ON THE DEVELOPMENT OF AUTONOMOUS AGENTS USING DEEP REINFORCEMENT LEARNING

Clara Barbu[1] and Ștefan Alexandru Mocanu[2]

This paper presents a study on the general concept of autonomous agents, with an accent on the development of such agents using deep reinforcement learning. This is combined with the domain of autonomous vehicles, as illustrated by a practical application: having a vehicle agent learn how to navigate and park by itself on a designated spot, in a virtual parking lot environment created in Unity. The reinforcement learning method Deep Q–Learning is implemented, with the addition of a few improvements such as Double Deep Q–Learning and Experience Replay.

Keywords: reinforcement learning, Q-learning, autonomous agent, autonomous vehicle, Deep Learning, Experience Replay, Unity3D

1. Introduction

Autonomous agents are software programs developed to react independently to different situations and events, with the purpose of achieving a specific objective. They can be used to solve problems from various fields such as robotics, economics, medicine or games.

Due to an exponential growth in computer hardware in the past few decades, artificial intelligence has become increasingly popular — its applications being present almost everywhere, from mobile phones to online stores. Moreover, there has been a newfound interest in creating software agents capable of adapting to new situations with little to no human intervention while also finding optimal solutions to problems. Therefore, although coined around the 1980s, reinforcement learning has only recently sparked an interest in the field of artificial intelligence.

Another area of great interest at the moment is the subject of autonomous vehicles, multiple companies competing with each other using different methods and technologies in order to become the first to create a fully autonomous vehicle. A less popular approach to this topic, however, is using reinforcement learning. This paper thus tackles this approach in order to study and compare the results to the current state of the art.

[1]Faculty of Automatic Control and Computer Science, University "Politehnica" of Bucharest, Romania, e-mail: clara.barbu@yahoo.ca

[2]Faculty of Automatic Control and Computer Science, University "Politehnica" of Bucharesti, Romania, e-mail: smocanu@rdslink.ro

In theory, an *agent* is any object or being capable of perceiving its environment through sensors (such as eyes, ears or video cameras) and taking action inside this environment using actuators (such as hands, voice or various motors). A *rational agent* is an agent that will always select the action that is expected to maximize its performance measure (through which the agent's level of success is determined), given the agent's history of perceptions and former knowledge regarding the problem that is being solved. [14]

Finally, an *autonomous agent* can be viewed as a rational agent that can interact independently with its environment, using sensors and actuators, by being able to learn. The advantage that comes with this ability is that the agent can initially operate in completely unknown environments and in time become more competent than its initial knowledge may have permitted. More specifically, it is considered that full autonomy is achieved by an agent if it is: *adaptable* to changes that take place in its dynamic environment quickly and efficiently, *robust* by not suffering a total collapse from minor changes to the properties of the environment, and *tactical* in order to maintain multiple objectives and balance their priorities depending on the situation ([1]).

One of the fundamental components of a learning agent is *machine learning*. This field of study seeks to answer the question "How do we create computer programs that improve with experience?" (Tom Mitchell, [10]). Generally, machine learning is categorized into three methods of approaching a problem, depending on the feedback that is being received by the program.

In *supervised learning*, the agent program is given input-output pairs in order to learn from them and be able to then map future inputs to their corresponding outputs. In this case, the inputs are the agent's perceptions, the outputs are labels which provide information about the inputs, and thus the feedback comes instantly. On the other hand, in *unsupervised learning* the feedback is nonexistent and the agent must learn patterns by itself, given a series of inputs. A common method is to separate example inputs into groups called *clusters* and to categorize future inputs into them. Lastly, in the case of *reinforcement learning* the feedback is present, but delayed instead of instant. The agent learns from a series of reinforcements — rewards or punishments.

Reinforcement learning is, by definition, the process of learning what needs to be done (i.e. the mapping of situations to actions) with the purpose of maximizing a numerical signal called a *reward*. Whilst learning, the agent is never told which actions must be taken, but instead it must discover by itself which actions produce greater rewards through trial and error. In some cases, actions can affect not only the immediate reward, but also the subsequent states and thus affecting all future rewards. These two characteristics — searching through trial and error, and delayed rewards — are the most important aspects of reinforcement learning ([19], chapter 1).

Delving into the mathematical component of this subject, the most valuable and most often used formula is the *Bellman equation*. It represents the most important step for all reinforcement learning algorithms, regardless of category (e.g. value–based or policy–based methods). The equation expresses that a state's *utility* — a numerical signal which symbolizes the value of a specific situation, or state, in which the agent can be found at a given moment — can be calculated by adding the current state's immediate reward to the discounted utility of the next state, assuming that the agent selects the optimal action. In the equation, s represents the current state, s' is the next state, U is the utility function, R is the reward function, γ is the discount factor, A is the action space, and P is the probability distribution of reaching state s' from state s by executing action a.

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') \tag{1}$$

Reinforcement learning algorithms can be classified into two categories, the first being *model–based methods* in which the agent has access to a complete model of its environment which contains the transitions between states and the rewards associated to those states, facilitating the search for an optimal strategy by being able to plan ahead efficiently. The second category is known as *model–free methods*, in which the agents bases its decisions on its own experience by learning utilities and/or strategies and gaining an optimal behavior through iterations. The two terms previously mentioned, *value–based methods* and *policy–based methods*, both fall in the second category and are approaches that are currently used in algorithms. The value–based method decides upon the optimal action for each state through a process of repeatedly calculating the utilities for all possible states using the aforementioned Bellman equation and then selecting the actions which return the highest utilities. On the other hand, the policy–based method begins by selecting an initial strategy and then repeatedly evaluating and improving the current strategy until no further improvements can be made.

2. State of the Art

The field of artificial intelligence has recently blown up, with many manual processes (which previously would have been handled by human experts) being automated. This has resulted in many advancements in the field of autonomous cars.

