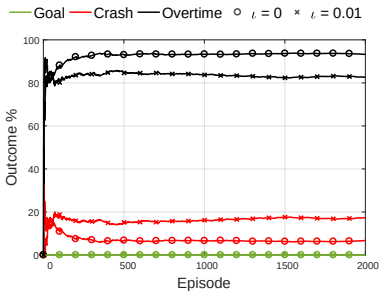
(b) Traffic density $\tau = 0.75$

Figure A.16: Testing the *improved binary* model trained on the AI cluster (TensorFlow) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.8b.

A.3 *Simple Speed* Model

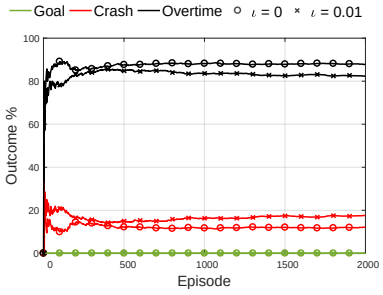


(a) Traffic density $\tau = 0.25$

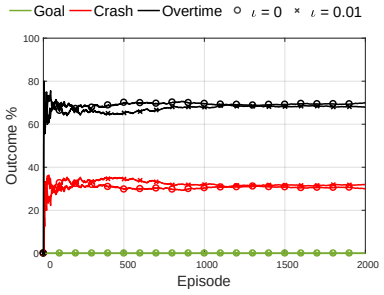


(b) Traffic density $\tau = 0.75$

Figure A.17: Testing the *simple speed* model trained on Hydra (Theano) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.9a.

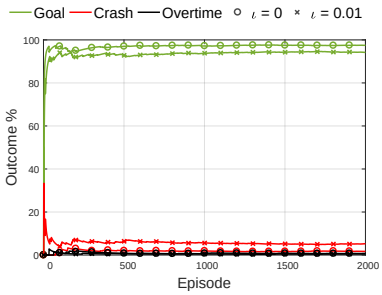


(a) Traffic density $\tau = 0.25$

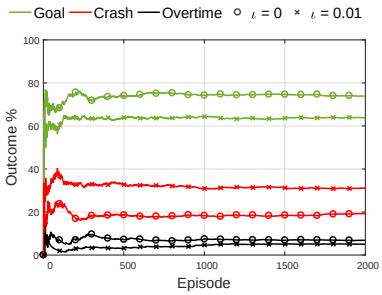


(b) Traffic density $\tau = 0.75$

Figure A.18: Testing the *simple speed* model trained on the AI cluster (TensorFlow) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.9b.

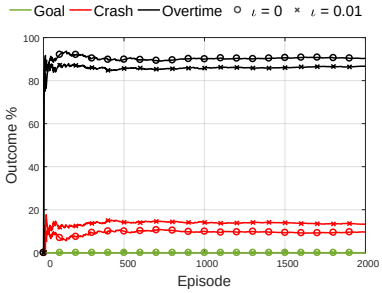


(a) Traffic density $\tau = 0.25$

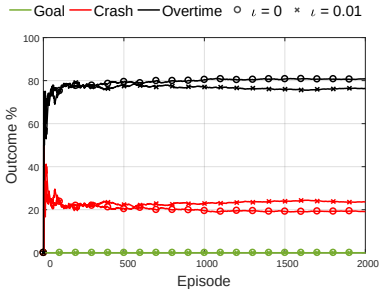


(b) Traffic density $\tau = 0.75$

Figure A.19: Testing the *simple speed* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.10a.

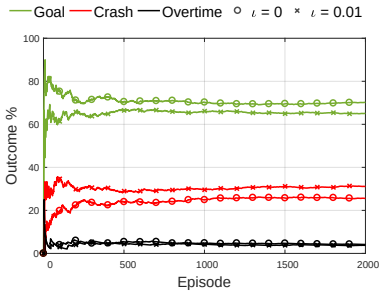


(a) Traffic density $\tau = 0.25$

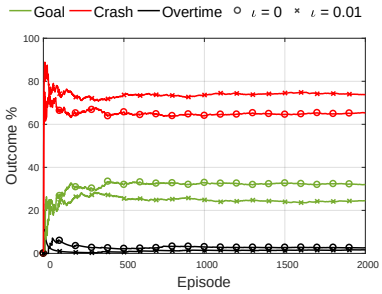


(b) Traffic density $\tau = 0.75$

Figure A.20: Testing the *simple speed* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.10b.

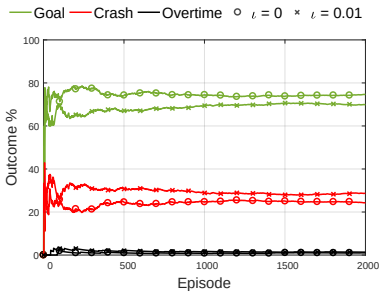


(a) Traffic density $\tau = 0.25$

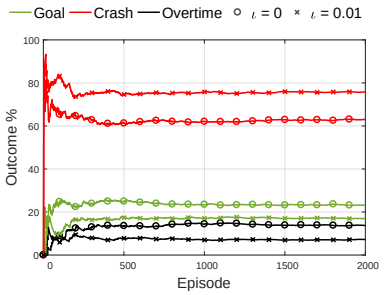


(b) Traffic density $\tau = 0.75$

Figure A.21: Testing the *simple speed* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. For the training results, see Figure 11.11a.

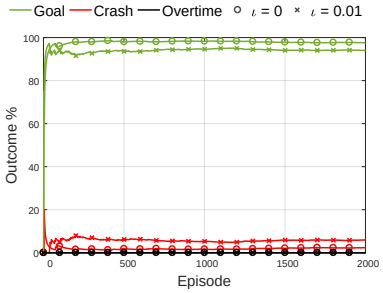


(a) Traffic density $\tau = 0.25$

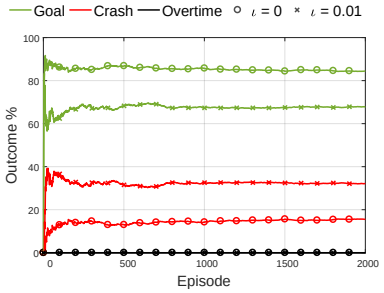


(b) Traffic density $\tau = 0.75$

Figure A.22: Testing the *simple speed* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. For the training results, see Figure 11.11b.

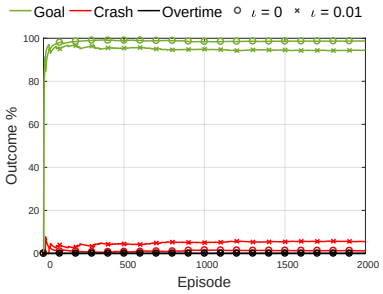


(a) Traffic density $\tau = 0.25$

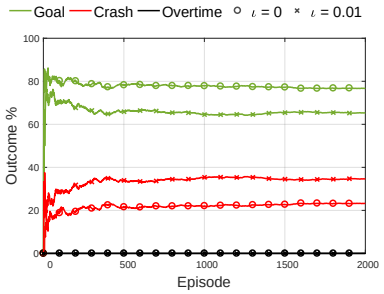


(b) Traffic density $\tau = 0.75$

Figure A.23: Testing the *simple speed* model trained on Hydra (Theano) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.12a.



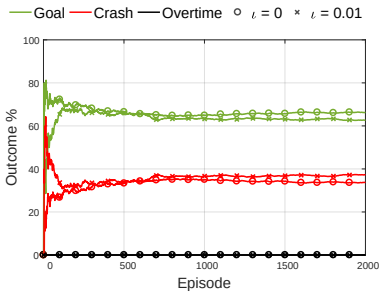
(a) Traffic density $\tau = 0.25$



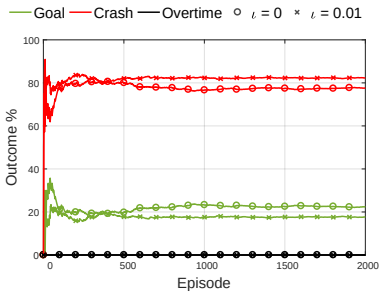
(b) Traffic density $\tau = 0.75$

Figure A.24: Testing the *simple speed* model trained on the AI cluster (TensorFlow) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.12b.

A.4 *Cars Speed* Model

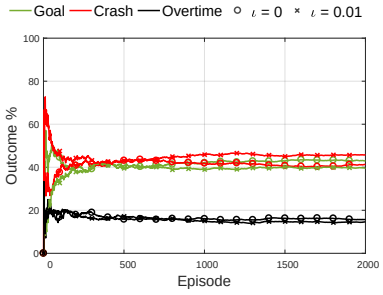


(a) Traffic density $\tau = 0.25$

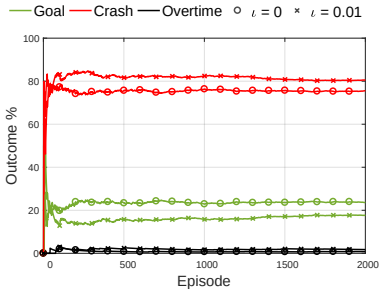


(b) Traffic density $\tau = 0.75$

Figure A.25: Testing the *cars speed* model with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.13.

