Acknowledgements

I would like to thank for the support and encouragement my father, Valentin, my brother, Tudor, my mother, Daniela, and my girlfriend, Adelina.

I would also like to thank my supervisor for all the guidance and help.

Table of Contents

1. Introduction

I have chosen to approach the topic of reinforcement learning for continuous action spaces thanks to its high relevance in the current development and the future of artificial intelligence. Continuous action spaces can be associated with everything from the movement of an arm to flying a helicopter (Abbeel, Coates and Ng, 2010) and driving a car. Learning to adapt in an unknown environment is something we humans have been doing for thousands of years and most recently machines became able to do so as well, which benefited immensely the autonomous driving and robotics industries (Kober, Bagnell and Peters, 2013). I considered that a model-free off-policy problem, which implies learning by trial-and-error without prior knowledge of how the environment behaves, would be of most practical importance. For example, learning how to drive a car requires multiple attempts even for humans and hard-coding an algorithm that would be able to drive is probably going to take longer than letting the algorithm teach itself, given the complexity of our world and the endless factors that have to be taken in account. An alternative to reinforcement learning for teaching an agent to perform such tasks is imitation learning, but it involves a human supervisor that would offer examples of behaviors which means that every time a new environment or situation is presented to the algorithm a human supervisor has to provide an example. This does not achieve great generalization and is also time-consuming. From this point of view, in reinforcement learning the algorithm is both the supervisor and the action taker, something that becomes more apparent in actor-critic architectures.

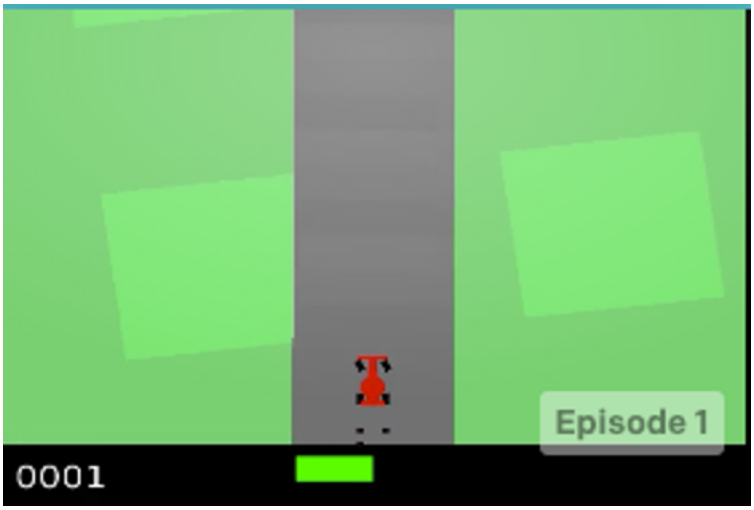
A very popular example of this intertwining has been showcased when the Deep Q Network algorithm (Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra and Riedmiller, 2013) was first introduced, achieving human level performance while playing many Atari games using as inputs only raw pixels, an approach which I also use throughout this thesis in many of the algorithms. The main concept with which this algorithm has innovated is that it uses a deep neural network in order to approximate the action-value function and a replay buffer in order to alleviate the problem of correlation in the data. The algorithm on which this innovation is built on is the Q-learning algorithm, which cannot tackle efficiently problems with a large number of state-action pairs, as it usually runs into the so-called curse of dimensionality (Bellman and Ernest, 1957).

One further improvement to this algorithm is the Double Deep Q Network (van Hasselt, 2010), which does not use online learning, employing two different deep neural networks, one for prediction and one for training, which overcomes the problem of overestimating values of

certain actions that we can come across while training models with Deep Q Networks. Where these previous models need a discrete action space, Deep Deterministic Policy Gradients (Lillicrap, Hunt , Pritzel, Heess, Erez, Tassa, Silver and Wierstra, 2015) can deal directly with a set of continuous actions as it does not require to optimize for each individual action, but directly on the control policy using gradient descent. One other popular alternative to reinforcement learning algorithms are evolutionary algorithms, which are built on the Darwinian principles of evolution (Darwin, 1859), and can provide similar performance.

However, combining reinforcement learning with deep neural networks is quite delicate, as the latter require large datasets and the data in reinforcement learning problems consists only in past experiences, more precisely the reward signal, which is delayed, noisy and sparse. Moreover, most deep learning algorithms consider the inputs independent, which is not the case in reinforcement learning (RL) as the states, which are the inputs, are highly correlated. Furthermore, deep neural networks expect data having the same distribution, whereas in RL the distribution of the data changes as the algorithm improves. Due to these reasons I have decided to test which of these algorithms fares better in coupling neural networks with RL for continuous action spaces, more precisely autonomous driving, and determine which of these has the greatest potential to contribute to the development of autonomous motion in general. I am also going to explore for each algorithm which parameters are the most influential in terms of reaching an optimal policy and learning experience.

2. Background



In this thesis most of the algorithms will be applied to an environment made available by

OpenAI through their OpenAI gym, called CarRacing-v0. It is a continuous control task and is represented by a top-down racing environment. The state consists of a 96x96 pixels frame. The reward is -0.1 for each frame and +1000/N for every tile visited, N being the number of total tiles in a track. As an example, if the episode finished in 629 episodes, the reward would be 1000 0.1*629 = 927.1 points. An episode finishes when all tiles are visited, or the maximum number of iterations is reached. Initially, the number of iterations allowed was 300 but I changed the number to 1600 in order to allow the algorithm to explore most of the states and observe how it behaves in longer runs. The environment renders the movement of the car and in the frame, besides the environment and care itself the are several sensors, from left to right: true speed, four ABS sensors, steering wheel position and gyroscope. The environment is considered solved when the agent achieves consistently a reward of 900 in under 100 episodes.

2.1. Reinforcement learning

Reinforcement learning, according to Sutton et al (2018), can be seen as the computational way in which an agent would learn from interacting with the environment surrounding him. As a child learns from experimenting with the different actions that he disposes of and observes the cause and effect of his choices he starts to comprehend which course of action he should take when faced with a certain situation. In a very similar manner, an algorithm can learn which action from his action space he should choose depending on the state it finds itself in. This similarity that algorithms share with the way in which humans learn is the main reason why I am fascinated about this method, as it in a way can replicate our behavior and even surpass it, as it was the case with many of the algorithms I am talking about in this thesis (Mnih et al, 2013, Lillicrap et al, 2015).

This type of learning is under the spectrum of machine learning techniques but stands in a category of its own as it is goal-oriented and aims to maximize a reward function. It can not be considered a supervised learning algorithm as the latter type of learning implies a predefined training dataset and the ultimate goal of the algorithm is to extrapolate in order to behave well on newly presented data, whereas in problems involving interaction having a dataset with all the possible situations in which the agent can find itself in is at least impractical. It can not be classified as unsupervised learning either as that implies finding hidden relationships within the provided dataset, something that could be useful also in an interactive setting but contributes nothing to the maximization of the reward function.