In the development and operation of such vehicles it is considered that the driving agent goes through three phases (repeatedly), each using different technologies and processes. The first is called the *sensory phase* — in which a combination of cameras, radars and LiDAR (Light Detection and Ranging) sensors are used to detect objects around the vehicle, such as other vehicles, pedestrians, obstacles, road signs or car lanes. Cameras are used for detecting and recognizing the shape of objects around the car, radars are used for discerning the speed and location of the objects, while LiDAR sensors are used to create a 3D representation of the environment for an extra boost in (long distance) accuracy. In the *processing phase*

the information detected in the previous phase is interpreted in real time using Deep Learning for Computer Vision for tasks such as image segmentation, image classification or object detection. Lastly, the *control phase* entails further processing of the information in order to guide the vehicle — calculating the optimal route using GPS and correctly adjusting the car's mechanisms while following the rules of traffic.

One of the current leading manufacturers of autonomous vehicles is Tesla, Inc. Its methods are distinguished by the fact that Tesla cars do not use LiDAR technology, instead making use of eight cameras, one frontally positioned radar and short distance ultrasonic sensors. The recorded information from all sensors is then used to train a multi-headed Convolutional Neural Network. Because of this, a strong point of Tesla cars is the automated system that continuously improves the cars' software programs — every day the cars gather new information, which is then used to retrain the networks; once retrained, the software programs are updated with the improved networks, thus taking place a constant increase in the quality of the artificial intelligence used to control the cars. Among Tesla's competitors is Waymo (a subsidiary of Google) whose strength comes from manufacturing all of its components — such as the cameras, radars, LiDAR sensors and chips — allowing for considerable optimization. The most crucial sensor is its LiDAR, which is currently capable of detecting objects at a distance of 300 meters with high accuracy [3], giving the Waymo cars an edge over Tesla. In the future, the number of self-driving cars in use will increase considerably, as other companies such as Uber and Lyft are already testing fully autonomous vehicles in select regions in North America. Despite being tested in the presence of a safety driver, the final objective is using the cars for taxi purposes without a driver.

In the past decade reinforcement learning has gained the attention of numerous researchers, with successes in various fields and one field in particular standing out — games. For instance, *AlphaGo* [2] (developed by DeepMind) was the first software program capable of defeating a professional player and even the world champion at Go, eventually being considered the strongest Go player in history. Go — a two–player strategy board game originating in China — has a total of 10^{170} possible board combinations, making it a very complex game. The agent program uses two neural networks which receive as input information about the board: the first network decides the agent's next move and the second network predicts the current match's winner. Initially, the agent began its training by learning basic strategies from amateur Go players, but then started playing thousands of matches against itself. In this second step, reinforcement learning was used in order for the agent to learn from its own mistakes. After being defeated by AlphaGo in 2016, Lee Sedol, who is considered the best player in the past decade, even stated that the agent was "creative".

While AlphaGo makes use of a model–based approach by having a perfect model of the environment, the following two papers use a model–free approach. *OpenAI Five* [12] is a team of five neural networks (created by OpenAI) which was able to defeat an amateur team at the video game called Dota 2 — a real–time

strategy game in which two teams of five players compete against each other by controlling a character called a "hero". The program trains against itself for 180 years worth of gameplay everyday, on 256 GPUs and 128.000 cores. As opposed to the problem of Go in which the action space (the number of possible actions per agent state) is discrete and the states are observable, in the case of Dota 2 the state and action spaces are continuous. For this reason, the model–free reinforcement learning method called *Proximal Policy Optimization* [15] is used to train the agent.

Another important paper is [11]. It describes a reinforcement learning agent which is trained to play Atari games; tested on seven games, the agent managed to defeat human experts at three of them. The program receives only the pixels on the screen as input and the current score as a reward in order to take in–game decisions. A remarkable aspect about this agent is that when switching to a different Atari game there are no modifications made to the contents or parameters of the software program. Considering the fact that in Atari games the action space is discrete, and also that the state space is enormous (each state is a frame taken from the game screen) the chosen algorithm for training the agent is *Deep Q–Learning* — as opposed to *Q–Learning* in which the state space must be small enough to be stored in a table. This method is further described in the following sections.

Alongside the value–based method Deep Q-Learning and the policy–based method Proximal Policy Optimization, other model–free methods include *Asynchronous Advantage Actor-Critic* (A3C) and *Policy Gradient* (PG). Among those that are based on a model of the environment are *AlphaZero* [16] — a successor of AlphaGo trained to play chess, shogi and Go, and *World Models* [4] — in which the agent learns a model of the environment from its own "dreams".

Although still mostly researched at a theoretical level, a first step has been made for reinforcement learning towards a more practical field — that of autonomous vehicles. In 2018 the company Wayve described in [7] the development of the first reinforcement learning algorithm (which is based on the Actor Critic method) attached to a real car. Using only a series of stills taken in real time as input, the agent learns how to follow a curved road in just 20 minutes, or about 11 episodes. An episode begins by manually positioning the vehicle in the center of the road and switching on the autonomous control until human intervention is needed. At that moment the current episode of exploration is stopped, a reward quantified as the number of meters traveled is given to the agent, and the car is repositioned in order to begin a new episode. The authors of the paper have stated the belief that their method represents an improvement over current algorithms used in autonomous vehicles, as it is inspired by the human trait of adaptability.

Another advancement in the field of autonomous driving is [18], which successfully resolves traffic conflicts and congestion problems. The techniques used include a Deep Q–Network and the Double Q–Learning scheme. The aim is to provide optimal driving policies for multiple vehicles, such that they navigate safely and rapidly to their destinations, avoiding collisions. An approach based on deep

convolutional neural networks is [13]. Using only raw pixels, the agent learns difficult control policies in the OpenAI Gym CarRacing self–driving environment. The authors successfully optimize the Deep Q–Network architecture, surpassing even the base Double Deep Q–Network architecture.