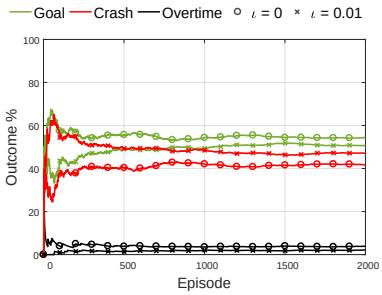


(a) Traffic density $\tau = 0.25$

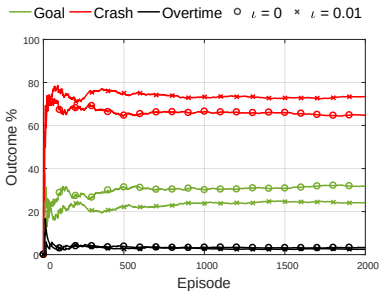


(b) Traffic density $\tau = 0.75$

Figure A.26: Testing the *cars speed* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.14.

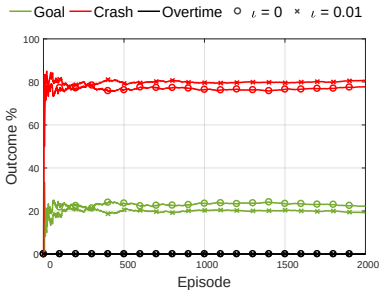


(a) Traffic density $\tau = 0.25$

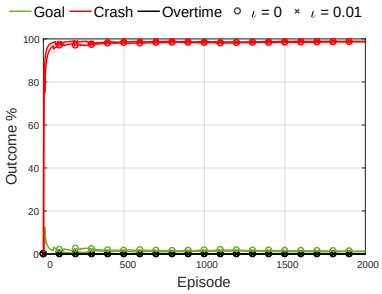


(b) Traffic density $\tau = 0.75$

Figure A.27: Testing the *cars speed* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. For the training results, see Figure 11.15.



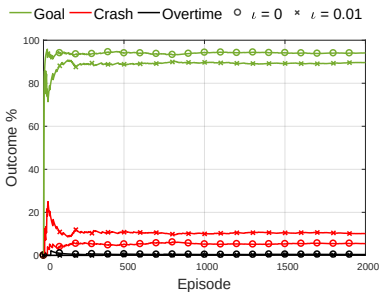
(a) Traffic density $\tau = 0.25$



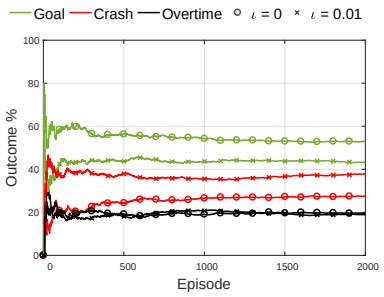
(b) Traffic density $\tau = 0.75$

Figure A.28: Testing the *cars speed* model with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.16.

A.5 *Cars t* Model

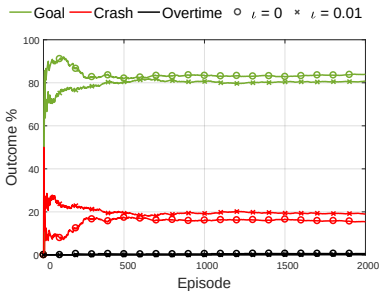


(a) Traffic density $\tau = 0.25$

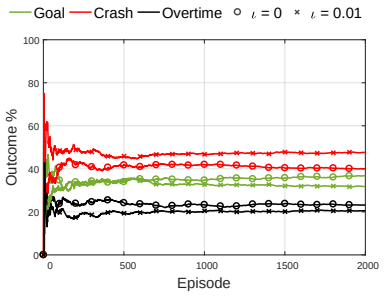


(b) Traffic density $\tau = 0.75$

Figure A.29: Testing the *cars t* model with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.17. This model was trained and tested with an older version of Keras and Theano.

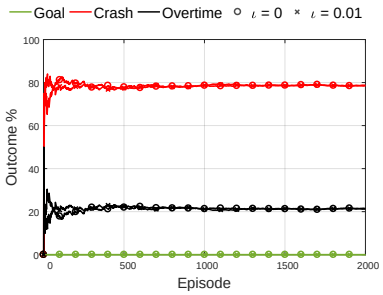


(a) Traffic density $\tau = 0.25$

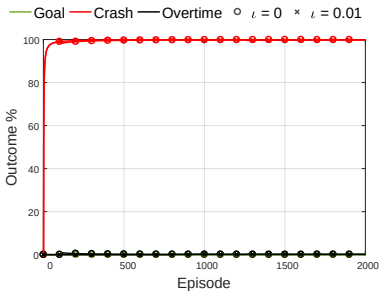


(b) Traffic density $\tau = 0.75$

Figure A.30: Testing the *cars t* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.18. This model was trained and tested with an older version of Keras and Theano.

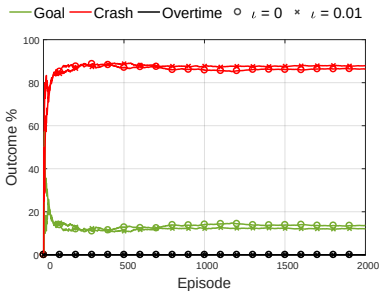


(a) Traffic density $\tau = 0.25$

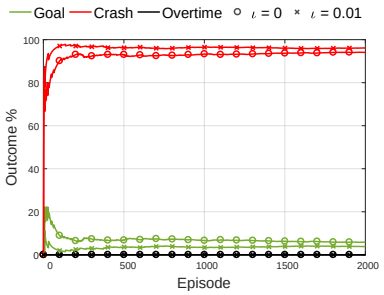


(b) Traffic density $\tau = 0.75$

Figure A.31: Testing the *cars t* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5. For the training results, see Figure 11.19. This model was trained and tested with an older version of Keras and Theano.



(a) Traffic density $\tau = 0.25$



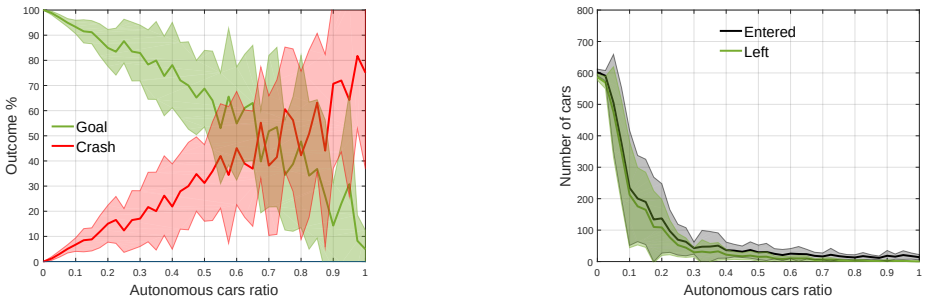
(b) Traffic density $\tau = 0.75$

Figure A.32: Testing the *cars t* model with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout. For the training results, see Figure 11.20.

Appendix B

Single-Agent Trained Networks in Multi-Agent Simulator

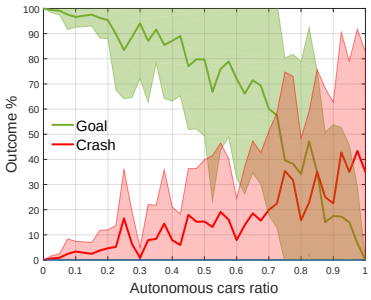
B.1 *Simple Binary* Model



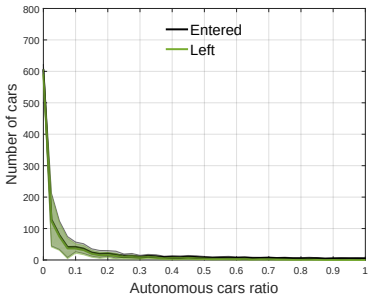
(a) Percentage of each outcome

(b) Throughput

Figure B.1: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

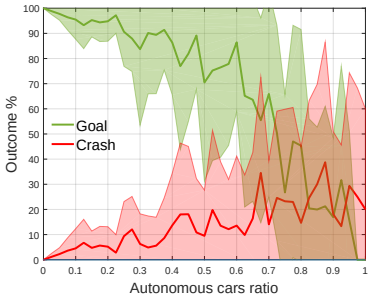


(a) Percentage of each outcome

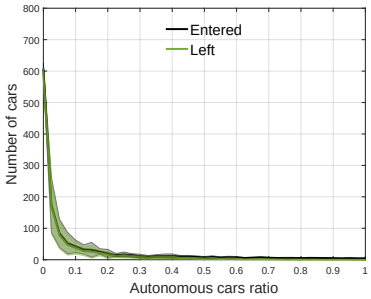


(b) Throughput

Figure B.2: Testing the *simple binary* model trained on the AI cluster (TensorFlow) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

