Self-driving Car with Reinforcement Learning

Machine Learning project 2017/2018 - Matteo Prata and Antonio Pio Ricciardi

The objective of this project was to build a model capable of training a car to drive autonomously in an unknown environment. We used a reinforcement learning approach called Q-learning to build such a model, we trained it using Tensorflow and we evaluated its performance.

1 Introduction

Automobile invention have influenced the 20th century more than any other invention ever did. Most of us use cars daily, as our primary mode of transportation. There are currently over 800 million cars on the road worldwide. Beside the usefulness of such an invention, there may be critical situations for which driving a car can be risky, expensive and even not convenient. In the United States alone, 42,000 people die annually in nearly 6 million traffic accidents. Traffic jams account for 3.7 billion wasted hours of human time and 2.3 billion wasted gallons of fuel. [1]

Moved by what expressed above and by our intrinsic curiosity on how to tackle the problem of allowing a car to drive autonomously, without crashing to the nearby weak entities, we exploited the knowledge acquired during the Machine Learning course and we decided to focus on solving this problem using Machine Learning algorithms that we learned.

We used reinforcement learning approach exploiting Q-learning in two ways: using a lookup-table and neural networks. We evaluated the performance of our agent, the results indicate that our trained agent is capable of successfully controlling a car to navigate around any simulation environment.

After this section, in the document we will present the project following this outline: *section 2* defines the idea of the problem we want to solve; *sections 3 and 4* present the tools we used to solve the problem: Reinforcement Learning and Tensorflow; *section 5* illustrates how the project has been developed; *section 6* illustrates the experimental analysis; *section 7* ends the article with a recap and by showing what interesting insight we have acquired working on this project.

2 Definition of the problem

The problem is the following: we want to allow a car to drive autonomously and to reach a goal state without crashing to weak entities in the path.

An autonomous car or self-driving car is defined as a vehicle that is capable of sensing the environment and navigating without human input. Generally the most important goal of a self-driving car is to reach a certain goal state without colliding to nearby entities in the environment.

Autonomous driving software generally achieves this tasks by factoring into three main functional areas: **perception**, **planning**, and **control**. Perception addresses the problem of mapping sensor data into internal beliefs and predictions about the environment. Planning addresses the problem of making driving decisions. Control then actuates the steering wheel, throttle, brake, and other vehicle controls. We used exactly this structure for building the software behind our self-driving car. We will discuss of this structure in *section 5*.

In order to set up **the environment** we built a simple 3D scenario composed of a car moving within a path delimited by walls, weak entities in this scenario. We used Unity [2], a game development platform to build 3D scenes. The car occupies a certain **state** in the environment.

The car can perform **actions** in the environment: moving forward, turn left or turn right. The car has three sensors on board, by means of which it can sense the environment and plan for further actions.

The car gets **rewards**. We define *good actions* those which lead to *safe states*, where no collision with weak entities can happen. So when the car performs *good actions* it gets a positive reward, otherwise it gets a negative one.

Ideally we want a model which can predict always the *good action* to perform given any state. So we built it. We chose a Reinforcement Learning approach called Q-learning to build the model, we did it exploiting two solution:

a) a lookup-table, the q-table

b) a neural network

Now we introduce three very important tools we used to build the model and run it: Reinforcement Learning, Q-learning and Tensorflow.

3 Reinforcement Learning

Reinforcement learning is a learning paradigm which aims to teach agents to learn how to complete a certain task by trial and error. More precisely the agent learns the optimal action to take in any state is. Feedback is given in the form of a reward. The reward is defined in terms of the task to be achieved: positive reward is given for successfully achieving the task or for any action that brings the agent closer to solving the task; while negative reward is given for any actions that impede the agent from successfully achieving the task.

Reinforcement learning lies between the extremes of supervised learning, where the policy is taught by an expert, and unsupervised learning where no feedback is given and the task is to find structure in the data. [3]

This paradigm is the model of how humans and animals learn. We receive rewards in the real life: from our parents, grades in school, salary at work - they are all examples of rewards. If the reward is positive, it gives us motivation to continue doing it, if the reward is negative it pushes us to stop. The same works for the agents in the reinforcement learning model, they act on the environment guided by rewards.

We can reason about reinforcement learning model by thinking of a Markov decision process.

3.1. Markov Decision Process

A Markov Decision Process (MDP) model is defined in the following terms. There is an *agent* which is an autonomous entity that observes and acts upon an *environment*. The agent is in a certain state within the environment. The actions he performs sometimes result in a *reward*. The possible *actions* transform the environment and lead to a new state, where the agent can perform another action, and so on. The best next action to perform, given a state, is predicted by a policy. The environment in general is stochastic, which means the next state may be random.

The set of states and actions, together with rules for transitioning from one state to another and for getting rewards, make up a Markov decision process. One episode of this process forms a finite sequence of states, actions and rewards.

$$(s_0, a_0, r_1, s_1), (s_1, a_1, r_2, s2), \ldots, (s_{n1}, a_{n1}, r_n, s_n)$$

Where s_i represents the state of the agent, a_i is the action performed by the agent, r_i is the reward obtained performing action a_{i-1}. The episode ends with terminal state s_n.

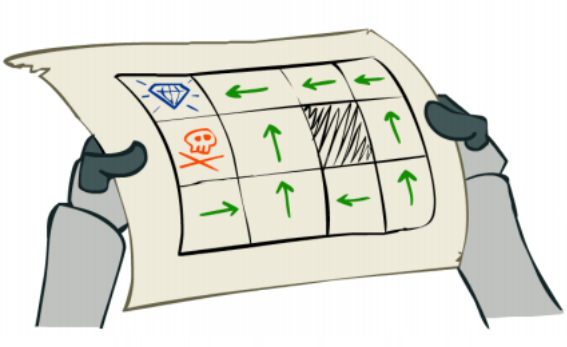

Figure 1. This is the optimal policy that a certain agent should follow in order to reach the treasure and escape death. [4]

A Markov decision process relies on the Markov assumption, that **the probability of the next state s_{i+1} depends only on current state s_i and performed action a_i**, but not on preceding states or actions.