Any reinforcement learning problem includes the following four elements: a policy, a reward signal, a value function, and sometimes a model of the environment, which will not be the case in the analysis we will perform.

The policy represents the way in which the agent would behave at a given point in time. It can be considered the mapping from perceived states to actions. Even though all the algorithms we consider in this thesis are off-policy, which means they are not improving on an existing policy, we could consider the neural networks used as our policy, as they approximate value-functions. Policies represent the core element of reinforcement learning as they are key to determine an agents behavior.

One other very important concept is the reward signal, which defines the goal the agent wants to attain. On each time step the agent received a numeric value called a reward. This value can be larger or smaller depending on how appropriate the action that the agent takes is for the state that he finds itself in, thus defining whether it is good or bad. The agents sole purpose is to maximize this reward in the long run. The reward function stays consistent over time and there is no way in which an agent would be able to change the way in which this function behaves. The only thing that an agent can change is the value of the reward signal through his actions. We will see that the way in which a reward function is configured can heavily alter the performance of an algorithm. In the previously mentioned environment, the reward function provided by OpenAI gym has proven appropriate when trying to solve the environment with a discrete set of actions but problematic when an approximation function for a continuous set of actions has been implemented.

We can consider the reward signal the immediate consequence of an agents action whereas a value function takes a broader perspective on the overall problem. Basically, the value of a state represents the total reward an agent can expect to obtain a given state. Rewards are of primary importance in reinforcement learning as values, prediction of rewards, can not exist without them. However, when making and evaluating decisions of primary importance are the values, as action choices are taken based on value judgements. Actions are chosen based on which generates higher values, not higher rewards, as we are concerned about the performance on the long run. Whereas rewards are clear cut and easy to determine, values are harder to estimate in an efficient and consistent way, as they are derived from observations of the sequence of observations an agent creates over the whole number of episodes. The approximation of value functions represents the biggest hurdle in almost any reinforcement learning problem, an issue which I am going to tackle by using deep neural networks.

The last element of a reinforcement learning problem is the model of the environment.

This element allows inferences to be made about the way in which the environment will behave. By being provided a state and an action the model might be able to predict which are the next states and rewards. In planning problems models play a key role as this allows the agent to consider future situations even before these occur. These types of problems are called model-based but I am not going to expand further on these as they are not of interest for this thesis. The problems I am tackling are model-free, meaning that the agent learns only by trial and error, which is considered the easiest type of reinforcement learning problems.

When reinforcement learning problems are simple, that is the state and action spaces are small, tabular methods are used, meaning that the value of the action-value function and the associated state and action can be included in a table. There are three classes of methods used to solve finite Markov decision problems: Monte Carlo methods, dynamic programming and temporal-difference learning. Even though the first 2 are not of interest to this thesis as they are not suitable for step-by-step incrementation and, respectively, require a model of the world (Sutton and Barto, 2018) they all share generally one property, the Markov Property, a property inherent to the state signal. This translates to having a state signal that summarizes past experiences/sensations in the most condensed way such that all relevant information is retained. In the case of the CarRacing environment, current velocity, positioning of the steering wheel and position are everything that matter for its future position, previous information not being relevant. This information can be retrieved in a number of ways. In this thesis I am using convolutional neural networks to retrieve it directly from pixels as well as other preprocessing techniques that would return scalar values for the aforementioned variables. In order to express the Markov property mathematically let us consider a finite number of states and reward values and consider how the agent would behave at iteration t+1 given an action taken in the previous iteration. The formulas presented in this chapter can be found in (Sutton and Barto, 2018). The probability distribution, considering a general case where future actions depend on all previous ones, would be the following:

$$Pr\{R_{t+1} = r, S_{t+1} = s^{'} \| S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{1}$$

for all r, $s^{'}$ and $S_i, A_i, i\epsilon[0, t]$. If, on the other hand, we would consider a state signal which satisfies the Markov property, the response at iteration t+1 would only depend on the

state and action representation at iteration t:

$$p(s', r \| s, a) = Pr\left\{R_{t+1} = r, S_{t+1} = s' \| S_t, A_t\right\} \tag{2}$$

for all r, $s^{'}$ and S_t, A_t. Therefore, we can say that the state signal satisfies the Markov property is equal for (1) and (2) for all possible values of the variables. A reinforcement learning problem that satisfies the aforementioned property is called a Markov Decision Process (MDP). Such a reinforcement learning task allows the computation of the following values: expected rewards for state-action pairs,

$$r(s, a) = \mathbf{E}[R_{t+1} \| S_t = s, A_t = a] = \sum_{r \epsilon R} r \sum_{s' \epsilon S} p(s^{'}, r \| s, a) \tag{3}$$

state-transition probabilities,

$$p(s^{'} \| s, a) = Pr\left\{S_{t+1} = s^{'} \| S_t = s, A_t = a\right\} = \sum_{r \epsilon R} p\left(s^{'}, r \| s, a\right), \tag{4}$$

and expected rewards for state-action-next-state triples,

$$r(s, a) = \mathbf{E}[R_{t+1} \| S_t = s, A_t = a, S_{t+1} = s^{'}] = \frac{\sum_{r \epsilon R} r p(s^{'}, r \| s, a)}{p(s^{'} \| s, a)}. \tag{5}$$

Now I will go back to the definition of a value function and how it behaves given a MDP. Recall that π is the policy that maps each state, s $\in$ S, and action, a $\in$ A(s), to the probability $\pi(a|s)$ of taking action a when in state s. The value function can then be defined as

$$v_\pi(s) = \mathbf{E}_\pi[G_t \| S_t = s] = \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \| S_t = s] \tag{6}$$

With $\mathbf{E}_\pi[.]$ being the expected value of a variable given the policy that the agent follows and and t the iteration. For terminal states this value is always zero. This function, v_π, is called the state-value function for policy π. One other important definition especially for temporal difference learning is the q value, or the action-value function for policy π, which is defined as

$$q_\pi(s, a) = \mathbf{E}_\pi[G_t \| S_t = s, A_t = a] = [\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \| S_t = s, A_t = a] \tag{7}$$

This represents the value of taking action a in state s under a policy π, expressed mathematically as the expected return starting from state s, performing action a while following policy π. These two functions will be estimated from experience. When using Monte Carlo methods these functions can be represented as averages of the rewards following a state but this method is inefficient from a storage point of view in the environment that we consider as there is a very large number of possible states. In our case the action-value and value function are kept as parametrized functions which can improve their fit to better match the actual reward by modifying their parameters. One important aspect of the value function is that it satisfies a r

$$v_\pi(s) = \sum_a \pi(a\|s) \sum_{s',r} p(s', r\|s, a)[r + \gamma v_\pi(s')] \tag{8}$$

known as the Bellman equation. This expresses a relationship between the value of a state and that of the successor states. This equation is vital in many reinforcement learning algorithms as enables to approximate action-value functions by an iterative update of parameters.