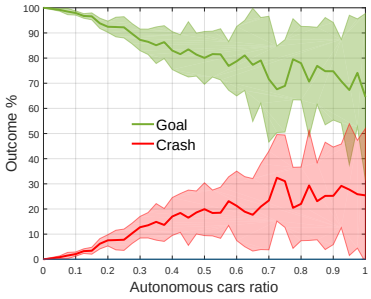


(a) Percentage of each outcome

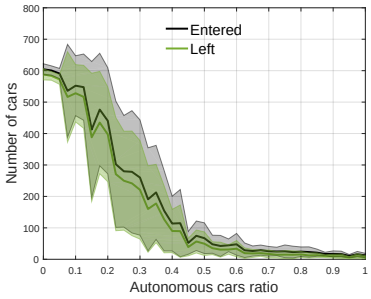


(b) Throughput

Figure B.3: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

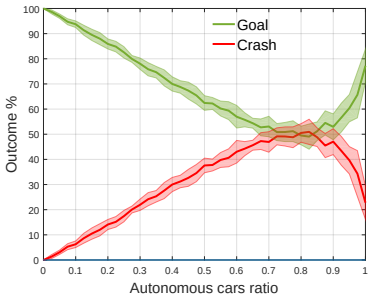


(a) Percentage of each outcome

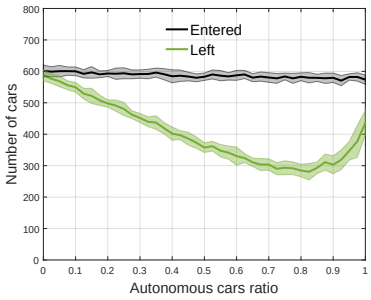


(b) Throughput

Figure B.4: Testing the *simple binary* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

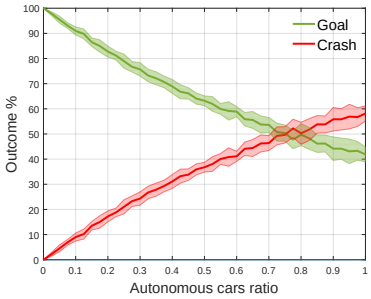


(a) Percentage of each outcome

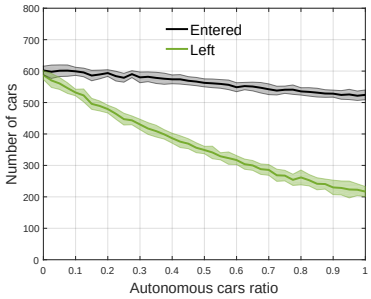


(b) Throughput

Figure B.5: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5 in a multi-agent simulation where all agents use the same network.

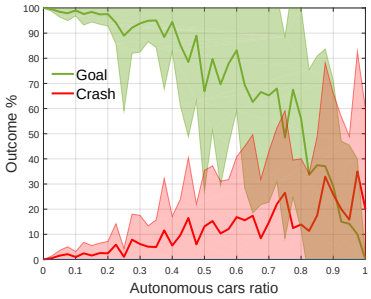


(a) Percentage of each outcome

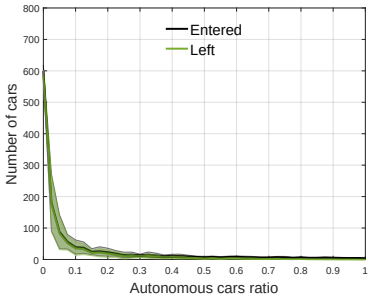


(b) Throughput

Figure B.6: Testing the *simple binary* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5 in a multi-agent simulation where all agents use the same network.

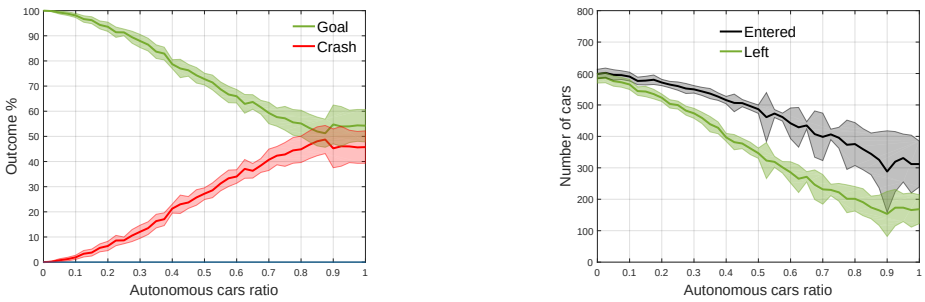


(a) Percentage of each outcome



(b) Throughput

Figure B.7: Testing the *simple binary* model trained on Hydra (Theano) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

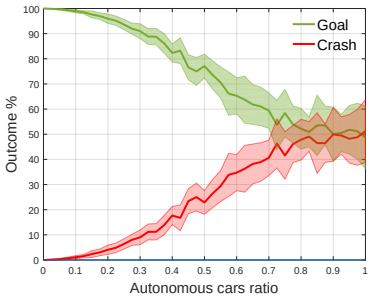


(a) Percentage of each outcome

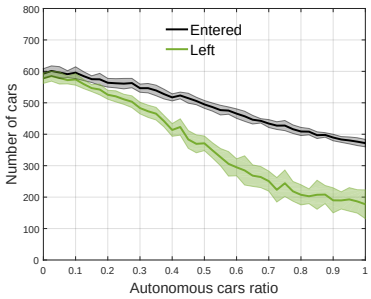
(b) Throughput

Figure B.8: Testing the *simple binary* model trained on the AI cluster (TensorFlow) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

B.2 *Improved Binary* Model

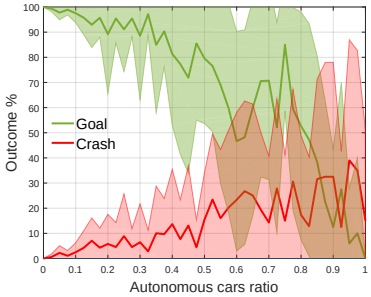


(a) Percentage of each outcome

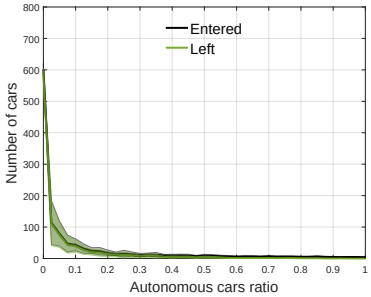


(b) Throughput

Figure B.9: Testing the *improved binary* model trained on Hydra (Theano) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

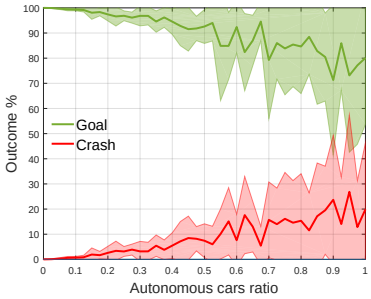


(a) Percentage of each outcome

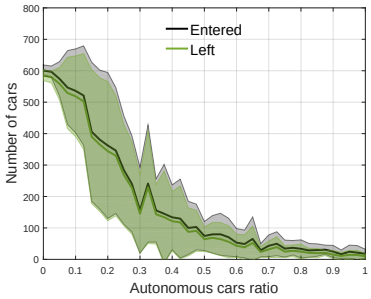


(b) Throughput

Figure B.10: Testing the *improved binary* model trained on the AI cluster (TensorFlow) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

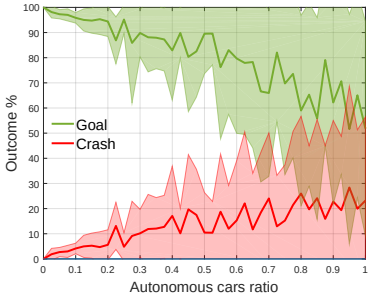


(a) Percentage of each outcome

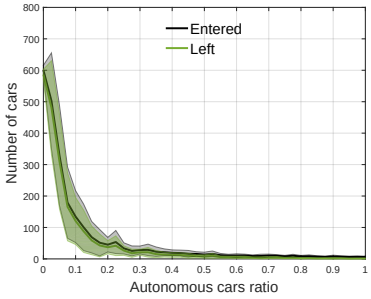


(b) Throughput

Figure B.11: Testing the *improved binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