3.2. Discounted Future Reward

To perform well on the long run, we need to take into account the immediate rewards and the future rewards we are going to get. Consider that in one episode one gets a total reward of:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-t} r_n$$

Where γ is the discount factor between 0 and 1. The more into the future the reward is, the less we take it into consideration. We can also write the total reward in terms of itself:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

To balance between immediate and future rewards, one should set discount factor to $\gamma = 0.9$. If the environment is deterministic and the same actions always result in same rewards, then we can set discount factor $\gamma = 1$.

We want the agent to always choose an action that maximizes the discounted future reward.

3.3. Q-Learning

The function $Q(s, a)$ represents the discounted future reward when we perform action a in state s. It is called Q-function, because it represents the quality of certain action in given state. [5]

So evidently what we want to learn is a policy π, that given a state tells me exactly what is the next action that helps me to maximizes the discounted future reward:

$$\pi(s) = \arg\max_a Q(s, a)$$

Figure 1 shows an optimal policy. So, just like with discounted future rewards in previous subsection we can express Q-value of state s and action a in terms of Q-value of next state s:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

The above is called **Bellman equation**. Maximum future reward for this state and action is the immediate reward plus maximum future reward for the next state.

We can approximate the Q-function using the Bellman equation and using an iterative approach. In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns. The gist of Q-learning algorithm is expressed by the following algorithm:

Algorithm 1 Q-learning training

init table $Q[states, actions] = 0$
init environment env
init state s
init learning parameters α, γ
init #epochs $epochs$
while $epochs \geq 0$:
$epochs = epochs - 1$
$a = max_{a'} Q[s, a']$
$\langle r, s' \rangle = env$.**perform_action(**$a$**)**
$Q[s, a] = Q[s, a] + \alpha(r + \gamma max_{a'} Q[s', a'] - Q[s, a])$
$s = s'$

The α vale in the algorithm is a learning rate that controls how much of the difference between previous Q-value and newly proposed Q-value is taken into account. If we perform this update enough times, then the Q-function will converge and represent the true Q-value.

Once the tables is created, for an agent to use it is straightforward, it only needs to look at its current state each time and pick the action that led to the highest Q-value. But what happens when our states are not a finite countable quantity? A table wouldn't be enough, an approximator of the Q-function must be used.

3.4. Neural Network as a Q-function approximator

While it is easy to have a table for a simple grid world, the number of possible states in any modern game or generic environment is nearly infinitely larger. For most interesting problems, tables simply don't work. We instead need some way to take a description of our state, and produce Q-values for actions without a table: that is where neural networks come in. By acting as a function approximator, we can take any number of possible states that can be represented as a vector and learn to map them to Q-values.

Instead of directly updating our table, with a network we will be using backpropagation and a loss function. Our loss function will be sum-of-squares loss, where the difference between the current predicted Q-values, and the target value is computed and the gradients passed through the network.

$$Loss = \sum (\text{Q-target} - \text{Q-predicted})^2$$

We can take two kind of approaches in simulating the Q-table with a neural network. We could represent our Q-function with a neural network, that takes the state and action as input and outputs the corresponding Q-value. Alternatively **we could take only states as input and output the Q-value for each possible action**. Figure 2 shows both approaches.

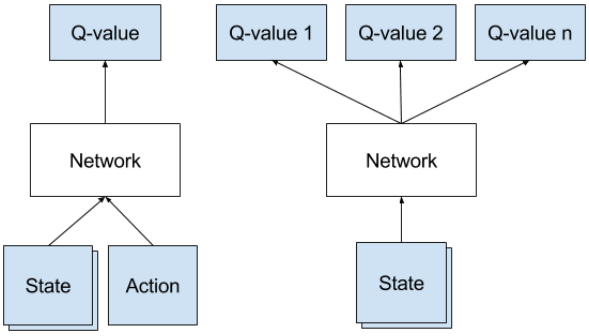

Figure 2. On the left the scheme of a neural network that takes the state and action as input and outputs the corresponding Q-value. On the right the scheme of a neural network that takes the state as input and outputs the Q-value corresponding to each of the possible actions.

3.4.1. Convergence tricks: Experience Replay

During the simulation all the experiences $\langle s, a, r, s' \rangle$ are stored in a replay memory. When training the network, random samples from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. Also experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm.

3.4.2. Convergence tricks: Exploration-Exploitation

When a Q-table or Q-network is initialized randomly, then its predictions are initially random as well. If we pick an action with the highest Q-value, the action will be random and the agent performs exploration. As a Q-function converges, it returns more consistent Q-values and the amount of exploration decreases. So one could say, that Q-learning incorporates the exploration as part of the algorithm. But this exploration is greedy, it settles with the first effective strategy it finds.

A simple and effective fix for the above problem is ϵ-greedy exploration, with probability ϵ choose a random action, otherwise go with the greedy action with the highest Q-value.

$\Diamond$

The pseudo code of an agent training without the use of a table is shown at Algorithm 2 (note: no experience replay is applied). What we do is to train the neural network so to predict the Q-values for each of the actions ($qvalues0$), then we compare them with the ground truth ($target$) and eventually minimize the error between the two by backpropagating it in the network.

As we saw whenever the environment is very complex, the size of the state-action space may become prohibitively large. In order to overcome these problems, Q-learning can be implemented using a neural network as a function approximator for Q-values. Let's analyze one of the most famous machine learning tool for building learning models (like Neural Networks) that we also used for our project, let us talk about TensorFlow.