Finally, the initial source of inspiration for this paper's application is the Youtube video [21]. In the video, the agent has a very similar task, and the author also uses Unity ML–Agents [6] for the learning agent. The differences lie in the level of environment generality in the testing phases, and also in the reinforcement learning architecture used. While our architecture uses Double Deep Q–Learning, the author uses the Proximal Policy Optimization algorithm instead.

3. Case Study

The case study of this paper is that of a software agent with a specific objective situated in a simulated environment. The agent is represented by a vehicle, the environment is a virtual parking lot and the objective is to park on a specific parking space.

The vehicle can be controlled simultaneously in two manners: back–and–forth acceleration and braking on the x axis, and direction control by rotating the front wheels on the y axis. The agent's movements are similar to those of a real car and respect the basic laws of physics such as gravity and friction.

Additionally, the vehicle is equipped with eight proximity sensors situated in the front, back, two laterals and four diagonals. The sensors have different sensing distances, with the front and back sensors being the longest, followed by the diagonal and then the lateral sensors.

The virtual environment is ideal because of the lack of bumps on the ground, the lack of wind and any other disruptive elements. The parking lot is thus comprised of: a horizontal floor, eighteen parking spaces, eleven static cars that are already parked (resulting in seven free parking spaces), two entrances to the parking lot from which the agent could start, and a set of (invisible) walls placed around the parking lot in order to keep the agent from wandering outside. The final component of the environment is the target parking space, represented visually by a red rectangle on the ground. The main objective of the agent is to learn how to park on this space.

Training the agent takes place through a series of episodes, and the termination of an episode is triggered by: parking successfully (i.e. navigating to the target spot and slowing down the vehicle), bumping into another car or a wall, or finishing a maximum number of steps per episode (with each step being a single action taken). With every new episode the environment is reset by repositioning the agent at its starting point in order to try again.

Through the use of rewards, the agent is able to discover its objective, which is to navigate through the parking lot while avoiding all obstacles and park on the target spot, at a faster or slower pace. For this reason it is important that the rewards

are chosen accordingly, as they can even be used to encourage a more correct parking position — the more parallel the agent parks to the parking stripes, the bigger the reward is. It is considered that the agent has parked when it is close enough to the center of the spot and its velocity is under a certain speed. The rewards have the following values:

- + *r_step points* for each step in which the episode hasn't ended. The value of *r_step* is calculated using the following equation:

$$r\_step(d) = \left(\frac{-1}{500}d + \frac{1}{10}\right) - 1,\ \forall d \geq 0, \tag{2}$$

  where d is the current distance between the agent and the target spot. Excepting the last term (-1), the function is linear with a negative slope in order to offer the agent a higher reward for being closer to the target. The last term is used as an adjustment, ensuring that $r\_step(d) < 0, \forall d \geq 0$ so as to motivate the agent to hurry in reaching its objective.
- *– 500 points* for colliding with an obstacle.
- *+ 1000 points* for parking on the target spot correctly, at an angle of 0° between the vehicle's x axis and the parking stripes.
- *+ 200 points* for parking on the target spot, but incorrectly, at an angle of 90° between the vehicle's x axis and the parking stripes.
- *+ r_park points* for parking on the target spot at an angle between 0° and 90°. The value of *r_park* is calculated using the following equation:

$$r\_park(\alpha) = \frac{80 * (90 - \alpha)}{9} + 200,\ \forall \alpha \in (0^\circ, 90^\circ), \tag{3}$$

  where α is the angle between the vehicle's x axis and the parking stripes on the ground.

4. Proposed Solution

The proposed solution to the above issue is to separate it into basic reinforcement learning elements (i.e. inputs, outputs, feedback) and to then apply the *Deep Q-Learning* method with a few tweaks in order to train the agent.

The implemented algorithm uses the agent's current state as input, returns the output in the form of the agent's next action, and receives rewards as feedback. The *current state* consists of: the agent's position (x and z coordinates), the target spot's position (x and z coordinates), the agent's velocity (x and z coordinates) and the distances detected by the eight proximity sensors. The *actions* that the agent can make at each time step are represented by an (x, y) vector, where x is the back–and–forth motion (0 = nothing, 1 = forward acceleration, -1 = backward acceleration, 2 = brake), and y is the direction control, or rotation of the wheels (-4 = -45° rotation, -3 = -30° rotation, -2 = -20° rotation, -1 = -10° rotation, 0 = 0° rotation, 1 = 10° rotation, 2 = 20° rotation, 3 = 30° rotation, 4 = 45° rotation). All possible combinations of x and y compose the agent's action space.

4.1. Creating the Environment in Unity

The game engine Unity [22] was used to create the visual component of the application, along with the physics of the environment and the movement of the agent. The agent's behavior and ability to learn are implemented separately in Python. In order for the Unity environment to be used to train the agent developed in Python, Unity ML-Agents Toolkit [6] was used. It contains numerous machine learning algorithms which have already been implemented to train agents, but also features a Python API (Application Programming Interface) which offers the freedom of developing new or personalized algorithms for learning from the environment.

Specifically, the environment was created using a set of "assets" chosen by aesthetics and functionality from the online Unity Asset Store. The assets in this paper's application are from the package called "Polygon City" [20]. The following components were added to the agent's car model asset: *Rigid Body* — which gives an object mass in order to respond to fundamental forces such as gravity, *Mesh Collider* — an invisible layer around the object which can detect collisions with other colliders, and *Wheel Collider* — similar to Mesh Collider, except for being used exclusively on vehicle wheels and allows for realistic movement through specialized functions. Additionally, a C# script is attached to the agent object, whose contents are described in the following paragraphs. The finalized agent and environment can be viewed in figure 3.