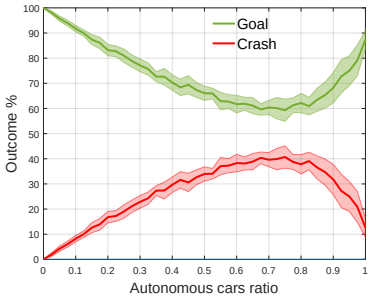


(a) Percentage of each outcome

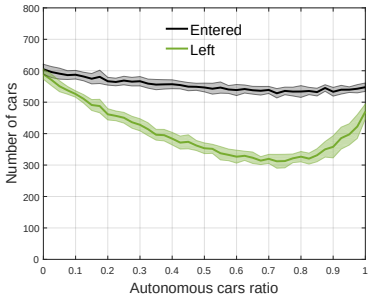


(b) Throughput

Figure B.12: Testing the *improved binary* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

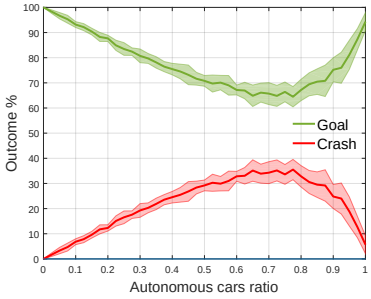


(a) Percentage of each outcome

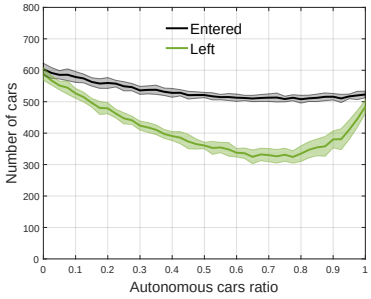


(b) Throughput

Figure B.13: Testing the *improved binary* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5 in a multi-agent simulation where all agents use the same network.

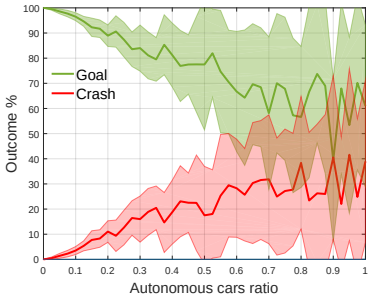


(a) Percentage of each outcome

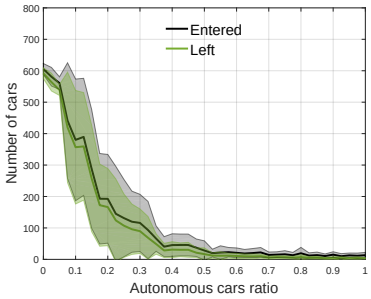


(b) Throughput

Figure B.14: Testing the *improved binary* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5 in a multi-agent simulation where all agents use the same network.

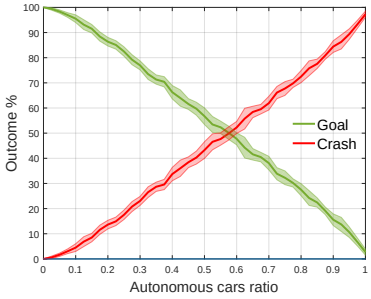


(a) Percentage of each outcome

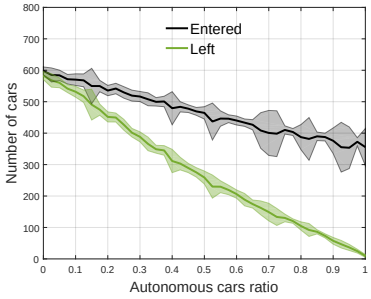


(b) Throughput

Figure B.15: Testing the *improved binary* model trained on Hydra (Theano) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.



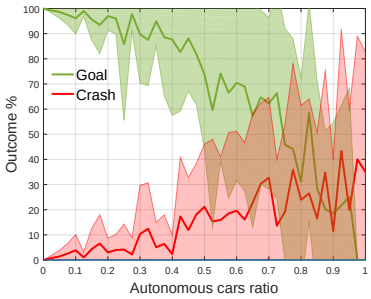
(a) Percentage of each outcome



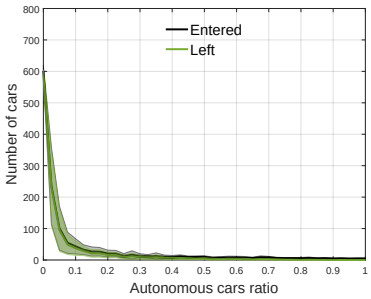
(b) Throughput

Figure B.16: Testing the *improved binary* model trained on the AI cluster (TensorFlow) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

B.3 *Simple Speed* Model

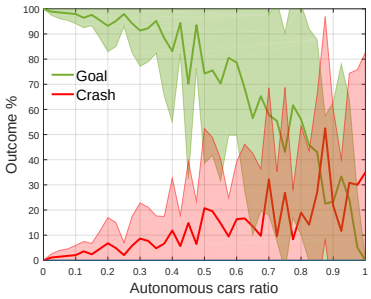


(a) Percentage of each outcome

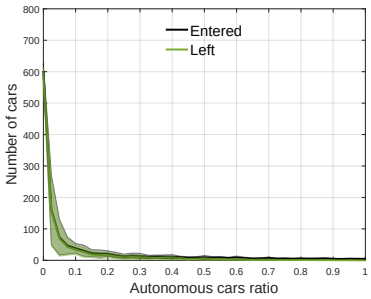


(b) Throughput

Figure B.17: Testing the *simple speed* model trained on Hydra (Theano) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

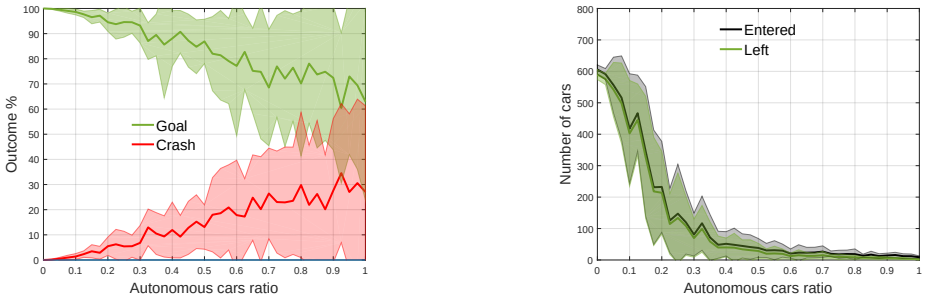


(a) Percentage of each outcome



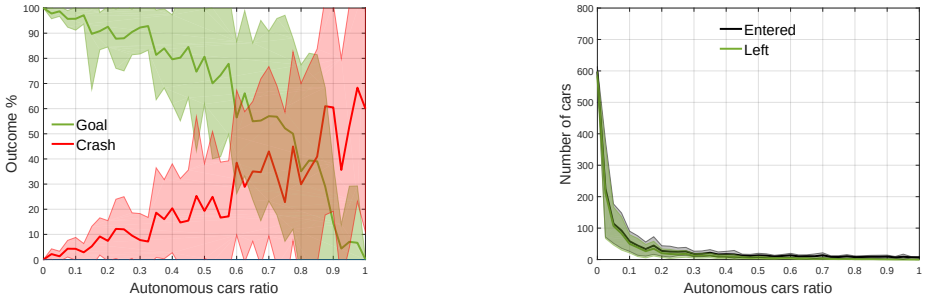
(b) Throughput

Figure B.18: Testing the *simple speed* model trained on the AI cluster (TensorFlow) with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.



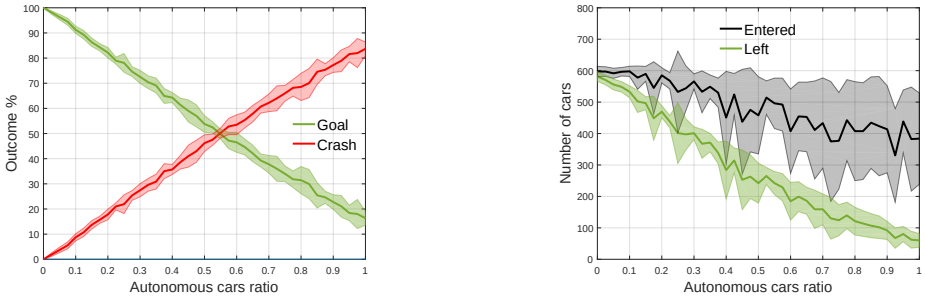
(a) Percentage of each outcome (b) Throughput

Figure B.19: Testing the *simple speed* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.