Going back to the main environment considered in this thesis, the action space is continuous. One way in which we can deal with continuous action spaces is to discretize them and consider only a limited number of actions, so that algorithms such as Q-learning or Deep Q Learning can be used. One pitfall of this method is that when discretizing the action space is that the distribution of the actions has to reflect how a real driver would behave, otherwise particular moves would be preferred to other, something that became obvious when performing experiments with this approach. One other way in which this problem can be dealt with is by using algorithms such as the deterministic gradient policy, which I will expand on later in this thesis.

Within the class of reinforcement learning algorithms there is one category of high importance to this thesis. This class is called temporal-difference learning, which combines ideas from Monte Carlo control, it can learn from trial and error experiments, and dynamic programming, allowing the agent to update estimates without waiting for a final outcome (bootstrapping) (Sutton and Barto, 2018).

I am going to make the standard assumption that rewards are discounted by a factor at each time-step and therefore the return at time t becomes $R_t = \sum_{t'=t} \gamma^{t'-t} r_{t'}$, T being the maximum number of iterations allowed in an episode.

TD learning, like Monte Carlo methods, learns from experiences derived from a policy

π and both update the estimate of their value function for any non-terminal state S_t. One example of an every-visit MC method is the following: $V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$, with G_t being the return following iteration t and α a constant, from which the name of the method, constant α - MC.

Where the novelty of TD shows is that while MC methods must wait for the episode to end (otherwise G_t cannot be computed), a TD method can perform the increment to V(S_t) at every iteration, using the R_{t+1}, the reward observed at the following iteration and the estimate of the value of the future state. Such an algorithm is TD(0): $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$ (Sutton and Barto, 2018). The way in which TD methods resembles DP is that when computing the state value function $v_\pi(s) = \mathbf{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \| S_t = s]$, instead of using the state value function for S_{t+1}, TD uses the current estimate V(S_{t+1}), as the previously mentioned value is not known.

Before explaining in detail the central algorithm of this thesis one other algorithm in the TD class deserves some words and that is Sarsa, the on-policy counterpart of the Q learning algorithm. The naming of this algorithm comes from the main components of an episode: $S_t, A_t, R_t, S_{t+1}, A_{t+1}$ (SARSA). This algorithm is built on generalized policy iteration and estimates the action-value function $q_\pi(s, a)$ instead of the state-value function. Transitions are now considered from state-action to state-action and the value for each state-action pair is computed. The update process for this algorithm is as following:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{9}$$

Updates are performed for each non-terminal state and if S_{t+1} is terminal the value defaults to zero. As in any on-policy method, the action value for the behavior policy π is continuously estimated while also changing π so that q_π keeps improving.(Sutton and Barto, 2018)

In order to find the optimal value function $Q^*(s, a)$ the maximum return achievable by following any strategy after observing a sequence s and an action a has to be taken, $Q^*(s, a) = max_\pi \mathbf{E}[R_t \| s_t = s, a_t = a, \pi]$, where π is the policy that is being followed.

This function satisfies the Bellman equation and the action-value function at (s,a) can be rewritten in a recursive form as a function of the successive states and actions. (Mnih et al, 2013) Intuitively, at the successive state $s^{'}$, considering that all possible optimal values are known for each action $a^{'}$, the problem becomes the choice of action $a^{'}$ such that it maximizes the expected value of $r + \gamma Q^*(s^{'}, a^{'}), Q^*(s, a) = \mathbf{E}_{s' \sim \varepsilon}[r + \gamma max_{a'} Q^*(s^{'}, a^{'}) \| s, a]$.

This allows for the iterative update that I have previously mentioned, $Q^*_{i+1}(s, a) =$

$\mathbf{E}_{s'\sim\varepsilon}[r + \gamma max_{a'} Q_i^*(s', a')\|s, a]$, which converges to the optimal value function as i $\rightarrow \infty$. (Sutton and Barto, 2018) In traditional reinforcement learning the to-go algorithm for off-policy and model-free problems is Q-learning (Watkins, 1989), the off-policy alternative for Sarsa. The one step Q-learning makes use of the previously explained Bellman equation and updates the value function iteratively using the following equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{10}$$

Unlike Sarsa, Q-learning approximates the optimal state value function directly, instead of relying on the policy π that is being followed. However, this process is impractical, as the action-value function is estimated for each sequence, thus lacking generalization (Mnih et al, 2013). Instead of directly computing the action-value function a very common procedure is to use an approximator for the aforementioned function $Q(s, a : \theta) \approx Q^*(s, a)$, which can be linear or non-linear, like a neural network. The neural network function approximator having weights θ will be referred to as a Q-network. This type of approximator, according to Mnih et al, (2013), can be trained by minimizing a sequence of loss functions at each iteration i,

$$L_i(\theta_i) = \mathbf{E}_{s,a\sim p(.)}[(y_i - Q(s, a; \theta_i))^2] \tag{11}$$

,with $y_i = \mathbf{E}_{s,a\sim}[r + \gamma max_{a'} Q(s', a'; \theta_{i-1})\|s, a]$ the target for iteration i, and $\rho(s, a)$ is a probability distribution over sequences s and actions a that we refer to as the behaviour distribution. When performing the optimization $L_i(\theta_i)$ the parameters of the previous iteration θ_i remain constant.

The gradient used in the optimization is obtained by differentiating the loss function with respect to the weights:

$$\bigtriangledown_{\theta_i} L_i(\theta_i) = \mathbf{E}_{s,a\sim p(.);s'\sim\varepsilon}[(r + \gamma max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)) \bigtriangledown_{\theta_i} Q(s, a; \theta_i)] \tag{12}$$

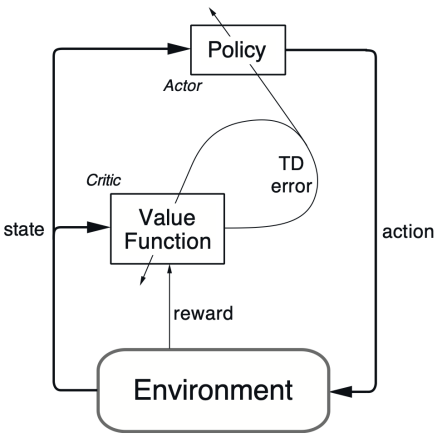
Computing the full expectations in as in the previous formula can prove computationally expensive, so it is often more efficient to optimize the loss function using the stochastic gradient descent. (Mnih et al., 2013).

Being an off-policy algorithm the agent learns about the greedy strategy that maximizes the action-value $a = max_a Q(s, a; \theta)$ while following a behavior distribution that ensures a proper exploration of the state-space. I am going to use an epsilon greedy strategy, selecting a random action with probability epsilon and the greedy one with probability 1-ϵ. The prob-

lem of balancing exploration and exploitation is recurrent in reinforcement learning. This issue is of importance as exploration gives low rewards in the short-term but higher rewards in the long run as more efficient actions are discovered and can be exploited many times whereas exploitation has the opposite effect. There are several mathematical solutions for this issue but they generally make strong assumptions about stationarity and prior knowledge, which are often times violated (Sutton and Barto, 2018). In this thesis I will employ a method called epsilon decay which promotes exploration in the first episodes by setting high values of ϵ and slowly decreasing it in order to exploit the best actions found for each state. I will use linear as well as non-linear decay and observe the difference in performance.