Algorithm 2 NNQ-learning

setupt Neural Network nn
init environment *env*
init state s
init learning parameters α, γ
init #epochs *epochs*

while $epochs \geq 0$:
$epochs = epochs - 1$
$qvalues0 = nn$.**feedforward_pass**(s)
$a =$ **maximum_value**$(qvalues0)$
with probability ϵ assign to a a random action
$\langle r, s' \rangle = env$.**perform_action(***a***)**

$qvalues1 = nn$.**feedforward_pass**(s')
$target = r + \gamma \max qvalues1$
nn.**train**$((target - qvalues0)^2)$
$s = s'$

4 TensorFlow

TensorFlow is an open source software library for numerical computation using data-flow graphs. It was originally developed by the Google Brain Team within Google's Machine Intelligence research organization for machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well. TensorFlow is cross-platform. It runs on nearly everything: GPUs and CPUs, including mobile and embedded platforms.

The central unit of data in TensorFlow is the **tensor**. A tensor consists of a set of primitive values shaped into a matrix of any number of dimensions. A tensor's rank is its number of dimensions, while its shape is a tuple of integers specifying the array's length along each dimension. Here one example of a rank 3 tensor with shape [2, 1, 3]:

```
[[[1., 2., 3.]], [[7., 8., 9.]]]
```

TensorFlow uses numpy arrays to represent tensor values.
TensorFlow Core programs consists of two discrete sections:

1. Building the computational graph (a `tf.Graph`)
2. Running the computational graph (via `tf.Session`)

A **computational graph** is a series of TensorFlow operations arranged into a graph. The graph is composed of two types of objects.

1. Operations: The nodes of the graph. Operations describe calculations that consume and produce tensors.
2. Tensors: The edges in the graph. These represent the values that will flow through the graph. Most TensorFlow functions return tf.Tensors.

To evaluate tensors we can use tf.Session object. A **session** encapsulates the state of the TensorFlow runtime, and runs TensorFlow operations.
One can populate the computational graph by using the following operations that build tensors:

- `tf.constant`: creates a tensor of fixed shape and values.
- `tf.placeholder`: creates a tensor of fixed shape, it can be fed of arbitrary values during the execution of a session. These are good for representing the input layer of a neural network.
- `tf.Variable`: creates a tensor of fixed shape and whose value can be modified running operations on it. These are good for representing the weights between two layers.

To optimize a model, you first need to define the loss. The `tf.losses` module provides a set of common **loss functions**. For example it's possible to compute the mean square error as follows:

```
loss = tf.losses.mean_squared_error(truth,
prediction)
```

TensorFlow provides optimizers implementing standard **optimization algorithms**. They incrementally change each variable in order to minimize the loss. The simplest optimization algorithm is gradient descent, implemented by `tf.train.GradientDescentOptimizer`. It modifies each variable according to the magnitude of the derivative of loss with respect to that variable. What follows is an application of gradient descent optimization function.

```
optimizer = tf.train.
GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
```

We used TensorFlow to build and train our neural network. We now present the pillars of our project and the TensorFlow implementation.

5 Project Development Details

We now present the development of the project by showing the two versions we worked on. In particular we focused on allowing the car driving autonomously using the two Q-Learning approaches we previously discussed in section 3: **Q-Neural Network implementation** and **Q-Table implementation**. But before introducing such implementations we will discuss about the creation of the 3D environment we used for both simulations.

5.1. 3D Environment Creation

We wanted to develop a car which was able to drive in a realistic environment. In particular we aimed to create a simple 3D world in which the car could move and potentially collide with weak entities such as walls delimiting the path that the car should follow.

We decided to use Unity, a game development platform to build our 3D world. The world is composed of mainly 4 kinds of entities:

1. **Plane**: it is the surface over which the car and the walls stand on.
2. **Car**: it is the agent of our system. It can perform actions on the environment and it occupies a state in it.
3. **Walls**: they are the delimiters of the path that the car must follow in order to reach the goal. When the car hits the wall, the simulation is restarted.

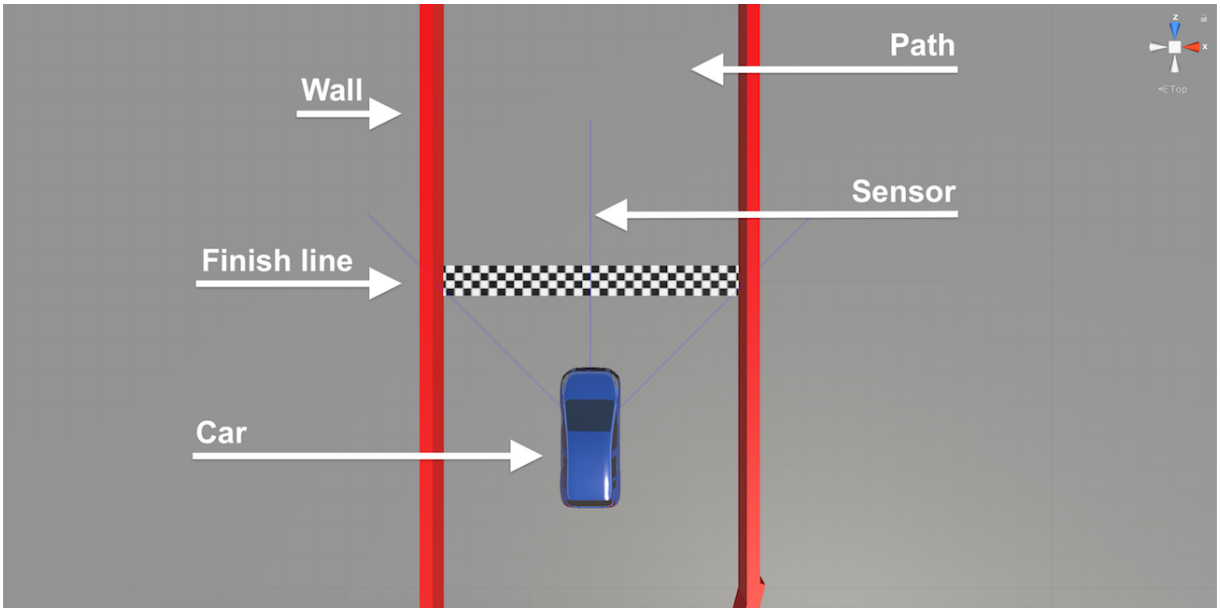

Figure 3. It shows the components of our 3D world.