Each training episode begins by initializing the agent and environment, which is implemented in the method `OnEpisodeBegin()` from the `Agent` class (from the ML-Agents Toolkit). It is called automatically and is overridden to reset the agent's position and rotation to the starting point (at the entrance of the parking lot), as well as zeroing the velocity. Similarly, any changes to the environment can be done in this method.

During an episode, at each time step, the agent collects a set of information regarding its current state using the method `CollectObservations()` from the `Agent` class. This method then sends the agent's state to the Python component in order to store and process the information. While details such as the agent's position or velocity can be accessed directly (using specialized Unity functions), a method called `Sensors()` was implemented in order to collect the sensor data. In it, the sensors are individually generated using an element called *Raycast*: a laser–like line used to detect collision distances which is defined by a starting position (relative to the agent vehicle object), an angle which describes its direction, and a maximum detection distance. Whenever an object is detected by one of the sensors, the distance to the object is saved and sent back to `CollectObservations()`.

The agent's movement in the environment takes place by choosing a specific action at each time step on the Python side, sending it to Unity, and calling the method `OnActionReceived()` from the `Agent` class, which receives the (x, y) action vector described earlier as input. The two elements from the action vector are then attributed to the following Wheel Collider variables: `motorTorque` or `brakeTorque`

for the back–and–forth movement from x, and `steerAngle` for the direction control from y.

Ending an episode takes place only if at least one of the ending conditions is met. When this happens, the method `SetReward()` — which receives the reward value as input — is called, followed by `EndEpisode()`. Both methods have already been implemented in the `Agent` class. Failure through collision with another object is detected with the Unity function `OnTriggerEnter()`, followed by negatively rewarding the agent and ending the episode. Successful parking is detected by continuously calculating the distance between the agent and the target spot. If this distance is under a specific threshold and the velocity is also small enough, the vehicle's rotation relative to the target spot is used to calculate the given reward (formula 3), followed by the episode end. Besides the above two cases, rewards are given at each time step in `OnActionReceived()` (formula 2).

4.2. Deep Q–Learning and Additional Concepts

Implementing Deep Q–Learning necessitates the action space to be discrete, as it was previously described. The fundamental algorithm on which this method is based is called *Q–Learning*, a model–free value–based algorithm.

The basic principle is to search for the utility values, or *Q–values*, of all state–action pairs and to save them into a table called the *Q–table* (where Q stands for quality). The utility values are calculated using a derived form of the Bellman equation:

$$Q_{new}(s,a) = Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right], \tag{4}$$

where α is the learning rate which determines the speed at which the agent abandons the current Q–value for the new value, γ is the discount factor which is multiplied by the maximum value among the possible actions from state s', and r is the immediate reward from state s. Once the table is complete, the agent then has the possibility of choosing what action to take in which state depending on the maximum action utility value from that state's row [17]. However, when a problem has an extremely large state space, as is the case of this paper (multiple unique states per second leading to millions of states), using an immense table is too inefficient. For this reason, a *neural network* is used to *predict* the Q–values of all possible actions, given a specific state. Thus, the neural network's input nodes are the set of characteristics which describe the agent's state, and the output nodes are the Q–values for each action. A more in–depth description of the implemented algorithm can be found further down in this section.

On top of the fundamental Deep Q–Learning algorithm, a few other techniques have been added in order to improve the performance and robustness of the agent. Firstly, *Experience Replay* [9] is an important method inspired by biology which uses an element called a *Replay Buffer* to store the agent's experiences. An *experience* is defined as a set containing the agent's current state, its chosen action, the new state that action lead to, and a variable which tells if the episode has ended

or not. The technique says that an agent should not learn from its current experiences at every time step, but instead learn from a random batch of experiences from the Replay Buffer every few time steps. The main advantage of using Experience Replay is that the agent's experiences aren't thrown away, but are used multiple times to train the agent; this aspect is important if the agent sometimes finds itself in rare situations. Furthermore, the agent can tend to forget older experiences as time passes, and using this method helps to remember and relearn from them. A final aspect is that in an environment, consecutive states are usually correlated; using a set of random experiences from different moments eliminates the correlations between states and assures a more robust training of the agent.

In the paper [5] on *Double Deep Q–Learning*, the authors demonstrate the fact that using a single estimator can lead to overestimating the Q–values, resulting in weaker performances from the agent. The paper thus proposes using two separate estimators (in this case an estimator being a neural network): one which selects the appropriate action given the current state, and one which estimates the Q–values inserted into the derived Bellman equation in the learning phase. This technique assures that adjusting a neural network's weights is not done in order to predict something that it also generates. In this paper's practical application, the neural network that chooses the action has been named the *local network* and the latter network is called the *target network*.

Soft Target Network Update was introduced in [8] and suggests updating the target network's weights gradually every time Experience Replay is used. Before, the target network weights would be adjusted by matching their values to those of the local network every few time steps, but it has been observed that the adjustments were being made too drastically and were causing a lack of learning stability from the agent. In order to assure a slower and more stable weight adjustment, equation 5 (where θ represents a network's weights or parameters, and $\tau \ll 1$) is used for the target network parameters.

$$\theta_{target} = \theta_{target} * (1 - \tau) + \theta_{local} * \tau \tag{5}$$

Finally, an *epsilon–greedy strategy* has been applied to the agent's decision process. To give context, when implementing an algorithm which selects an agent's next action depending on some value (in this case Q–values), the agent begins to always choose the best suited action based on the information it currently has. This behavior is called adopting a *greedy strategy*. Unfortunately the problem which arises from this is that if the agent always chooses the same actions, it never has the chance to find out if other actions might be better, and thus must be encouraged to explore. To solve this problem, the concept of an epsilon–greedy strategy was introduced, which forces the agent to select a random action with a chance of ε (exploring new possibilities) and to choose the greedy action with a chance of $1 - \varepsilon$ (exploiting gathered knowledge). Generally, the value of ε starts off higher and decreases as the episodes pass (also referred to as the *decaying* epsilon–greedy strategy).