Actor-Critic

The reinforcement learning algorithms that I have previously described are called action-value methods and in the following I am going to present a non action-value method that is instrumental to solving a reinforcement learning problem with a continuous action space. The main difference between the 2 types lies in the fact that the non action-value method does not directly compute action or state values to select actions, instead the policy is represented directly, with weights independent of any value function (Sutton and Barto, 2018). The method that I am going to implement is the actor-critic method. This a temporal difference learning algorithm that has a separate memory structure that is used to represent the policy independent of the value function. The two main components, as the name suggests, are the actor, which selects the actions, and the critic, which criticizes the actions proposed by the actor. This type of learning is generally used in on-policy problems, as the critic has to criticize at all times the policy followed by the actor. The way in which the critic works is that it produces a scalar signal, called TD error. Using this signal the reward for each action of the actor is modified according to the following formula: $y_t[k] = rewards[k] + GAMMA * targetqvalues[k]$, with gamma being a scalar value between 0 and 1, thus modifying the reward associated with a certain action and improving the policy followed by the actor. The architecture of this process is portrayed in the following graph from (Sutton and Barto, 2018):



This method of learning has been very popular at the inception of temporal difference learning (Witten, 1977), but in the recent times algorithms such as Q-learning and Sarsa have been used more often.

Policy Gradient

The actor-critic method is built upon the policy gradient theorem (Sutton and Barto, 2018; Peters and Bagnell, 2011; Bhatnagar, Sutton, Ghavamzadeh and Lee, 2007; Degris, White and Sutton, 2012), which I will implement in this thesis based on the Deep Deterministic Policy Gradient algorithm developed by DeepMind (Lillicrap et al., 2015).

First of all I need to explain what this method consists in and how it is relevant for our setting. It is an approach that directly optimizes a parametrized control policy by gradient descent. Unlike the action-value methods, it maximizes the expected return of a given policy. Some advantages entailed by this method is that they allow a straightforward incorporation of domain knowledge in the policy parametrization and require fewer parameters in order to represent the optimal policy. However, one major drawback is that it is hard to implement such method in an off-policy problem (Peters et al., 2011), something which will become evident throughout this thesis.

The policy used in in policy gradient algorithms can be stochastic, $a \sim \pi_\theta(s)$, or deterministic, $a = \pi_\theta(s)$, which means that the epsilon greedy approach cannot be used anymore in order to promote exploration and instead a disturbance of policy parameters is required.

One way in which an off-policy actor-critic method can be implemented is by using the deterministic policy gradient algorithm (Silver, Lever, Heess, Degris, Wierstra and Riedmiller, 2014). This method uses the actor critic architecture. The critic Q(s,a) is trained

using the Bellman equation the same way as it would be done in Q learning. On the other hand, the actor is trained using an off-policy policy-gradient method. This entails the use of trajectories sampled from a distinct behavior policy $\beta(a|s) \neq \pi_\theta(a|s)$ (Silver et al., 2014). In this type of problem the goal of the actor is to maximize the cumulative discounted reward from the initial state, represented by the performance objective $J(\pi) = \mathbf{E}[r_l^\gamma|\pi]$. The update is performed by applying the chain rule to this equation which yields:

$$\bigtriangledown_{\theta^\mu} \approx \mathbf{E}_{s_t\sim\rho^\beta}[\bigtriangledown_{\theta^\mu} Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t|\theta^\mu)}] = \mathbf{E}_{s_t\sim\rho^\beta}[\bigtriangledown_{\theta^\mu} Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t)} \bigtriangledown_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}] \tag{13}$$

This has been proven by Silver et al., 2014 that it is the gradient of the policys performance, the so-called policy gradient.

2.2 Evolutionary Algorithms

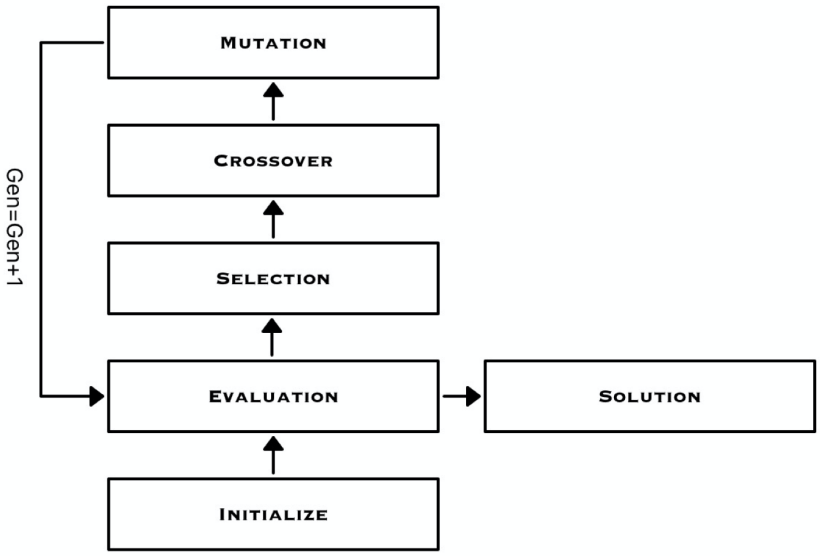
One other way to approach reinforcement learning problems is by using genetic algorithms, which remove the need to perform the estimation of value functions (Sutton and Barto, 2018). The naming comes from the fact that this type of learning is very similar to the way in which the biological evolution endows agents with skills even when they do not perform any type of learning during their lifetime. It relies on spawning multiple non-learning agents with separate policies and select the ones that achieve the highest reward. This type of methods can be especially effective if the number of states is not particularly large or there is plenty of time available to run the algorithms. Evolutionary algorithms are not a new topic, as also pioneers of computer science Alan Turing (Turing, 1952) and John von Neumann (Neumann and Burks, 1966) have proposed ideas linked to Machine Learning and Biological Automation and Mathematics. The reason this subject has come across is related to the exploration-exploitation balance that I have previously mentioned, as scientists considered that exploitative methods have their limitations in solving complex problems and more exploration is required. (Sloss, 2019)

Generally speaking, a value based approach is considered more efficient for reinforcement learning problems (Sutton and Barto, 2018) as evolutionary algorithms ignore some key aspects: the fact that the policy that has to be improved is not considered to be a function from states to actions and no record of states passed or action taken is kept. For the aforementioned reasons evolutionary algorithms are not considered a reinforcement learning method.

There are several key definitions required to understand how genetic algorithms works.

(Sloss, 2019) The evolution process is made out of three elements: replication, variation and selection. Replication refers to how new entities are formed, which can be done either be creating a completely new generation or by altering individuals from the same population. Variation represents the process of diversifying the population, which can be done through recombination (crossover) and mutation. The crossover creates new individuals by combining elements from other individuals, whereas mutation randomizes to a certain extent the genes of an entity by applying stochastic processes to them. Lastly, selection, which is based on the natural selection theory promoted by Charles Darwin (Darwin, 1859). This process consists in selecting only the top performers, or best fits, from a population of individuals by ranking them to create a new generation by applying variation procedures to them.