4. **Sensors**: they are fixed objects on the car itself. They are able to estimate the distance of the car from the objects around it. Distance 10 represents the maximum distance, distance 0 is the minimum distance which corresponds to the collision of the car. Sensors are responsible to defining the state of the car.

Let us define the link between the environment that we just described and the reinforcement learning model.

- the **agent** of our system is the *car* object.
- the **state** of the agent is modeled by the *sensors*. The distances of the car from the nearby objects, computed by the sensors, represent the state of the car. The car has three sensors on board, one 45 degrees oriented on the left, one frontal, one 45 degrees oriented on the right. So the state is represented by the tuple containing the three distances.

  For instance if the tuple `(10, 10, 10)` represents the state in which the car isn't colliding with any wall; the tuple `(10, 5, 0)` represents the state in which the car is not colliding with any wall on the left, it is getting closer to a frontal wall, it is touching the right wall.
- the **actions** that an agent can performs on a given state are free, from all the states all of the available actions can be performed. Actions don't modify the environment. The allowed actions are: *move right*, *move left* and *move forward*, from all the states.
- the **goal** of the agent is to drive within the path trying the maximize the total reward. What affects the total reward is the correspondent immediate reward of an actions performed give a state.
- the **reward** is given as a function of the position of the car in the path, so if the car is critically close to the wall it gets a negative reward; it gets a positive reward otherwise. The actual values of each of the rewards is a hyperparameter for us and all the variants of these values are reported in section 6.
- one **epoch** ends when the agent reaches a *reset* state. The reset state is when the car collides or when the car reaches the finish line on the path.

Figure 3 shows the components of our 3D world. The movement of the agent in the environment, the logic behind the movement, the physics of the agent, the collision system, and so on, were implemented in `C#` programming language. The car moves by `C#` commands and through `C#` it is possible to retrieve the state of the car in the environment (distances from the sensors and collisions with walls).

5.2. Version 1: Q-Neural Network

As we anticipated in section 3.4, solving the problem of allowing a car to drive autonomously using Q-Learning in a simple grid world is relatively easy. It becomes problematic when the number of possible states is nearly infinitely larger, because it would be required to keep a huge table.

For most interesting problems, tables simply don't work. We instead need some way to take a description of our state, and produce Q-values for actions without a table: that is where neural networks come in. By acting as a function approximator, we can take any number of possible states that can be represented as a vector and learn to map them to Q-values.

Our first version of the solution was based on the prediction by a Q-Neural Network (a Neural Network that can predict Q-values given a state as input) of the proper action that the car should perform when in a certain state. In our case the input vector of the neural network contains the values of the three distances recorded by the sensors.

We implemented algorithm 2 reported in section 3. Before showing the code, it follows a description of the Q-Neural Network we built for solving this problem.

5.2.1. Neural Network Description

Figure 4 shows the neural network we implemented and trained in TensorFlow.

The neural network is composed of three **layers**: one input layer composed of three neurons, one output layer composed of three neurons, one hidden layer composed of 5 to 15 neurons.

The **weights** over the edges that connect input layer with the hidden layer and those that connect the hidden layer with the output layer are initialized randomly.

The **activation function** between the input layer and the hidden layer is the `ReLu` function; activation function between the hidden layer and the output layer is the `linear` function. In particular `ReLu` function or rectified linear unit is a function that encourages sparsity and reduces the likelihood of vanishing gradient. The constant gradient of `ReLu` results in faster learning. The function is this:

$$R(x) = \max\{0, x\}$$

The input of this function is the result of the weighted sum between the inputs of the neurons on the preceding layer and the weights over the edges between the preceding neurons and this neuron. The wighted sum is defined as follows:

$$net_j = \sum_{i=0}^{n} I_i \cdot w_{ij}$$

Where net_j is the input of the neuron j and w_{ij} is the weight over the edge ij. This input net_j is what the activation functions get as input. The linear activation function does nothing but returning net_j value.

In order to update the weights using backpropagation we used as a **loss function** the sum of the squared errors.

$$Loss = \sum(\text{target} - \text{predicted})^2$$

As an **optimizer** we experimentally decided to adopt *Adam optimization algorithm*. It is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing. Adam is different to classical stochastic gradient descent. Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. Adam instead works as follows: a learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. In the original paper, Adam was demonstrated empirically to show that convergence meets the expectations of the theoretical analysis. [6]

5.2.2. Unity and TensorFlow Client-Server Communication

Something remarkable was the communication we used between the code written in `C#` and the code written in Python. In fact the neural network was coded using TensorFlow library, which is developed in Python; on the other hand Unity interprets `C#` only.