4.3. Creating and Training the Agent in Python

This subsection is structured as the main algorithm of the Python component. The numbered events are detailed in chronological order. The schematic representation of the algorithm is in figures 1 and 2.

1. The Unity environment is initialized together with the application hyperparameters. The environment is opened in an executable file or directly in Unity by using the function `UnityEnvironment()`, from ML–Agents, and saving it in the variable `env`. The next step is to reset the environment using `env.reset()` in order to obtain the agent's Behavior Name and ID (in order to identify the agent), the Action Size, and the State Size directly from Unity where they have been set.

2. The agent is created, which can be trained or untrained. A trained agent receives parameters in order to be observed, but does not do any learning in the next steps. The agent is an instance of the `Agent` Python class (different from the one in ML–Agents), which contains the parameters: `local_network` and `target_network` — instances of the `QNetwork` Python class described further down, `optimizer` — to adjust the neural network parameters using the optimization algorithm `torch.optim.Adam` from PyTorch, `replay_buffer` — an instance of the `ReplayBuffer` Python class also described further down, and `time_step` — a variable which keeps track of the current time step.

The `QNetwork` class is a subclass of `torch.nn.Module` which belongs to PyTorch. The neural network architecture is comprised of 14 input nodes (the components of a state), 36 output nodes (all possible combinations of actions) and 3 hidden layers with 64 nodes each. The layers are fully connected using `torch.nn.Linear()`, and the chosen activation function for each layer is ReLU (Rectified Linear Unit: $ReLU(x) = max(0,x)$).

3. For each episode, repeat the next steps until the maximum number of episodes is reached:

3.1. The environment is reset and the episode's first state is observed. After resetting the environment again (which calls `OnEpisodeBegin()` described earlier), the agent's current state is observed with `env.get_steps()` and saved in the variable `state`. It contains the collected observations from Unity.

3.2. For each time step, repeat the next steps until the max number is reached or an exit condition is met:

3.2.1. The agent's next action is selected. To do so, the method `act()` is called, which belongs to the `Agent` Python class. It receives the agent's `state` and the value of ε as input. Then, using the *local* neural network, the actions' Q–values

are predicted and the agent's next action is selected using the epsilon–greedy strategy (either the action with the highest Q–value with a chance of ε%, or a random action otherwise). The value of ε is higher at the beginning of training and decreases gradually in order to move from exploration to exploitation.

3.2.2. The agent's reward is observed together with the new state. The previously chosen action is sent to the environment using `env.set_actions()`, followed by `env.step()` in order to advance the agent to the next time step and generate a new state. Once the agent enters this new state, it is saved in the variable `next_state` (again using `env.get_steps()`). Additionally, because choosing an action has triggered the receipt of a reward before moving on to the next state, this reward is saved in the variable `reward` to be used in the learning phase.

3.2.3. The learning phase. By calling `step()` (a member function of the `Agent` Python class) with the newly collected information as input, the agent begins the process of learning and updating its parameters. Firstly, the new experience is added to the agent's `replay_buffer`. This is an object which is defined by an `experience` — a namedtuple under the form of which the information is stored, a `buffer` — a chronologically ordered list containing all of the agent's experiences, and a `batch_size` — the number of random experiences sampled from the `buffer` at each learning phase. As such, the `ReplayBuffer` class's method `add()` is used to add the agent's state, action, reward, new state and whether the episode is finished or not, to the `buffer` as an `experience`. Next, the `time_step` is incremented and checked if it is a multiple of four. If so, `get_sample()` (from `ReplayBuffer`) is called to extract a batch of experiences from the buffer, which is further used as input for the method `learn()` (from `Agent`). The idea of learning only at every four time steps was implemented through trial and error in order to increase stability by learning slower and avoiding a sudden increase in performance by the agent, followed by an immediate collapse.

In `learn()`, the batch of experiences is first separated into lists: `states`, `actions`, `rewards`, `next_states`, and `dones` (which are equal to 1 if the respective episode is finished and 0 otherwise). So as to update the parameters of the `local_network`, the loss function is calculated between the lists `Q_targets` and `Q_values`. The first term is computed using the following formula, which is the Bellman equation [19] implemented in Python:

```
Q_targets = rewards + (gamma * Q_next * (1 - dones)),        (6)
```

where `gamma` is the discount factor and `Q_next` is the list of maximum Q–values for the new states, $\max_{a'} Q(s',a')$. In order to obtain this list, the target network is used to predict all of the Q–values for each state from `next_states`, and then the maximum value is selected from each set of actions per state. The last component of the formula is multiplied by `(1 - dones)` because a terminated episode does not contain a `next_state`, and so this bit would not be needed in that case.

For computing the second term, `Q_values`, the local network is used to predict the Q–values of all state–action pairs from the sampled batch (i.e. for each element from `states`, the Q–value of the corresponding element from `actions`).

Basically, the loss function is calculated between the "real" Q–values obtained from the Bellman equation and the target network (`Q_targets`) and the current Q–values of the previously chosen actions predicted by the local network (`Q_values`). The selected loss function is the Mean Squared Error Loss. After backpropagating the loss, the local network's parameters are adjusted using the Adam optimization algorithm. Lastly, the parameters of the target network are also updated using the method `soft_update()` (from `Agent`) which applies equation 5.

3.2.4. `state` ← `next_state`

3.2.5. Verify exit conditions. If the agent has collided with an obstacle from the environment or has successfully parked on the target spot, the current episode ends and the algorithm breaks from this loop.

3.3. The value of ε *is decreased.*

4. The local network's parameters are saved to be used in the future. This is done with `torch.save()`, which saves the parameters file to disk. It allows for the file to be later used to observe the behavior of the trained agent at a normal environment speed, by calling `load_state_dict(torch.load())` with the file name as input.