The digital evolutionary process which is the one I am implementing is represented by the following process: a new generation is created from the top performers of a previous one, then it is evaluated against the goal the species aims to reach and fitness is established and the selection process is applied. Afterwards replications ensue and variation techniques are applied.

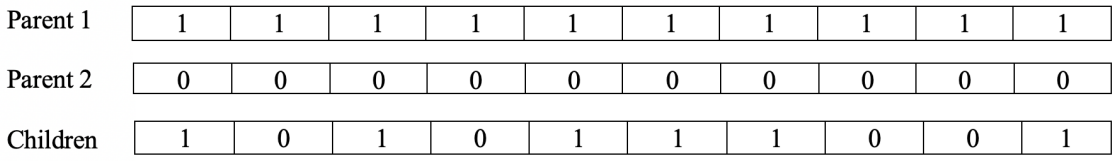


Graph taken from Sloss, 2019

Another important component of an evolutionary problem is the population, which represents the candidates for reaching a solution. Initially, the population can be seeded randomly or from a known solution. It can evolve either by evolving the entire population to create a new generation or only a select few (called Steady-State).

There are many types of evolutionary techniques: Evolutionary Strategies, Genetic Programming, Genetic Improvement etc. but we are going to be concerned only with Genetic Algorithms (Goldberg, 1989). This is the most popular type of evolutionary algorithms and it applies evolution to strings of fixed length, such as the weights of a neural network, representing the parameters used in determining the performance of the algorithm. The length of the strings represents the dimensionality of the problem and the evolution is done by using both crossover and mutation. Generally, this type of algorithm is fitted for problems where the number of variables is too large for the use of traditional methods.

Crossovers are of particular importance for genetic algorithms as they determine the way in which new generations are created. Crossovers can be performed between 2 or more individuals but the implementation I am proposing employs only 2. A first type of crossover used is the Single Point Crossover in which, considering only 2 sets of genes, a point is set on the first set from which onwards the genes of the second set are used to create the children. This type of crossover is the simplest form of N-point crossover which, as the name suggests, is a method in which N crossover points are set and the genes are combined. The method that I am going to use is called Uniform Crossover, which means that genes are selected at random from the 2 set of genes in order to create the child.



Uniform Crossover

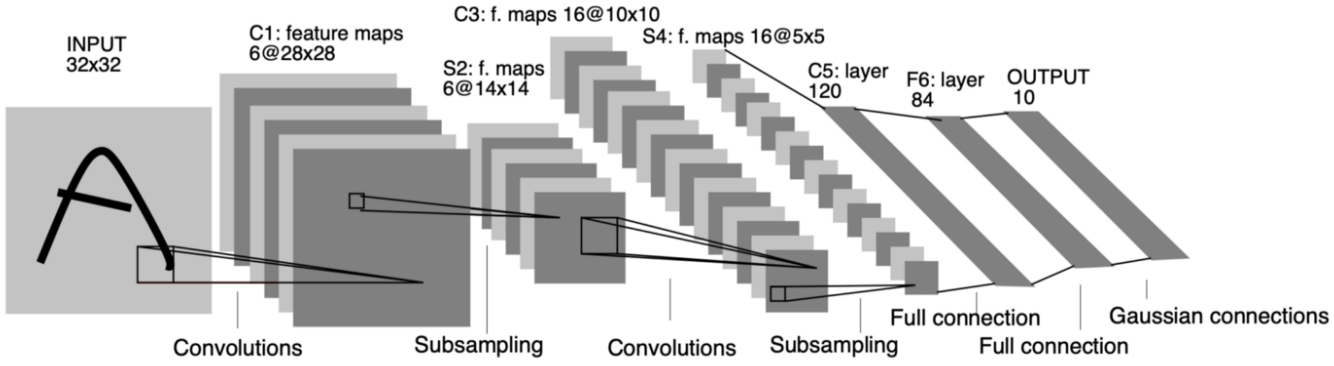
In this thesis I am implementing an evolutionary algorithm that makes use of a Artificial Neural Network. This particular setting is called Deep Neuroevolution (Such, Madhavan, Conti, Lehman, Stanley and Clune, 2018) and is a hybrid method, as it makes use of both Genetic Algorithms and Deep Neural Networks. The purpose of this algorithm is to optimize the weights of the Artificial Neural Networks such that they would achieve the highest reward in our reinforcement learning problem. There are two types of encoding that can be used for this problem, the first one being direct encoding, the approach that I am going to employ. This method implies that in the final solution every weight in the network is going to be included and the architecture of the network stays fixed. On the other hand, indirect encoding implies that not only the connection weights can change but also the topology of the network. One notable example of such an algorithm is Neuroevolution of Augmenting Topologies(NEAT) (Signorelli, Vinciotti and Wit, 2016), which allows the integration of gene

enrichment analysis with the information on gene relationships retrieved from gene networks.

2.3. Convolutional Neural Networks

One other pivotal element of this thesis is represented by the convolutional neural networks, a type of artificial neural network. This type of network has seen extreme improvement in performance as deep learning techniques have developed. Its applications include image and video recognition, image classification, natural language processing, recommender systems etc.

The most relevant capability of CNNs for this thesis is computer vision. The breakthrough paper that has first successfully combined convolutional layers with a Multi-Layer Perceptron (Krizhevsky, Sutskever and Hinton, 2012) has used the ImageNet dataset (15 million labeled high-resolution images divided in 22000 categories) in order to create a classification algorithm that considerably improves on the previous state-of-the-art. This has enabled the use of these deep convolutional networks in, among others, autonomous vehicles in order to better identify their surrounding and coordinate (Fridman, 2018) and also in the use of autonomous robots, either in medical endeavors or industrial applications.



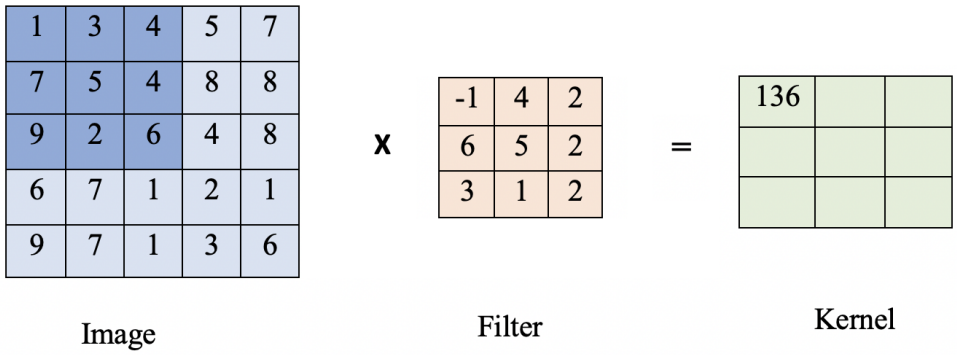
Above is the architecture of LeNet-5, used for digits recognition. Yann LeCun, the creator of this architecture, is one of the pioneers of object recognition through the use of convolutional neural networks, contributing immensely to their evolution, creating the first architecture to recognize hand-written zip codes (LeCun, Bottou, Orr and Mller, 1998) , (LeCun, Boser, Denker, Henderson, Howard, Hubbard and Jackel, 1989), (LeCun, Haffner, Bottou and Bengio, 1999).