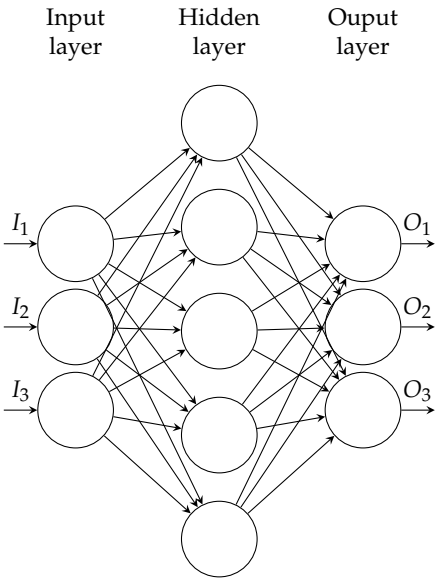

Figure 4. The Q-Neural Network we used for our project.

After several experiments, the latter decision we made was to create a Socket communication in which `C#` was the **client** sending informations to Python and Python was the **server** elaborating the informations and sending them back to client.

That structure came out to be the most efficient. Another solution we tried was to allow the communication via standard input/output, but that was a bad idea since the overhead for all the I/O was dominating the computation and thus the car was physically moving slowly.

5.2.3. Q-Neural Network Implementation in TensorFlow

We now describe and discuss the implementation of the code that allowed us to build the Q-Neural Network using TensorFlow written in Python.

Part 1 is shown in figure 5.

This part of the code is about **importing the libraries** that are necessary in the further code. We used `numpy` for mathematical operations in python, we used `tensorflow` to build the Q-Neural Network, we used `pickle` to serialize the data that characterize a Neural Network in order to the reuse those characteristics rebuilding in a different moment. In particular we used serialization to save the neural network weights after the car reached the finish line, in order to reuse the same neural network configuration in the future, we saved our good model.

In this part we also create the `ReinforcementQNN` class that is our neural network. We initialize some constants that represent the number of nodes on each of the layers. And then we prepare for building the TensorFlow computational graph (discussed in section 4).

Part 2 It is shown in figure 6.

This is the core of our implementation. This part is where most of the hyperparameters are, those we modified most of the times for tuning our model.

This part is not immediately interpreted, as TensorFlow works, it is done in a lazy way. After a session is setup,

we want to ask to the session to run the computation on certain elements of the computational graph.

So we start **building the computational graph** defining the tensor representing the input layer. It is a placeholder since it will be filled at running time by the session.

We then build the tensor representing the weights connecting `input_neurons` to `hidden1_neurons` neurons, in a complete way, this matrix is filled of random values between 1 and -1. Notice that this tensor is a variable, meaning that its values will be modified during the backpropagation phase.

We then define the activation function for the hidden layer, which is a `ReLu`, it takes the result of the matrix multiplication between the tensor of weights `weights_input` and the tensor of input values `input_layer`.

After that we define the tensor of weights between the hidden and the output layer, similarly as above. The output layer is a tensor of `output_neurons` neurons that predict, given a state as input to the Q-Neural Network, the right action to make based on their intensity. These neurons compute their output based on a linear activation function.

Eventually we **define the loss function**, which is the sum of the squared difference between the tensor of predicted Q-values and the tensor of target Q-values. The Adam Optimizer is the one we chose to minimize the loss.

Part 3 is shown in figure 7.

This part of code is responsible for setting up some **useful parameters** for the subsequent computation. In particular we have the discount factor used in Bellman equation, and the epsilon value representing the probability with which we take a random action. Then we **start the session** and we initialize all the variables of the computational graph, we prepare the computational graph for running.

Part 4 is shown in figure 8.

The computational graph is ready to run operations on tensors. From this part we define some functions that are called by Python server, that will be discussed later.

Python server calls `produce_action(state)` which given a state sent by the client `C#`, demands to the Q-Neural Network to produce the best action to perform. This function may also suggest a random action for exploration purposes.

More specifically we ask to the session to perform a **feed-forward pass** to the neural network by asking the interpretation of `prediction_qvalues`, after feeding the input layer with the current `state`. The prediction is the tensor `predicted_qvalues` containing Q-values for each of the possible actions. The highest Q-value represents the best action to perform. Note that if with probability `e` the systems choses to perform the random action, the random action is performed. This function returns the best action or the random action, to the client `C#` which will be responsible of performing that action in the environment.

Part 5 is shown in figure 9.

In this part we **train the Q-Neural Network** to produce Q-values that always lead to the best action possible. Python server calls `train(next_state, reward, reset, action, epoch)` which from a new state; a reward; the information wether `next_state` is a rest state or not; the previous action; the number of epochs, it generates the **ground truth**: the tensor representing the target Q-values from the previous state.

More specifically we ask to the session to perform a feed-forward pass to the neural network by asking the interpretation of `prediction_qvalues`, after feeding the input layer with the `new_state`. The prediction is the tensor `predicted_next_qvalues` containing Q-values for each of the possible actions. The highest Q-value represents the best action to perform and also a parameter of the **Bellman equation**. Multiplying such value with the discount factor and summing the reward we get the new Q-value for `action` from the previous state.

We push the evaluation of the `loss` function by running `update_model` operation of the computational graph. This invocation produces an update to all the variables of the graph representing the weights of the Q-Neural Network, according the gradient.

We eventually update the epsilon value `e` responsible for determining the selection of the random action, by decreasing it in function of the number of epochs.

Part 6 is shown in figure 10.

This last part is composed of two functions: one the `save_weights(weights_list, file_name)` that from a tuple containing two tensors of weights it serializes them in a file, for further usage; and the latter `close_session()` which is responsible for closing the previously opened session. When this function is called, no more evaluations over the computational graph may be done.

5.2.4. Client Implementation

As already said the communication between the code written in `C#` and the one written in Python is possible thanks to the client-server paradigm. In particular what the client does is: (1) get the state of the agent in the environment, (2) send it to the server, (3) wait for a proposed action, (4) perform the action in the environment, (5) start again from step 1.

The code of the entire `C#` infrastructure is in the attachments of this project.