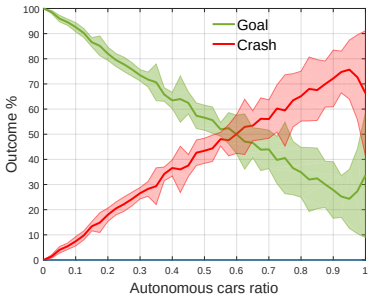
(a) Percentage of each outcome (b) Throughput

Figure B.20: Testing the *simple speed* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

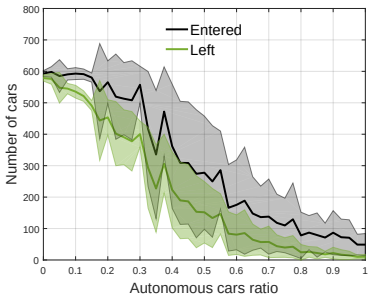


(a) Percentage of each outcome (b) Throughput

Figure B.21: Testing the *simple speed* model trained on Hydra (Theano) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5 in a multi-agent simulation where all agents use the same network.

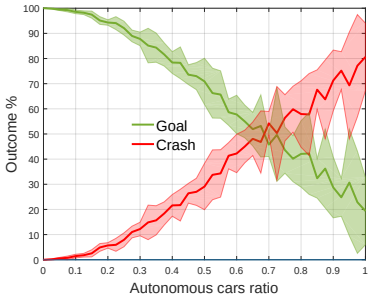


(a) Percentage of each outcome

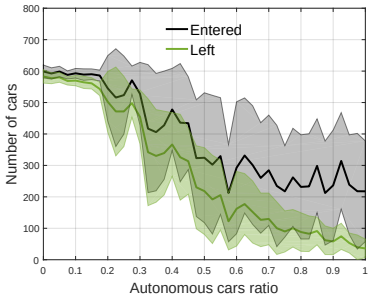


(b) Throughput

Figure B.22: Testing the *simple speed* model trained on the AI cluster (TensorFlow) with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5 in a multi-agent simulation where all agents use the same network.

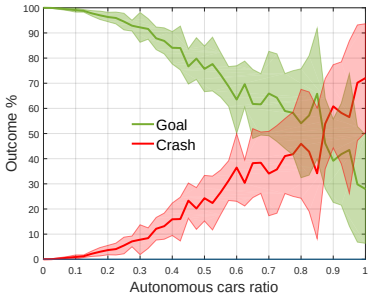


(a) Percentage of each outcome

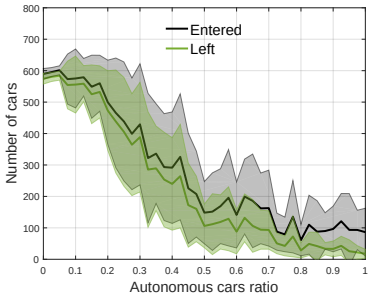


(b) Throughput

Figure B.23: Testing the *simple speed* model trained on Hydra (Theano) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.



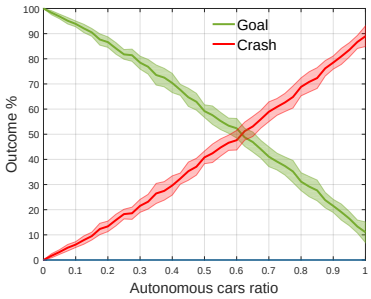
(a) Percentage of each outcome



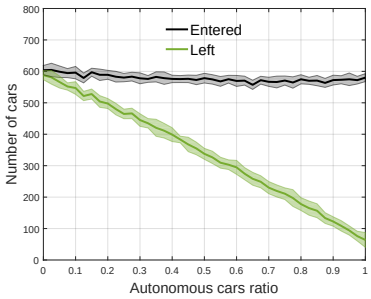
(b) Throughput

Figure B.24: Testing the *simple speed* model trained on the AI cluster (TensorFlow) with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

B.4 *Cars Speed* Model

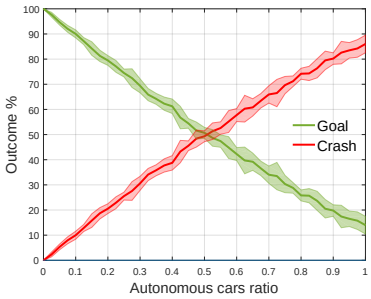


(a) Percentage of each outcome

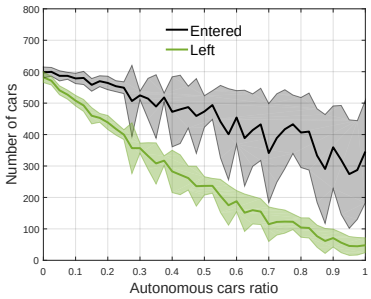


(b) Throughput

Figure B.25: Testing the *cars speed* model trained with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

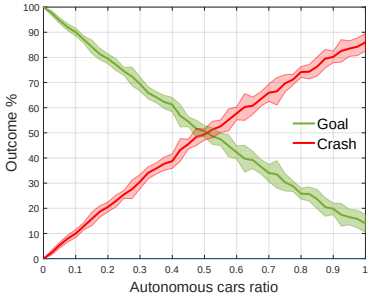


(a) Percentage of each outcome

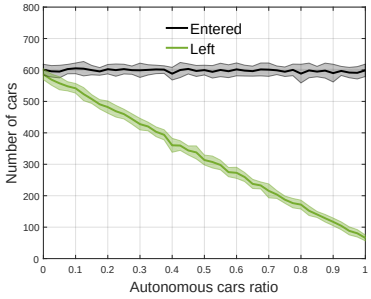


(b) Throughput

Figure B.26: Testing the *cars speed* model trained with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

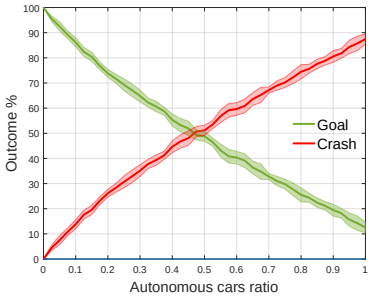


(a) Percentage of each outcome

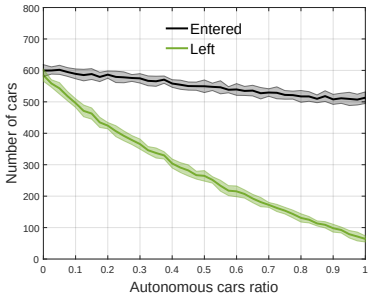


(b) Throughput

Figure B.27: Testing the *cars speed* model trained with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5 in a multi-agent simulation where all agents use the same network.



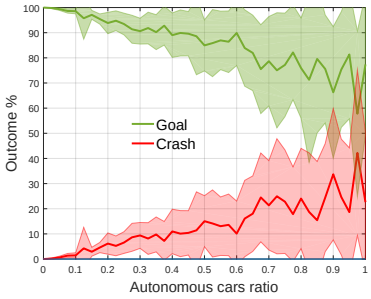
(a) Percentage of each outcome



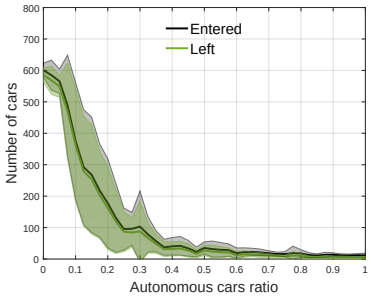
(b) Throughput

Figure B.28: Testing the *cars speed* model trained with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

B.5 *Cars t* Model

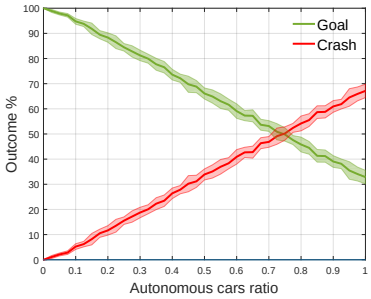


(a) Percentage of each outcome

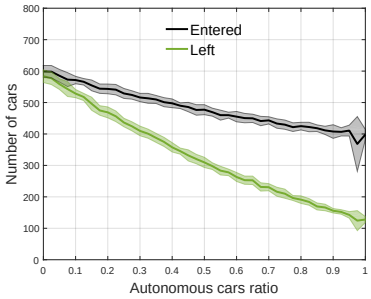


(b) Throughput

Figure B.29: Testing the *cars t* model trained with hidden layers of 60 (*tanh* activation function) and 30 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

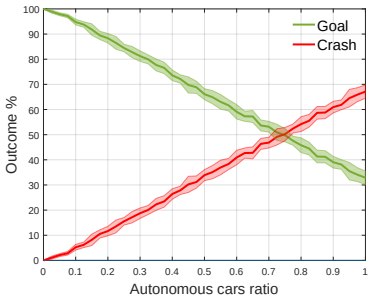


(a) Percentage of each outcome

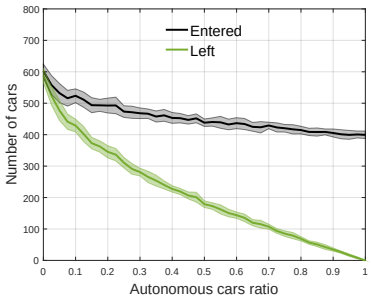


(b) Throughput

Figure B.30: Testing the *cars t* model trained with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

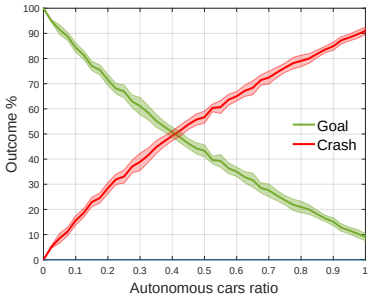


(a) Percentage of each outcome

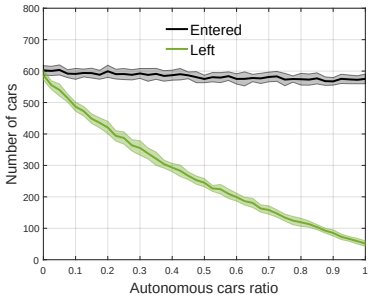


(b) Throughput

Figure B.31: Testing the *cars t* model trained with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5 in a multi-agent simulation where all agents use the same network.