Theoretically speaking, convolutional neural networks combine three ideas: local receptive fields, shared weights and spatial sub-sampling (LeCun et al, 1999). An example of such a network is displayed above. The main types of layers are the following: input, convolutional, pooling, fully connected and output. The input layer receives an image in the form of a matrix of pixels which can be multidimensional, depending on the number of channels in the image (e.g. RGB). Generally, images are gray scaled in order reduce the dimensionality of the image and reduce them to only one channel. Then the convolutional layer receives inputs only from a small neighborhood in the previous layer, thus connecting units to local receptive fields. The reason why this method is applied dates back to the 1960s, when Hubel and Wiesel discovered such local connections in the visual system of cats. (Hubel and Wiesel, 1962). By using local receptive fields, neurons can extract visual features like oriented edges, end-points or corners, which are then combined in subsequent layers to extract high-level features. In each convolutional layer units are organized in planes within each all components share the same sets of weights. The output resulted from processing a set of inputs through this plane is called a feature map. The units in each feature map perform the same operation in different parts of the image and in order to create a complete convolutional layer several feature maps are required in order to extract multiple features at each location. The number of feature maps is also called features and the dimension of the neighborhood selected is called the dimension of the kernel/filter or receptive field. The kernel, or filter, is the set of connection weights used by the units in a feature map.

Take for instance the architecture depicted in the graph above. It employs 6 feature maps or planes which consist of 25 inputs which are connected to 5x5 filters/receptive fields. The receptive fields generally overlap as they move from an area of the input to another. The parameter which determines how much the filter moves on the input map is called the stride. A filter with a stride of 2 would skip two columns if applied row-wise and two rows is applied column wise. In the above example a stride of 1 is applied which make the overlapping of receptive fields be of 4 columns and 5 rows for horizontally contiguous units. As it was previously mentioned, the same filter is applied over the whole image in order to determine all possible location where a particular feature exists. All units in the feature map share the same weights and bias. By applying several layers of filters more feature maps are obtained which all extract different features, thus rendering a more precise representation of the input. This operation is called a convolution. One important property of the convolutional layers is that the feature map will be shifted the same amount the input image is shifted, which makes the convolutional networks robust when it comes to shift or distortions of input. (LeCun et

al, 1999)

Figure 1: Convolution Process



Once features are extracted their precise position is not relevant anymore, only an approximate location is required. To some extent, too much precision can actually be harmful (LeCun et al, 1999), this being the reason why reduction of the precision through sub-sampling layers is performed. This type of layer, more commonly known as pooling layer, performs local averaging and sub-sampling, thus reducing the resolution of the feature map and the sensitivity of the output to shifts or distortion. In the LeNet 5 architecture the pooling layer used is represented by a 2 by 2 receptive field that averages it 4 inputs received from the feature map in the previous layers, multiplies them by a trainable coefficient to which a trainable bias is added and then uses a sigmoid function to obtain the output. In this architecture there is no overlapping between contiguous receptive fields, unlike the convolutional layers. One other more common sub-sampling procedure is MaxPooling which, if applied in the mentioned architecture, would only take the maximum value in the respective receptive field. The sub-sampling layers have to match the number of feature maps in the convolutional layer, as each pooling is done for a separate feature map.

After the convolution and sub-sampling procedures are repeated for the desired number of times the output of the last layer is connected to a fully connected layer, which can then be connected to other hidden layers, the latter being the familiar artificial neural network. Therefore the dot product between the input and the weight of the connection is computed to which a bias can be added. As it is the case with MLPs, the weights of the network are updated through backpropagation (Rumelhart, Hinton, Williams, 1986).

The backpropagation algorithm works in the following way: the whole input is passed

through a feedforward on the whole network (an episode), which means that all computations associated to each layer are made for each unit of the input, the result is observed and the error is computed according to the cost function of choice for the last layer and then the error is backpropagated through the network, assigning to each layer an error. In the end, the gradient of the cost function is computed, representing the rate of change of the cost with respect to any weight in the network. Afterwards, the optimization function of choice is applied, stochastic gradient descent and Adamax being 2 popular choices.

3. Algorithms

All the algorithms in this thesis can be found in Radu Burtea (2020) and are either developed from scratch or heavily adapted versions of Luc Prieur (2020) algorithm for the CarRacing environment or Phil Tabor (2020) DDPG algorithm for the BipedalWalker environment. The codes have been written in python 3.7 with TensorFlow-GPU as the deep learning package of choice. All algorithms gave been trained on GPU, using a Nvidia GeForce GTX 750 2GB graphics card. Training has been done locally and varied between 7 and 36 hours of training time. The environments (CarRacing-v0, LunarLanderContinuous-v2) on which I have applied these algorithms are provided through the OpenAI Gym (2020) platform.

3.1. Q Learning

The first algorithm that I have implemented is the classic Q-learning algorithm. This requires utilizing a discrete action space as Q-learning can not tackle a problem with continuous outputs. The reason is that this algorithm attributes a certain Q-value to each action and a future action is selected only if it has the highest Q-value among all possible actions, meaning that this value has to be computed for every action in the action space in order to reach a solution. Clearly, this can not be done with an infinite or considerably large number of choices. This algorithm falls in the category of tabular methods as you can think of a table where each row is an action and each column is a state and their intersection is the maximum Q-value reached in that state for that action. When the number of states is very large, as it is the case in our environments, this method is infeasible as it runs in the so-called curse of dimensionality. Some other major drawback of this method is that when it is combined with off-policy problems (Leemon, 1995) the Q-network tends to diverge (Mnih

et al., 2013).

The way in which I have mitigated the explorations vs exploitation problem is by employing an epsilon-greedy approach (Sutton and Barto, 2018). This means that before selecting an action a random number is sampled and if it is greater than the epsilon that was set (by the user) a greedy action is taken, meaning that the action a with the highest Q-value in state s is selected, and otherwise a stochastic action is used, by sampling randomly for the available action space. I have implemented the method that was proposed in (Sutton and Barto, 2018), based on the following pseudocode from the book:

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

This algorithm runs for the set number of episodes desired or until convergence is achieved.

3.2. Deep Q-Learning

Deep reinforcement learning has emerged once deep neural networks successfully processed raw inputs using stochastic gradient descent and perform better than when handcrafted features are used as inputs (Mnih et al., 2013). These major improvements have determined researchers from DeepMind, a subsidiary of Google that works towards advancing the state of the art in artificial intelligence, to combine the aforementioned Q-learning algorithm with a deep neural network that uses only raw RGB images as inputs.