5.2.5. Server Implementation

What the Python server does is: (1) wait for the state of the agent from the client, (2) send it to the Q-Neural Network, (3) wait for a proposed action, (4) send it to the client, (5) start again from step 1.

The code of the entire Python infrastructure is in the attachments of this project.

```python
import numpy as np
import tensorflow as tf
import pickle

class ReinforcementQNN:

    def __init__(self):

        # Number of neurons in the layers of the NN
        input_neurons = 3
        output_neurons = 3
        hidden1_neurons = 8

        # Prepares the computational graph
        tf.reset_default_graph()

```

Figure 5. Q-Neural Network implementation *Part 1*.

```python
        #---------------------------------------------NN-Setup-----------------------------------------

        # Set up of the placeholders for the inputs of the input layer
        self.input_layer = tf.placeholder(shape=[None, input_neurons], dtype=tf.float32)
        # Set up of the tensor containing the weights connecting input layer with hidden1 layer
        self.weights_input = tf.Variable(tf.random_uniform([input_neurons,hidden1_neurons], -1, 1))

        # Activation function on the neurons of the hidden layer. Represent the inputs of the hidden layer
        self.activations_hidden1 = tf.nn.relu(tf.matmul(self.input_layer, self.weights_input))
        # Set up of the tensor containing the weights connecting hidden1 layer with output layer
        self.weights_hidden1 = tf.Variable(tf.random_uniform([hidden1_neurons, output_neurons], -1, 1))

        # It is the function computing the ouput tensor of predicted Q-values
        self.prediction_qvalues = tf.matmul(self.activations_hidden1, self.weights_hidden1)

        #-------------------------------------NN-Backpropagation-------------------------------------

        # It is a tensor which represents the ground truth, the target Q-values
        self.target_qvalues = tf.placeholder(shape=[None, output_neurons], dtype=tf.float32)

        # It is a function computing the sum of the squared error between the target Q-values and the predicted ones
        self.loss = tf.reduce_sum(tf.square(self.target_qvalues - self.prediction_qvalues))

        self.trainer = tf.train.AdamOptimizer(learning_rate=0.001)
        # It is a function computing the minimization of the loss
        self.update_model = self.trainer.minimize(self.loss)

        #--------------------------------------------------------------------------------------------

```

Figure 6. Q-Neural Network implementation *Part 2*.

```python

        # Learning parameters
        self.y = .99
        self.e = 0.1

        # Some variables that will be used in some functions
        self.actions = None
        self.state = None
        self.predicted_qvalues = None

        # It is a function computing the inizialization of all the Varibales
        self.init_op = tf.global_variables_initializer()
        # It starts the a new session and the then it executes the inizialization of the Variables
        self.sess = tf.InteractiveSession()
        self.sess.run(self.init_op)

```

Figure 7. Q-Neural Network implementation *Part 3*.

```
61
62      def produce_action(self, state):
63          self.state = state
64
65          # It makes a feedforward pass on the network and predicts the Q-values for each action
66          self.predicted_qvalues = self.sess.run(self.prediction_qvalues, feed_dict={self.input_layer: [self.state]})
67          # It stores the action to perform
68          predicted_action = np.argmax(self.predicted_qvalues)
69
70          # With probability epsilon select a random action
71          if np.random.rand(1) < self.e:
72              rand = np.random.randint(0,3)
73              predicted_action = rand
74
75          return predicted_action
76
```

Figure 8. Q-Neural Network implementation *Part 4*.

```
77
78      def train(self, next_state, reward, reset, action, epoch):
79          # It makes a feedforward pass on the network and predicts the Q-values for each action
80          predicted_next_qvalues = self.sess.run(self.prediction_qvalues, feed_dict={self.input_layer: [next_state]})
81          # It stores the Q-values of the action to perform
82          predicted_max_qvalue = np.max(predicted_next_qvalues)
83          # It stores the new Q-value computed by Bellman equation
84          new_qvalue = reward + self.y * predicted_max_qvalue
85
86          # It stores the new Q-values for next state
87          target = self.predicted_qvalues
88          target[0, action] = new_qvalue
89
90          # Updates the network by backpropagation computing the error between the prediciton and the truth
91          self.sess.run(self.update_model, feed_dict={self.input_layer: [self.state], self.target_qvalues: target})
92
93          # Decreases the probability of picking a random action
94          if reset and self.e > 0:
95              self.e = 1./((epoch/50) + 10)
96
```

Figure 9. Q-Neural Network implementation *Part 5*.

```
97
98      def save_weights(self, weights_list, file_name):
99          with open(file_name, 'wb') as handle:
100             pickle.dump(weights_list, handle)
101
102
103     def close_session(self):
104         print("Closing TensorFlow session.")
105         self.sess.close()
106
```

Figure 10. Q-Neural Network implementation *Part 6*.

5.3. Version 2: Q-Table

Another version we wanted to present is the version of the problem solved using a Q-Table. This problem wasn't really solved with the Q-Table, but we managed to understand a very nice reason why, that we will express in section 6.

Q-Tables are tables stored in memory. In particular if we use Python we can model this data structure as a list of lists (a matrix), or a dictionary (in which the keys are the references of each row and the values are the rows themselves. What is clear is that a Q-Tables cannot be too big, in order to model a good problem with Q-Tables, they need to fit in memory. If they don't, the problem cannot be solved. What generally make Q-Tables huge is the states-space. In other words the states can be so many that we cannot represent them as rows of a matrix. Another issue could be that, if the table is too big, it is hard to update it, it may require too much time.