(a) Percentage of each outcome



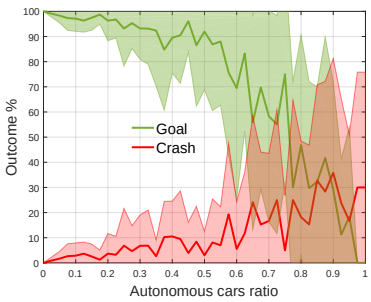
(b) Throughput

Figure B.32: Testing the *cars t* model trained with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout in a multi-agent simulation where all agents use the same network.

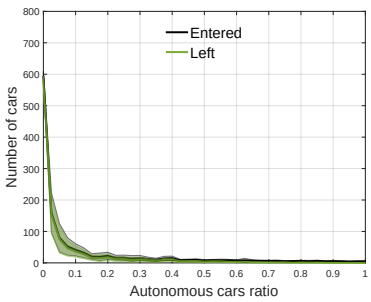
Appendix C

Multi-Agent Testing Results

C.1 *Simple Binary* Model

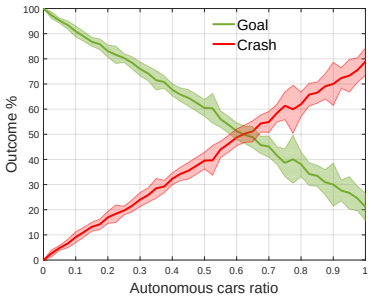


(a) Percentage of each outcome

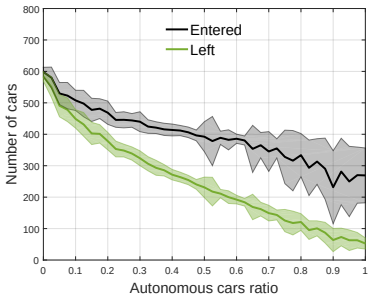


(b) Throughput

Figure C.1: Testing the *simple binary* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout.

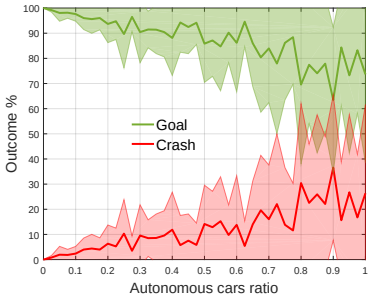


(a) Percentage of each outcome

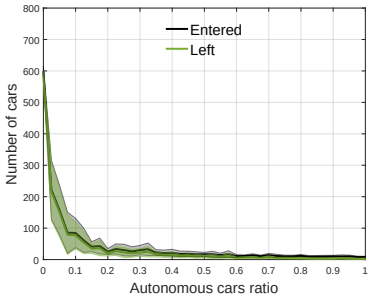


(b) Throughput

Figure C.2: Testing the *simple binary* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5.

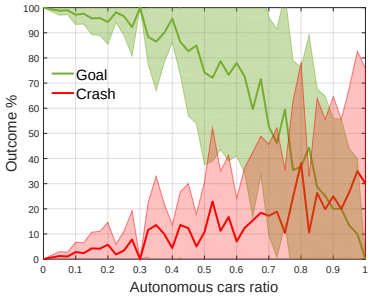


(a) Percentage of each outcome

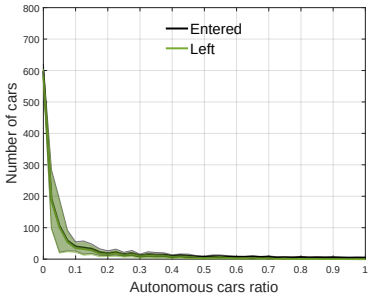


(b) Throughput

Figure C.3: Testing the *simple binary* model with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout.

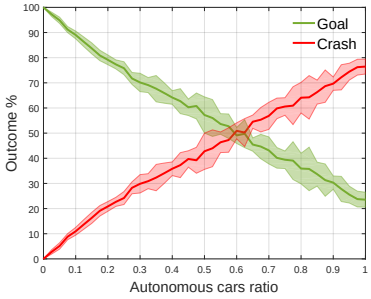


(a) Percentage of each outcome

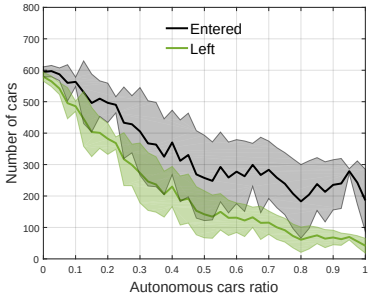


(b) Throughput

Figure C.4: Testing the multi-agent extended *simple binary* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout.

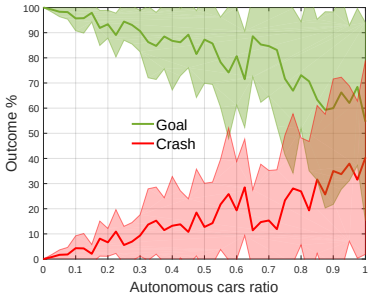


(a) Percentage of each outcome

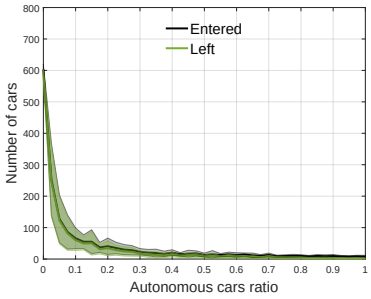


(b) Throughput

Figure C.5: Testing the multi-agent extended *simple binary* model with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5.

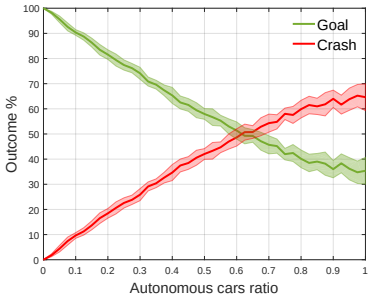


(a) Percentage of each outcome

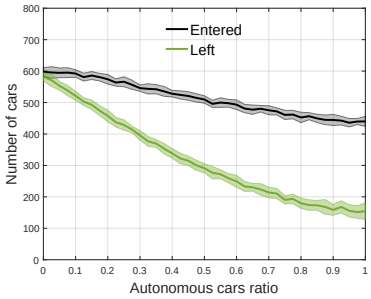


(b) Throughput

Figure C.6: Testing the multi-agent extended *simple binary* model with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout.

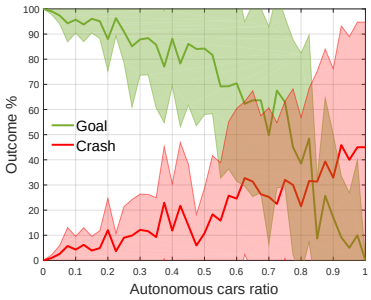


(a) Percentage of each outcome

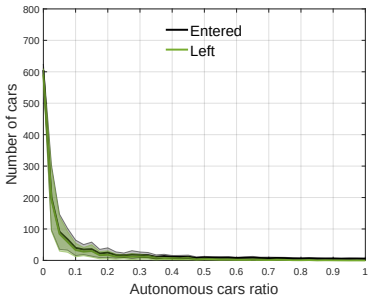


(b) Throughput

Figure C.7: Testing the model using a fully trained *simple binary* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5.