However, Deep Q-Learning is not the first reinforcement learning algorithm to be coupled with neural network. This honor goes to Tesauros TD-gammon, a program that learned to play backgammon by using reinforcement learning and self-play, achieving a super-human performance (Tesauro, 1995). This algorithm is also model-free and used a multi-layer perceptron with one hidden layer to approximate the value function. Upon TD-gammons success there have been attempts to apply similar algorithms to play chess, Go or other games, most of them unsuccessful, which deflated the excitement for this type of method for a while.

The DQN algorithm that I have implemented follows closely the approach suggested by Mnih et al, 2013. It starts from building up on the method employed in TD-Gammon. The

algorithm updates the parameters of a network that is used to estimate the value function for each action from the considered action-space, using on-policy samples of experience (s,a,r) drawn from the interactions of the algorithm with the environment.

Instead of using a standard online approach, a technique known as experience replay is used (Lin, 1993). Agents experience are stored in a buffer of dimension 2000, where each experience is represented by the state at time step t, the action taken, the reward obtained and the state at time $t+1, e_t = (s_t, a_t, r_t, s_{t+1})$. This buffer is called replay memory, $M = e_1, ..., e_N$, with N being the dimension of the replay memory. At each iteration of the algorithm a mini-batch is sampled randomly, $e \sim M$, and Q-learning updates are applied for each experience in the sample (batch) and the neural network is fitted with the new values. After the mini-batch (due to the reduced number of samples, generally 32 or 64) update is completed an action is selected for the current state using the aforementioned epsilon greedy method. In order to make the implementation of this algorithm easier the inputs of the network will be of the same dimension and preprocessed by a function that we will call φ. A simplified depiction of the algorithm can be found below:

```
observation = env.reset()
while not done:
    state = preprocess(observation)
    action = model.epsilon_greedy_action(state, epsilon)
    observation, reward, done, info = env.step(action)
    new_state = preprocess(observation)
    model.remember(state, action, reward, new_state, done)
    model.replay()
    state = new_state
```

The update on the minibatch is performed as in the following:

```
samples = random.sample(memory, batch_size)
for sample in samples:
    state, action, reward, new_state, done = sample
    target = model.predict(new_state)
    if done:
        target[np.argmax(target)] = reward
    else:
        target[np.argmax(target)] = reward + np.max(target) * GAMMA
    model.fit(state, target)
```

The first advantage that this method has over the standard Q-learning algorithm is that each experience can be used multiple times to update the network, thus improving data efficiency. Secondly, by using the replay buffer one major issue is taken away, that being the fact that consecutive samples are highly correlated and randomizing them reduces the variance of the updates. One other issue with traditional Q-learning is that when learning on-policy the future action on which the network will be trained is determined by the current parameters, therefore if steering left is the maximizing action the training distribution will be mostly represented by actions that suggest steering left, thus creating a training loop. In

my experience, even with a replay buffer the algorithm can be stuck in a loop, this being primarily caused by the discretization function used to represent the action space.

I have used several discretization functions for the CarRacing-v0 environment. A first approach was to list a set of actions that would represent the action space. One important aspect to look out for when discretizing is to not over-sample a certain action, such as steering left or right, as the car would always end up in loops. This method of discretization did not work out as the car would always end up running in circles. Therefore I have used a discretization function inspired by Luc Prieur which rendered good results, about which I will talk in the Experiments section. As for the value of epsilon I have used several approaches: fixed epsilon, linear decay and non-linear decay.

I have also developed 2 separate models for each type of input, one for handcrafted features and one for raw images. The preprocessing for the handcrafted features has also been heavily inspired by the one Luc Prieur performed, using the information provided by the environment (gyroscope, abs, steering wheel position) to determine the most relevant parameters for each state: gyroscope, speed, steering and 4 different values for the abs by cropping the part of the images relevant for each of these values. On the other hand, for the algorithm using CNNs I have preprocessed each frame by cropping it to 84x84 pixels, as CNNs require square inputs, and reduced the dimension of the image to 1 by gray-scaling it. I have also tried using 4 stacked images as inputs but I have not observed improvements so I conducted most of my experiments using only one image as it is lighter computationally and from a memory standpoint.

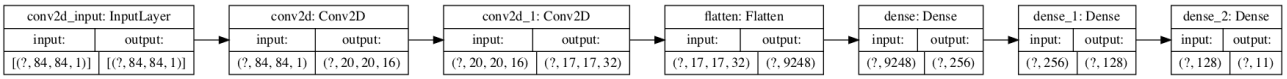
Below you can find the architecture used for the model which uses convolutional neural networks. This architecture is similar to the one proposed in DeepMinds paper (Mnih et al., 2013), adapted to the requirements of the discretization function that I use. The network takes as inputs an 84x84 frame, applies the first convolutional layer which consists in 16 filters with 8x8 receptive fields and a stride of 4, followed by another convolutional layer with 32 filters, 4x4 receptive fields with a stride of 1. Both layers use the rectified linear uniform (max(0,x)) activation function.

The following layers are fully connected. The first hidden layer consists of 256 neurons using a LeCun uniform initialization function (LeCun et al, 1998) and ReLU as an activation function. The final layer consists of 11 neurons using the same initialization function as the previous layer and a linear activation function, as the discretization function that I use requires the outputs to take values between 0 and 10. The error function is the Mean Squared Error and the optimization function is AdaMax, a variant of the Adaptive Moment

Estimation method, using instead of the second moment of the gradients in the update rule the maximum between the second moment times a constant beta and the absolute value of the current gradient,

$$u_t = \beta_2^{\infty} v_{t-1} + (1 - \beta_2^{\infty})|g_t|^{\infty} = max(\beta_2 v_{t-1}, |g_t|)$$

(Ruder, 2016) The learning rate is fixed at 0.001.



The architecture for the network which takes as inputs handcrafted features is quite similar and it can be found below. The only difference is that there are no more convolutional layers. The first layers is a fully connected layer with 512 neurons that takes as inputs the outputs of the preprocessing and additionally the pixel representation of the car and the field immediately in front of it. The activation and initialization are the same as in the fully connected layers of the previous network.

3.3. Double Deep Q-Learning

Double Q-learning (van Hasselt et al., 2010) has appeared as a solution to the overoptimistic value estimates that Q-learning was outputting due to the fact that it was using the same weights to evaluate and select actions. The solution that Double Q-learning proposes consists in decoupling the selection and evaluation process. The overestimation issue of Q-learning has first been discussed by Thrun and Schwartz (1993), who showed that when actions contain errors uniformly distributed on $[-\epsilon, \epsilon]$ then targets are each overestimated by $\gamma\epsilon\frac{m-1}{m+1}$, with m being the number of actions.