As we repeatedly said we know how to overcome these limitations of Q-tables, we use Neural Networks as approximators. **We used the Q-Neural Networks to approximate Q-values, but why?** Because our state-space could potentially be huge. As we said our states are the distances from the car to the nearby objects. We set to 10 the maximum distance that a sensor can sense and 0 the minimum, which corresponds to a collision. If we use floats to represent that distance, we fall into an **uncountable state-space**. Such a state space can only be represented with an approximator, but what if we represent the state as a set of inte-

gers going from 0 to 10? That would make the sensing much less precise in a 3D environment, but would definitely make the **states-space more narrow**. If we do the math there are 3 sensors and each can assume values from 0 to 10 at each point in space. So the total number of stets would be $10^3 = 1000$ definitely a small states-space.

We used Q-Table prediction using exactly this configuration. We have 1000 states representing the sensed data of the sensors and 3 possible actions from all of the states. We decided to implement the Q-Table as a dictionary, representing the states as the key and the Q-values as the associated value. The states were encoded as the concatenation of the distances sensed by the sensors.

5.3.1. Q-Table Implementation in Python

Python implementation of the Q-Table approach follows the same interface (Java speaking) as the Q-Neural Network version. We have a class with the two functions `produce_action` and `train` that act exactly on the same way server side. In fact there's again the server communicating with the `C#` client, the paradigm is just the same as above.

What changes is essentially the way in which the best action is predicted. **Now look at the code in figure 11**.

In the initialization of the class we can see that in addition to the initialization of some learning constants, we have the initialization of the Q-Table itself. The call `initialize_states()` fulls a dictionary of key-value pairs. The keys are the states encoded as the concatenation of all the three possible sensed distances. The values are lists of size three that will contain Q-values for each of the possible actions.

The **`produce_action` function**, given a state returns the best action to perform. It does it by checking in the dictionary, the Q-Table, what is the action that corresponds to the maximum Q-value in this state. Note that the list of Q-values associated to a state can be altered by randomness in order to encourage exploration.

The **`train` function**, given a new state; a reward; the information wether `next_state` is a rest state or not; the previous action; the number of epochs; the information wether `next_state` is a goal state or not, it updates the Q-Table for the previous state according to the full Bellman equation:

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

that we explained in section 3.

6 Experimental Analysis

In this section we will discuss the results observed executing the two versions of the algorithm presented above.

6.1. Version 1: Q-Neural Network

Several experiments have been performed for executing this version of the algorithm. Before introducing some of them it is worth it to give the references to a video illustration of the training process of the Q-Neural Network and the execution of good models over different paths.

The video is available at this url: https://youtu.be/ryUEZAMI1DE

6.1.1. Multiple Paths Training and Evaluation

We focused of training the agent over 2 different paths for two very important reasons:

1. to see how the learning process could vary, keeping the whole configuration the same on both paths.
2. to compare the efficiency of each of the generated models, over a path that is different from the one over which the model was trained.

6.1.2. Hyperparameters Tuning

We use the term hyperparameter to denote parameters of the training model that we can modify in order to improve the learning rate of our model. It is known that simple variations of learning hyperparameters may lead to remarking improvements in the time needed for building the model.

The hyperparameters we focused on for tuning are exposed and described below:

- **Number of hidden layers:** as we described in previous sections, the Q-Neural Network that we preferred has 3 layers (1 input, 1 hidden, 1 output). Before coming to this conclusion we tested some configurations. We built two neural networks, one with 3 layers and one with 4; we kept fixed the number of neurons per layer and we eventually tested the performances of both networks.

  What it turned out was that 3 layers were more than enough for our approximator. In fact we could notice that adding hidden layers only increased the time required for convergence. The weights converged to the optimum, unnecessarily more slowly. Notice that we tried also a network without any hidden layer, but this simplistic structure for a network wasn't leading to any convergence at all.

- **Number of neurons in the hidden layer:** together with the previous hyperparameters testing we also started modifying the size of the hidden layers we put in the network itself.

  What it turned out was that having a number of neurons per hidden layer around 5 and 15 was a good choice. What we noticed is that increasing more this number led to a slower convergence of the weights to the optimum. In particular more epochs were needed for a car to complete the path one time.

- **Sensors precision:** sensors built on the car can sense the environment up to 10 points of distance. We could decide to keep this evaluation bland by representing it using integers (very imprecise), or make it more specific using floats (very precise).

  Training the Q-Neural Network using integers representing the input distances, requires a simpler version of the network: one hidden layer with 5 neurons is enough. Training it using floats is much more hard, since there are way more states for which a good action should be predicted: one hidden layer with 8 to 15 neurons can be necessary.

  What it turned out was that increasing the precision of the sensed data (aiming to precise floats) slows down the convergence of the weights to the optimum, with neural networks with a small hidden layer.

```python
import numpy as np

class ReinforcementQTable:

    def __init__(self):
        # Useful parameters and learning parameters
        self.state = None
        self.epochs = 0
        self.lr = .80
        self.y = .95

        # Initializes the Qtable as a dictionary
        self.Qtable = {}
        self.sensors_length = 10
        self.initialize_states()


    # It initializes the Qtable as a dictionary containing
    # the state a the key and the Q-values as the values
    def initialize_states(self):
        for i in range(sensors_length+1):
            for j in range(sensors_length+1):
                for k in range(sensors_length+1):
                    state = str(i) + str(j) + str(k)
                    self.Qtable[state] = [0, 0, 0]


    def produce_action(self, state):
        self.state = state

        # Stores the Q-values of state for each of the actions
        q_values = Qtable[state]
        # Stores the Q-values of the state for each of the actions altered with a random values for exploration
        altered_q_values = [x + np.random.randn(1)[0]*(1./(self.epochs+1)) for x in q_values]
        # Stores the action suggested by the Q-table
        action = np.argmax(altered_q_values)

        return action


    def train(self, next_state, reward, reset, action, epoch, goal):
        # Stores the maximum Q-value of next_state
        max_q_value = np.max(self.Qtable[next_state])

        # Stores the Q-value of state performed action
        old_q_value = self.Qtable[self.state][action]
        # Updates Q-value of state performed action using Bellman equation
        self.Qtable[self.state][action] = old_q_value + self.lr * ((reward + self.y * max_q_value) - old_q_value)

        if reset: self.epochs +=1
        if goal: self.save_q_table(self.Qtable)


    def save_q_table(self, q_table, file_name):
        with open(file_name, 'wb') as handle:
            pickle.dump(q_table, handle)

```

Figure 11. Q-Table implementation.