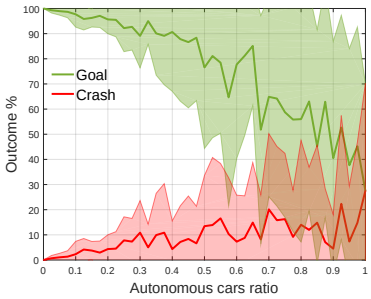
(a) Percentage of each outcome



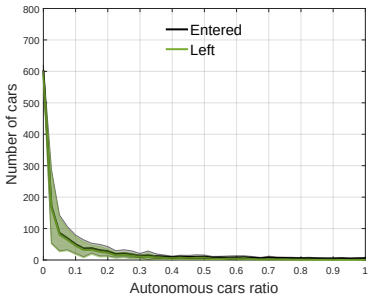
(b) Throughput

Figure C.8: Testing the model using a fully trained *simple binary* model network with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout.

C.2 *Improved Binary* Model

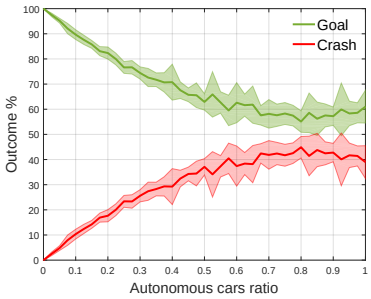


(a) Percentage of each outcome

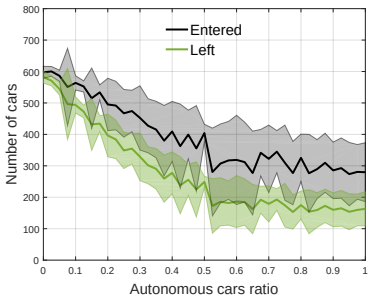


(b) Throughput

Figure C.9: Testing the *improved binary* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout.

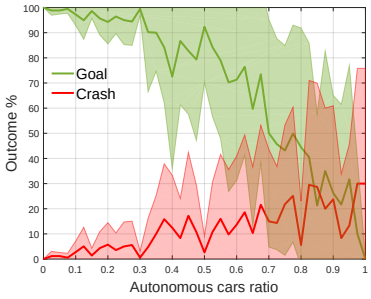


(a) Percentage of each outcome

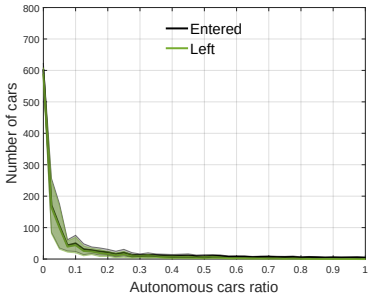


(b) Throughput

Figure C.10: Testing the *improved binary* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5.

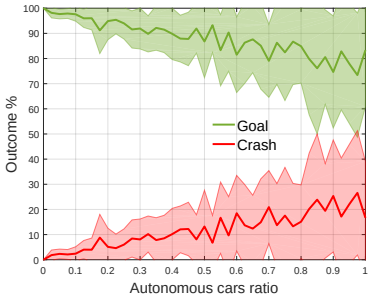


(a) Percentage of each outcome

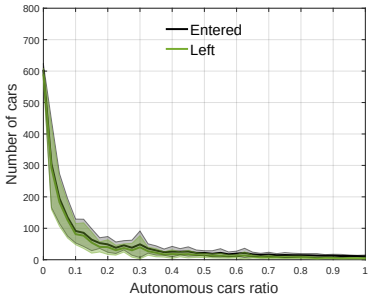


(b) Throughput

Figure C.11: Testing the *improved binary* model network with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout.

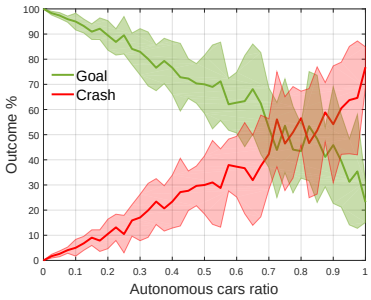


(a) Percentage of each outcome

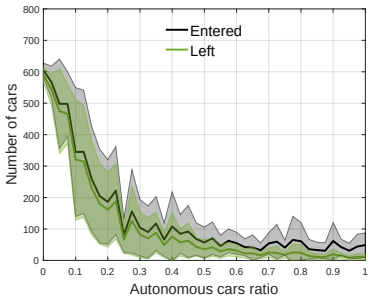


(b) Throughput

Figure C.12: Testing the multi-agent extended *improved binary* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout.

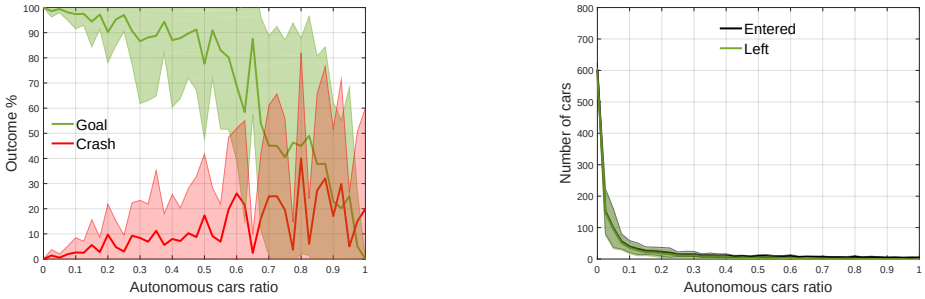


(a) Percentage of each outcome



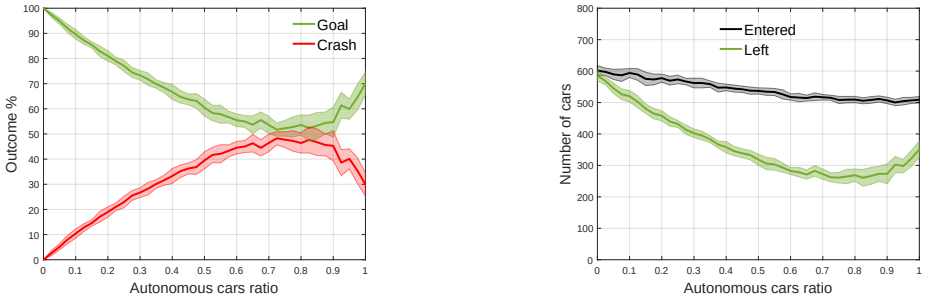
(b) Throughput

Figure C.13: Testing the multi-agent extended *improved binary* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5.



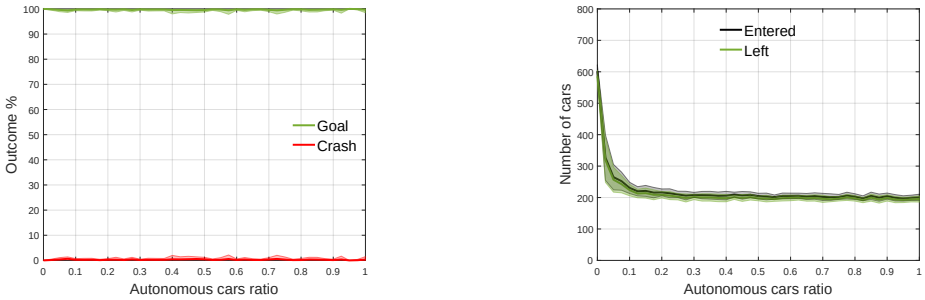
(a) Percentage of each outcome (b) Throughput

Figure C.14: Testing the multi-agent extended *improved binary* model network with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout.



(a) Percentage of each outcome (b) Throughput

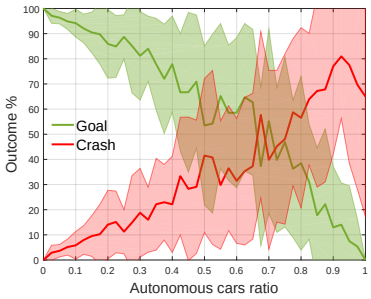
Figure C.15: Testing the model using a fully trained *improved binary* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5.



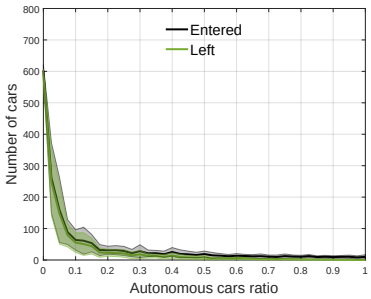
(a) Percentage of each outcome (b) Throughput

Figure C.16: Testing the model using a fully trained *improved binary* model network with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout.

C.3 *Simple Speed* Model

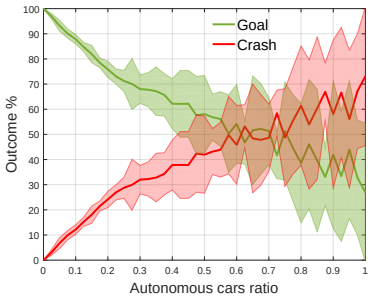


(a) Percentage of each outcome

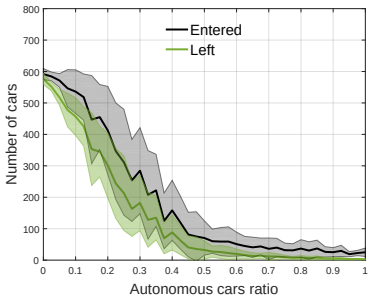


(b) Throughput

Figure C.17: Testing the *simple speed* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout.