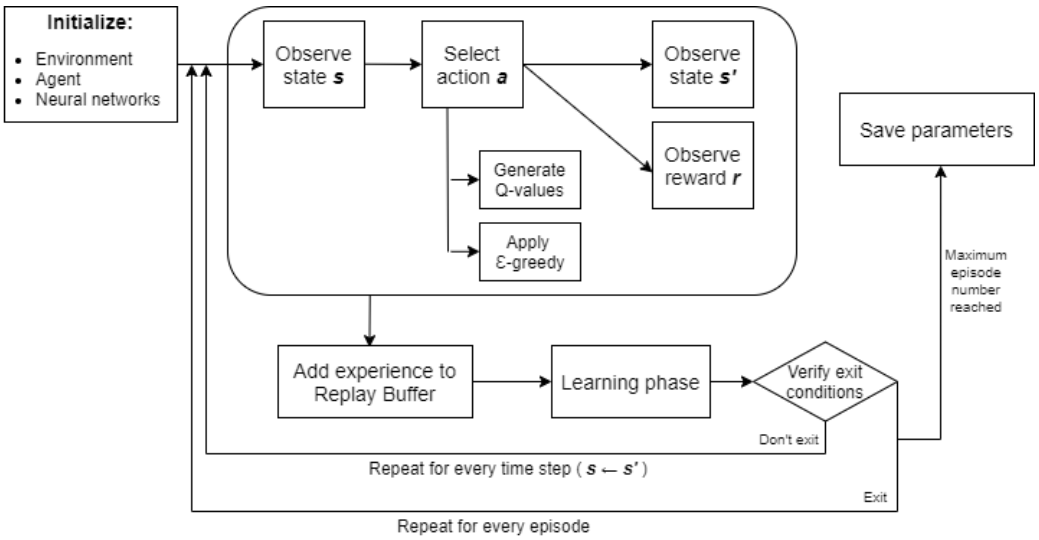


Fig. 1. Schematic representation of the main algorithm described in subsection **4.3**. The learning phase is represented in more detail in figure 2.

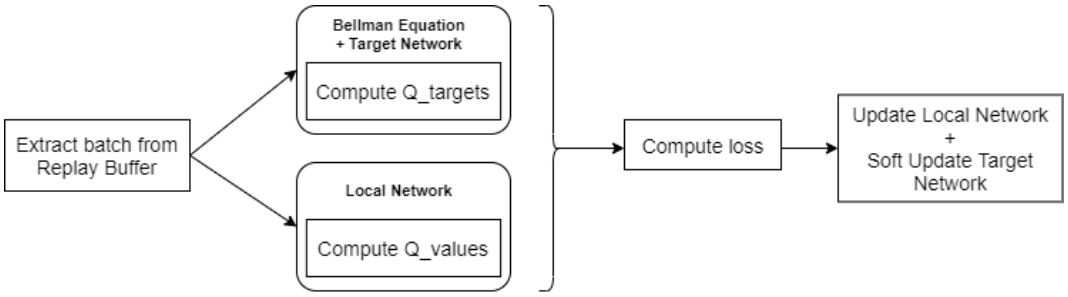


Fig. 2. Schematic representation of the learning phase segment from the main algorithm.

4.4. Hyperparameters

The algorithm makes use of a set of hyperparameters whose values can highly influence the agent's behavior. The process of choosing the values implied studying similar applications and also making multiple adjustments until the desired behavior was obtained. This subsection presents each hyperparameter's chosen value and usage.

- Number of episodes = 8.000. This determines how much time the agent has to develop its behavior.
- Maximum time steps per episode = 500. So as to avoid the agent getting stuck, this hyperparameter represents the maximum number of actions per episode.
- $\gamma = 0.99$. The discount factor from the Bellman equation (1). It influences the importance of future states when calculating the Q–values for the current state. In this paper's case, consecutive states are strongly correlated, resulting in a value that is very close to 1 (the maximum value).
- $\alpha = 0.0005$. The learning rate from the derived Q–learning Bellman equation (4). It determines how quickly the agent abandons the current Q–value in exchange for the new value (or in other words, how much the neural network learns at each iteration).
- Batch size = 64. The number of experiences that are sampled from the Replay Buffer at each learning step.
- $\tau = 0.001$. The parameter from equation 5. It represents the percentage with which the target network's parameters are updated, from the local network.
- $\varepsilon_{initial} = 0.1$, $\varepsilon_{decay} = 0.995$, $\varepsilon_{limit} = 0.01$. The hyperparameters which determine the value of ε when applying the epsilon–greedy strategy. The first is the starting value, the second is the rate of decay (the current value is multiplied by this parameter every episode), and the last is the lower limit under which it stops decaying.

5. Results and Discussion

While developing the practical application for this paper, a simplified environment was created prior to the one described in the previous section. This environment contains a 3D ball whose objective is to touch a cube by rolling around on a small flat platform and trying not to fall off the edge. The level of simplicity meant quicker testing times (10–15 minutes vs. 1–3 days for the vehicle environment), permitting more attention to be given to the development and adjustment of the reinforcement learning algorithm, as well as studying the effects of the hyperparameters on the agent.

For instance, multiple neural network architectures were considered for the agent. While experimenting, it was observed that although it may seem that adding more layers and nodes leads to better results, this was not the case: an architecture of 3 hidden layers with 32 nodes each obtained much worse results than an architecture of 2 hidden layers with 32 nodes each. In the end, an architecture of 3 hidden layers with 64 nodes each was chosen for both the simple and complex environments (with differences only in the number of nodes in the input and output layers).

The use of Double Deep Q–Learning was also compared. It was observed that the agent's average score increased more chaotically (and barely stabilized on a good score) when not using the technique, as opposed to a stable increase (and maintaining a near perfect score) toward the end of training.