Using precise sensed data helps a lot the precision of the model, the car moves more smoothly and is fails very less in estimating the right actions to make. So we decided to keep the precision of the sensed data to the second decimal digit. A good neural network that could process such a state-space turned out to be a Q-Neural Network with three layers and with 8 neurons on the hidden layer.

- **Learning rate:** such a value determines how fast weights should change. This value can be dynamic and change as a function of the epoch. In general the training should start from a relatively large learning rate because, in the beginning, random weights are far from optimal, and then the learning rate can decrease during training to allow more fine-grained weight updates. Also, different layers of the network could implement their own learning rate.

  We tried to make it vary as function of the the epoch on our best discovered network, but we didn't notice any improvement in the learning process. So we decided to make it fixed for all of the layers to a reasonable value that we decided to be $learning\_rate = 0.001$.

- **Activation functions:** as we plenty discussed in section 5.2, we tried to use at least three kind of activation functions: `ReLu`, sigmoid function, and linear function.

  It turned out that in order to compute the input of the hidden layer, `ReLu` leads to better results, it leads to a faster convergence. In order to compute the input of the output layer, linear function was the one performing best. In particular `ReLu` wasn't so proper because given negative inputs it always returns 0 which can be a problem in case of all negative weights.

- **Optimization function:** the choice of the optimization function was pretty much crucial. We tried Gradient Descend optimizer and Adam Optimizer. The latter was descried in section 5.2.

  What it turned out was that when we were about to give up certain neural networks infrastructures that seemed interesting, Adam Optimizer revolved things completely, leading to immediate convergence. We decided to use Adam optimizer.

- **Rewards:** the most influencing factor on the outcomes of our execution was the reward function. We made several experiments, in particular we gave punishments whenever the car was about to get close to critical states, those that led it to a crash with the walls; we gave positive rewards when the car was in safe states.

  In particular at the beginning we were used to give a -1 as a reward whenever the sensors recorded state (x, y, z) where x or y or z was less than or equal 2; and we gave a $+0.3$ otherwise. This was to encourage the agent to keep sensitive objects at a distance no less than 2 points.

  Other tests were done and we gave a -0.3 as a reward whenever the sensors recorded state (x, y, z) where x or y or z was less than or equal 2; and we gave a $+1$ otherwise. Giving a higher reward than a higher punishment led to a faster convergence.

6.1.3. Remarkable Findings

We observed several interesting events while training the network. As it is evident, our learning model doesn't take into account the position of the car in the environment, it is totally independent from that. Distances from nearby objects are the only influencing factors for the state. This has **a huge pro: the car knows how to move, independently from the *position* in the environment**. But it also has a cons: the car is somehow blind, so certain state can easily fool it, and during the training phase many bad actions were only taken because of misleading informations coming from the sensors.

Models are a very satisfying concept for us. A trained model could drive safely (without crashing any wall) on any path tested. This practically means that we could use a trained Q-Neural Network on any physical model of a self-driving car, it can follow a path and never collides with nearby objects.

6.2. Version 2: Q-Table

Executing version 2 of the algorithm using a Q-Table, shown in figure 11, didn't lead to a real solution of the problem. Running Python and `C#` infrastructures lead to disappointing results at the beginning. What happened was that even if the Q-Table was updated correctly, the car didn't show any sign of learning.

After some meditation we came out with the idea that the Q-Table approach is too much limited for our environment, even when the state-space is very limited (1000 states). In particular **the problem is that in Q-Learning the same action performed from a particular state cannot lead to different states**. This indeterministic behavior isn't allowed and in fact causes many problems.

In particular what happened is that from a state s performing action a sometimes led to a change of state, and sometimes not. This weird behavior was dependent of the environment itself. When the estimation of the distances by the sensors wasn't so precise (integer distances), to determine a change of the state was necessary a higher quantity of movement by the agent and this was not always the case.

7 Conclusion

We presented a learning approach for teaching an agent to drive autonomously on a path without colliding with nearby weak entities. We used reinforcement learning approach, exploiting Q-learning in two ways: using a lookup-table and neural networks as Q-values approximators. The Q-Neural Network approach led to interesting results, while the Q-Table solution couldn't allow us to solve the problem practically, but it helped us to make interesting suppositions and reasoning.

The results of our experiments indicate that our agent trained with the Q-Neural Network is capable of successfully driving on any simulation environment without colliding with weak entities. These results really motivated us and increased our desire to dig deeper on the topic of self-driving cars using Machine Learning.

Our approach is only one of many used in this field. We focused on monitoring the state of the agent based in it's position relative to other objects. The learned models can be used on *any* possible path under *only* the assumption that we have a way to retrieve the distances of the car from any surrounding object. This weak assumption made our approach very resilient. There's a countless quantity of possible additional features that one could add to this model. For example add weak entities randomly appearing on the path, the agent should learn to avoid them. This leaves this project open for possible further developments.

8 References

[1] Sebastian Thrun. Toward robotic cars. *Commun. ACM*, 53(4):99–106, April 2010.

[2] Unity website. https://unity3d.com.

[3] D. Wettergreen C. Gaskett and A. Zelinsky. Q-learning in continuous state and action spaces.

[4] Markov decision process explaination. https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/markov_decision_process.html.

[5] Website 2. http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/.

[6] Gentle introduction to the adam optimization algorithm for deep learning. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.