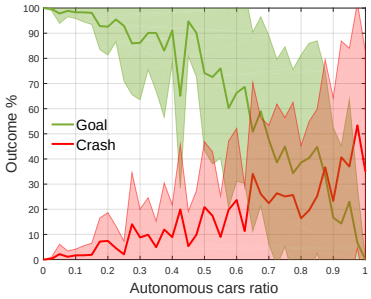


(a) Percentage of each outcome

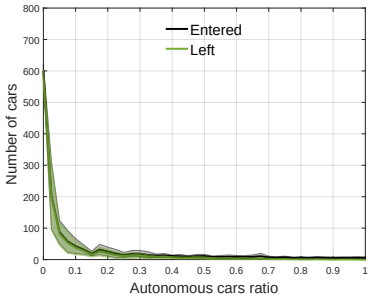


(b) Throughput

Figure C.18: Testing the *simple speed* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5.

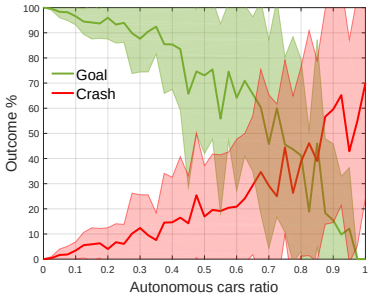


(a) Percentage of each outcome

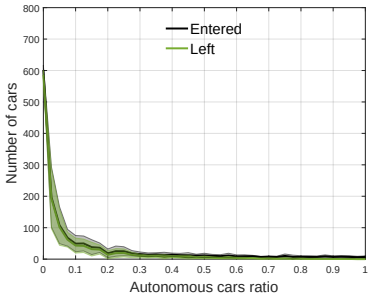


(b) Throughput

Figure C.19: Testing the *simple speed* model network with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout.

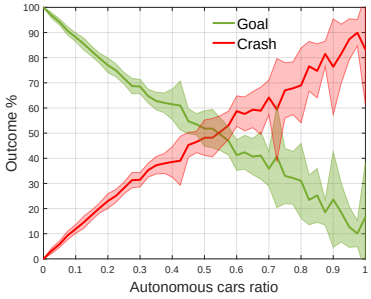


(a) Percentage of each outcome

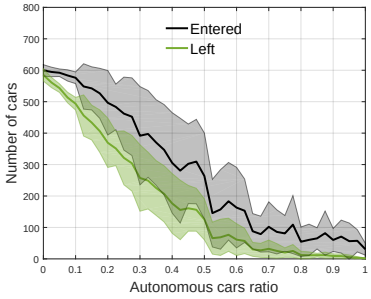


(b) Throughput

Figure C.20: Testing the multi-agent extended *simple speed* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons without dropout.

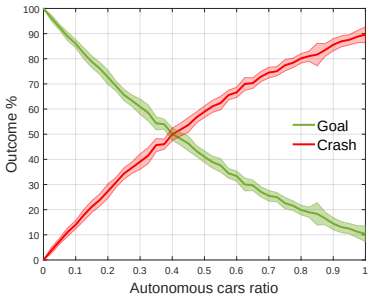


(a) Percentage of each outcome

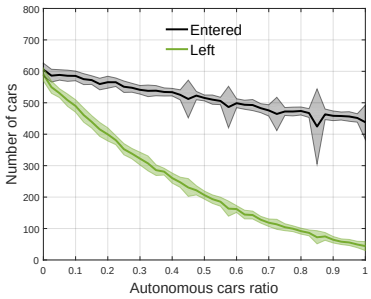


(b) Throughput

Figure C.21: Testing the multi-agent extended *simple speed* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5.

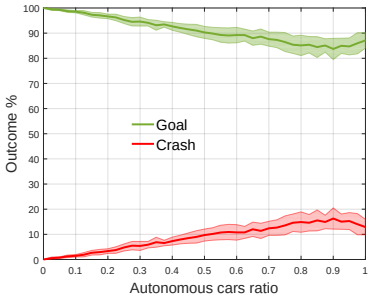


(a) Percentage of each outcome

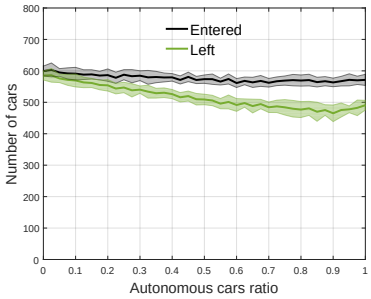


(b) Throughput

Figure C.22: Testing the model using a fully trained *simple speed* model network with hidden layers of 150 (*tanh* activation function) and 75 (*linear* activation function) neurons with a dropout rate of 0.5.



(a) Percentage of each outcome



(b) Throughput

Figure C.23: Testing the model using a fully trained *simple speed* model network with hidden layers of 300 (*tanh* activation function) and 150 (*linear* activation function) neurons without dropout.

Bibliography

Aizerman, M. A., Braverman, E. A., and Rozonoer, L. (1964). Theoretical foundations of the potential function method in pattern recognition learning. In *Automation and Remote Control*, volume 25, pages 821–837.

Arkin, R. (1998). *Behavior-based Robotics*. Bradford book. MIT Press.

Benvenuto, N. and Piazza, F. (1992). On the complex backpropagation algorithm. *IEEE Trans. Signal Processing*, 40(4):967–969.

Busoniu, L., Babuska, R., and Schutter, B. D. (2008). A comprehensive survey of multiagent reinforcement learning. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 38(2):156–172.

Deng, L. and Yu, D. (2014). Deep learning: Methods and applications. Technical report.

Egorov, M. (2016). Multi-agent deep reinforcement learning. Available at `http://cs231n.stanford.edu/reports/2016/pdfs/122_Report.pdf/`.

Foerster, J. N., Assael, Y. M., de Freitas, N., and Whiteson, S. (2016). Learning to communicate with deep multi-agent reinforcement learning. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *NIPS*, pages 2137–2145.

Genders, W. and Razavi, S. (2016). Using a deep reinforcement learning agent for traffic signal control. *CoRR*, abs/1611.01142.

Gupta, J. K., Egorov, M., and Kochenderfer, M. J. (2017). Cooperative multi-agent control using deep reinforcement learning. In *Adaptive Learning Agents Workshop*.

Hoogendoorn, S. P. and Bovy, P. H. L. (2001). State-of-the-art of vehicular traffic flow modelling. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 215(4):283–303.

Hubel, D. H. and Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology (London)*, 195:215–243.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N. C., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., and Hadsell, R. (2016). Overcoming catastrophic forgetting in neural networks. *CoRR*, abs/1612.00796.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Bartlett, P. L., Pereira, F. C. N., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *NIPS*, pages 1106–1114.

Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, 8(3):293–321.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. pages 322–328.

Mccloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. *The Psychology of Learning and Motivation*, 24:104–169.

McCulloch, W. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147.

Melo, F. S., Meyn, S. P., and Ribeiro, M. I. (2008). An analysis of reinforcement learning with function approximation. In Cohen, W. W., McCallum, A., and Roweis, S. T., editors, *ICML*, volume 307 of *ACM International Conference Proceeding Series*, pages 664–671. ACM.

Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT Press, Cambridge, MA.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Nielsen, M. (2013). Neural networks and deep learning. Available at `http://neuralnetworksanddeeplearning.com/` (2013/11/25).

Panait, L. and Luke, S. (2005). Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition.

Ratcliff, R. (1990). Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions. *Psychological Review*, 97:285–308.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408.

Rumelhart, D., Hinton, G., and Williams, R. (1986). *Learning Internal Representations by Error Propagation*.

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A modern approach*. Prentice-Hall.

Shapley, L. S. (1953). Stochastic games. *PNAS*, 39(10):1095–1100.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

Sutton, R. S., Maei, H. R., and Szepesvári, C. (2008). A convergent $O(n)$ temporal-difference algorithm for off-policy learning with linear function approximation. In Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L., editors, *NIPS*, pages 1609–1616. Curran Associates, Inc.

Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., Aru, J., and Vicente, R. (2015). Multiagent cooperation and competition with deep reinforcement learning. *CoRR*, abs/1511.08779.

Walkins, C. and Dayan, P. (1992). Q-learning. *Machine learning*, 8:279–292.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. PhD thesis, University of Cambridge England.

Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University.

Wiering, M. and van Otterlo, M., editors (2012). *Reinforcement Learning: State-of-the-Art*. Springer.

Wooldridge, M. (2002). *An Introduction to Multiagent Systems*. John Wiley & Sons, Ltd, Chichester, England.

Wooldridge, M. and Jennings, N. (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 2(10).