Moving on to the main environment, testing the agent was done in three phases which differ from each other by the level of environment generality. While the main objective of this paper had already been reached in the first phase, the following phases were created to push and test the limits of the agent's capabilities. Figure 3 shows the agent and the environment in different phases.

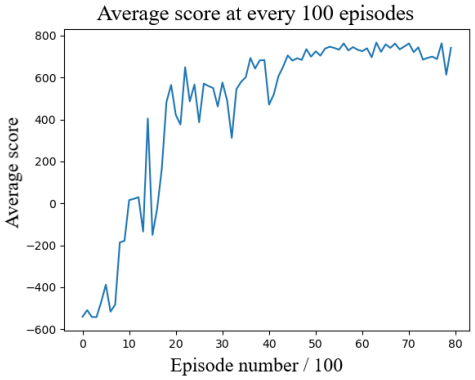
In the first phase, the environment is exactly the one described in section **3**. Before remaining with the current reward system, multiple adjustments had to be made: The initial system positively rewarded the agent for moving closer to the target even slightly, and negatively rewarded it for the opposite. Unfortunately, this method led to the agent staying still and occasionally moving a bit toward the target so as to not lose any points (without any intent of parking). After adding a negative reward for not moving at all, it was observed that the system was unbalanced: the agent could obtain a higher score by continuously and slowly moving toward the target than by actually parking correctly. Following further adjustments to the values for balance, the method described earlier was dropped and replaced with equation 2, which constantly rewards the agent negatively for lost time and does it more drastically the further away the agent is from the target.

Figure 4a displays the agent's results in this first phase. The value shown is the average score (summarized rewards) over a span of 100 episodes. At the beginning of training the agent's behavior is random, crashing often into obstacles, but slowly starting to move toward the target as the episodes pass. The behavior then visibly begins stabilizing at the half of training, parking more and more frequently on the target spot. In the final couple thousand episodes the agent is able to continuously

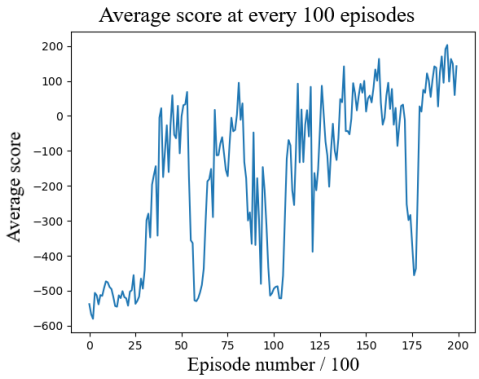
Fig. 3. The agent (blue car) and its environment created in Unity. The left sides are zoomed out versions of the right sides, in which the agent's sensors are visible. In the first set of stills, the agent is in the process of parking on the target spot of the first phase of testing. In the second set, the agent is in the third phase of testing in which the static cars are arranged randomly in the parking lot. This is the spot that the agent has learned best to park on.

park correctly without mistakes, the only differences in score being made by the angle of parking.

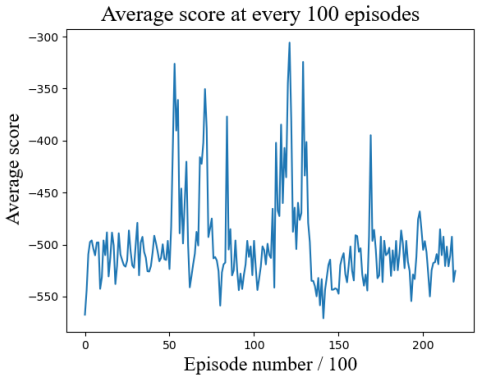
This successful result demonstrated that the reward system had been chosen correctly, and so a second, more difficult to solve, phase was created. In order to generalize the environment, an additional feature was added to `OnEpisodeBegin()`: at the beginning of each episode, the agent has a 50% chance of spawning at each entrance to the parking lot (as opposed to always starting from the same spot). Alas, it had not been predicted that occasionally spawning at a much higher distance from the target could pose a problem for the agent. For this reason, the results turned out to be inferior to the previous ones (figure 4b). After approximately 10.000 episodes the agent had already learned to park when starting from the closer entrance, but decided it would be wiser to stay still when beginning from the opposing entrance.



(a) Results obtained by the agent in the first phase, in 8.000 episodes, in approximately 4 hours.



(b) Results obtained by the agent in the second phase, in 20.000 episodes, in approximately 27 hours.



(c) Results obtained by the agent in the third phase, in 22.000 episodes, in approximately 3 days.

Fig. 4. The agent's average score at every 100 episodes, in the first (A), second (B) and third (C) phase.

Presumably, the agent preferred the score obtained for not moving the entire episode (or moving slowly and surely toward the target) rather than risking a traversal of the parking lot followed by a massive negative reward for colliding with an obstacle.

In an effort to bring the target spot physically closer to the agent's starting position at every episode, but also to generalize the environment even more and test the agent's limits, a third phase was created; this brought a few more changes to `OnEpisodeBegin()`. Besides randomly interchanging the agent's starting point like in the previous phase, the static cars' positions in the parking lot are randomly mixed each episode, generating a multitude of unique parking lot configurations that the agent can learn from and adding complexity to the scene. Furthermore, the agent's target spot is dynamically calculated for each configuration as the closest available parking spot to the starting position, and is maintained for the entirety of

the episode. The obtained results are displayed in figure 4c. It can be observed that the agent had a few spikes of good behavior around the half point of training, but dropped to average scores between -550 and -500 afterward; scores in this interval indicate that the agent generally ended the episode by colliding with an obstacle. Because of the complexity of this environment, I believe that the agent could have achieved better results (and even learned to park) had it trained for at least a triple amount of episodes. Unfortunately, the training had to be interrupted after 3 days of training due to lack of hardware processing power. This was caused by the vast amount of accumulated experiences stored in a vector which is processed multiple times per second. On the other hand, because the neural network parameters are saved to disk every thousand episodes, it was possible to observe the agent's best behavior from the training. Running the application with the parameters from episode 12.000, the agent was able to park correctly in $15-25\%$ of the episodes and acquired an average score between -200 and -250. Behaviorwise, the agent had clearly learned to park on a specific spot (whenever it was available) and in other cases it tended to slowly move toward the target while avoiding obstacles, but without finalizing the parking.

6. Conclusions

To sum up, this paper's intention is to realize a study on the concept of autonomous agents and their creation using artificial intelligence, diving deeper into the method of reinforcement learning. To illustrate, a practical application was implemented, whose main objective is to prove that an autonomous vehicle can learn to park by itself on a designated spot in a simulated environment.

Compared to the results in Arzt's paper mentioned as a source of inspiration for our work: Performance–wise, the agent from the video learned how to park in 310.000 episodes, while this paper's agent only needed 6.000 episodes in a less general environment and showed signs of improvement after 20.000 episodes in a similarly general environment.

Further improvements can surely be made to the application. The hyperparameters can be adjusted even more by testing out various combinations, and the code or algorithm itself could be optimized further so as to allow for a higher number of training episodes. Using a more powerful unit is also an option, as well as Google's Cloud TPU (Tensor Processing Unit) which was created specifically for neural network machine learning. Another possible improvement could be shifting to policy–based Actor Critic methods, which would allow continuous spectrums for the agent vehicle acceleration, as well as the wheel rotations. This would lift the current limitations of the agent movement, which may have caused the agent to be more imprecise when driving. An idea could be to implement the already successful algorithm Proximal Policy Optimization. A different direction could be swapping the vehicle's proximity sensors with a front facing camera, or even gathering input from both. This would imply the use of Convolutional Neural Networks for the task of image processing.

In closing, the concept of a vehicle agent learning how to park on a specific spot could potentially lead to solving even more complicated tasks, as reinforcement learning is continuously proving to be more and more powerful. It can be used to optimize problems not only in the automotive industry, but also in a multitude of different domains such as robotics, games or medicine.

REFERENCES

[1] *R. A. Brooks*. Intelligence without representation. Artificial Intelligence, **volume 47** (1-3):pages 139–159, 1991.

[2] *DeepMind Technologies*. AlphaGo. accessed 2020.
URL `https://deepmind.com/research/case-studies/alphago-the-story-so-far`

[3] *M. della Cava*. Waymo CEO Krafcik has 'a lot of confidence' his tech would have avoided deadly Uber accident. USA Today, 2019, accessed 2020.
URL `https://eu.usatoday.com/story/tech/2018/03/25/waymo-ceo-krafcik-has-lot-confident-his-tech-would-have-avoided-deadly-uber-accident/456819002/`

[4] *D. Ha and J. Schmidhuber*. World Models. arXiv preprint arXiv:1803.10122, 2018.

[5] *H. V. Hasselt*. Double Q-learning. In *J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta*, editors, Advances in Neural Information Processing Systems 23, pages 2613–2621. Curran Associates, Inc., 2010.

[6] *A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange*. Unity: A General Platform for Intelligent Agents. arXiv preprint arXiv:1809.02627, 2020.

[7] *A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah*. Learning to Drive in a Day. arXiv:1807.00412v2 [cs.LG], 2018.

[8] *T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra*. Continuous Control with Deep Reinforcement Learning. arXiv:1509.02971v6, 2019. Published as a conference paper at ICLR 2016.

[9] *L.-J. Lin*. Reinforcement Learning for Robots Using Neural Networks (CMU-CS-93-103). Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[10] *T. M. Mitchell*. Machine Learning. McGraw-Hill Science/Engineering/Math, 1997. ISBN-13: 9780070428072.

[11] *V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller*. Playing Atari with Deep Reinforcement Learning. arXiv preprint arXiv:1312.5602, 2013.

[12] *J. Pachocki, G. Brockman, J. Raiman, S. Zhang, H. Pondé, J. Tang, F. Wolski, C. Dennison, R. Jozefowicz, P. Debiak, et al.* OpenAI Five. https://blog.openai.com/openai-five, 2018, accessed 2020.

[13] *P. Rodrigues and S. Vieira*. Optimizing Agent Training with Deep Q-Learning on a Self-Driving Reinforcement Learning Environment. In 2020 IEEE Symposium Series on Computational Intelligence (SSCI), pages 745–752. IEEE, 2020.

[14] *S. J. Russell and P. Norvig*. Artificial Intelligence: A Modern Approach. Third Edition. Pearson Education, 2010. ISBN-13: 9780136042594.

[15] *J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov*. Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347, 2017.

[16] *D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al.* A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, **volume 362** (6419):pages 1140–1144, 2018.

[17] *T. Simonini*. Diving Deeper into Reinforcement Learning with Q-Learning. freeCodeCamp, 2018, accessed 2020.

URL https://www.freecodecamp.org/news/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe/

[18] *C. Spatharis and K. Blekas*. Double deep multiagent reinforcement learning for autonomous driving in traffic maps with road segments and unsignaled intersections. In 2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC), pages 1–6. IEEE, 2020.

[19] *R. S. Sutton and A. G. Barto*. Reinforcement Learning: An Introduction. Second Edition. MIT press, 2018. ISBN-13: 9780262039246.

[20] *Synty Studios*. POLYGON - City Pack, accessed 2020.
URL https://assetstore.unity.com/packages/3d/environments/urban/polygon-city-pack-95214

[21] *Samuel Arzt*. AI Learns to Park - Deep Reinforcement Learning, 2019, accessed 2020.
URL https://www.youtube.com/watch?v=VMp6pq6_QjI&t=230s

[22] Unity Technologies. Unity3D Documentation, accessed 2020.
URL https://docs.unity3d.com/Manual/index.html