The original method proposes two sets of value functions that are randomly assigned experiences to be trained on which renders two sets of distinct weights for the value functions, θ and $\theta^{'}$. In this architecture, for each update one value function is used to determine the greedy policy and the other to determine the value. When comparing the Q-learning and the Double Q-learning algorithm we can look at how the errors are written for each algorithm (Hasselt et al., 2010):

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t); \theta_t) \tag{14}$$

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t); \theta_t') \tag{15}$$

We can observe that the selection of the action is performed in the same way as in the Q-learning algorithm, estimating the values of the greedy policy using the weights θ_t but at the same time having another set of weights $\theta_{t'}$ to evaluate the values of the policy.

Therefore, Double Deep Q-learning (DDQN) (Hasselt et al, 2016) entails using 2 sets of networks, of which one is the target network which selects the actions and the other evaluates the action. This type of architecture might sound familiar to the actor-critic method previously presented but it is quite different, as the actor and critic networks are architecturally fundamentally different and the outputs of the critic network represent the basis on which the gradient ascent update is the performed on the actor.

In DDQN the 2 networks are not fully decoupled, the target network representing a candidate for the second value function. Hasselt et al., 2016 propose an evaluation of the greedy policy according to the online network and using the target network to estimate its value. The update for DDQN differs slightly from the one of Double Q-learning:

$$Y_t^{DoubleDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t), \theta_t^-) \tag{16}$$

as θ' is replaced by the weights of the target network θ^- in order to evaluate the current greedy policy.

The way in which the algorithm that I have implemented differs from the one used by Hasselt is that for the target networks I am using soft updates, as opposed to copying the weights. The way in which these weights are updated is as in the following: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$. By implementing this update the target values are changing slowly, increasing the stability of learning (Lillicrap et al, 2015).

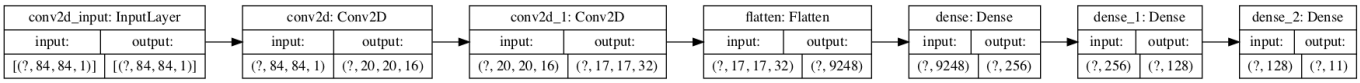
The implementation of the soft updates can be found below:

```
def target_train():
    weights = model.get_weights()
    target_weigths = target_model.get_weigths()
    for i in range(len(target_weigths)):
        target_weigths[i] = target_weigths[i]*tau + (1-tau)*weights[i]
    target_model.set_weights(target_weigths)
```

As for the updated algorithm to accommodate the change in network structure we have the following piece of code representing the updates performed on the mini-batch:

```
def replay_buffer():
    batch_size = 8
    if len(memory) < batch_size:
        return
    samples = random.sample(memory, batch_size)
    for sample in samples:
        state, action, arg, reward, new_state, done = sample
        target = target_model.predict(state)
        if done:
            target[0][arg] = reward
        else:
            Q_future = max(target_model.predict(new_state))
            target[0][arg] = reward + Q_future*gamma
        model.fit(state, target, epochs = 1)
```

As it can be observed the actions to be taken are provided by the target network. In terms of architecture the DDQN algorithm is also slightly different, adding another hidden layer.



3.4. Deep Deterministic Policy Gradients

I have included this algorithm in this thesis as it is the most straightforward approach to a model-free, off-policy reinforcement learning problem with a continuous action-space. This algorithm is developed by Lillicrap, 2016 and modifies the DPG algorithm proposed by Silver in a similar fashion to how DQN improved on traditional Q-learning, allowing a neural network to learn online in an environment with large state and action spaces. The fact that most optimization functions for neural networks assume independent and identically distributed samples means that also in this algorithm a replay buffer is required in order to remove the correlations between the samples.

This algorithm is built on the actor-critic architecture that I have mentioned in section 2, while also using target networks as in the DDQN algorithm and employing soft target updates. The actor network will be denoted $Q(s, a|\theta^Q)$ and the critic $\mu(s|\theta^\mu)$. A copy of the actor and the critic networks are created, $Q^{'}(s, a|\theta^{Q^{'}})$, $\mu(s|\theta^{\mu^{\theta}})$, which are used to compute the target values. Instead of the traditional online learning, when performing minibatch updates for both of them the main network is updated only at when the whole batch is processed, so that the network will become less prone to divergence. One further addition that improves the performance of learning on mini-batches is the batch normalization (Ioffe

and Szegedy, 2015). By using this method each dimension across the samples is normalized, imposing unit mean and variance. Furthermore, a running average of the mean and variance is kept in order to perform normalization during training.

When using a continuous action space not only the way in which actions are chosen is different from the discrete case, but also the way in which the algorithms tackles exploration. One main difference is that epsilon greedy actions can no longer be taken, as the update in DDPG is performed on-policy and sampling random actions from a different behavior distribution will not contribute to the improvement of the current policy that is being updated. The way in which exploration policy $\mu^{'}$ is designed is inspired by Lillicrap et al, 2015 and the actions proposed by the actor policy are modified is by adding some noise sampled from a process $\aleph$:

$$\mu^{'}(s_t) = \mu(s_t|\theta_t^{\mu}) + \aleph \tag{17}$$

The noise processed used is the Ornstein-Uhlenbeck process (Uhlenbeck and Ornstein, 1930). It generates temporally correlated noise used for an efficient exploration process in problems where physical control with inertia is used, which is the precise case of the CarRacing environment used by me, as the simulation is performed in a physical engine and the car maintains its speed if a braking action is not taken. The overall algorithm is the one proposed by Lillicrap et al, 2015 and the pseudocode provided in the paper can be found below:

Algorithm 1 DDPG algorithm

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

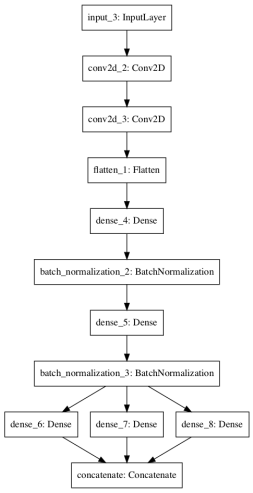
$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

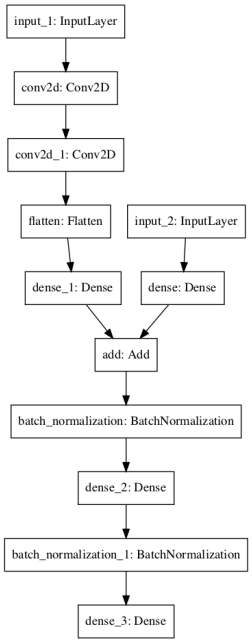
    Update the target networks:

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

  end for
end for

The architectures of the actor and respectively the critic can be found below:





The critic network takes 2 inputs which require different types of networks: a state and an action taken by the actor. The convolutional layers are very similar to the ones found in DQN or DDQN. There are 2 convolutional layers, one with 32 and other with 16 filters, both with kernels of size 8x8 and the first one has a stride of 3 while the other 2. On top of that a fully connected layer with 400 units is applied which uses ReLU as an activation function. Actions selected by the actor represent the other input of the network. The inputs are connected to a hidden layer with 400 neurons and a linear activation function. Following that the two sets of inputs are merged by adding the outputs of the hidden layers and then a fully connected layer with 200 neurons and ReLU activation is applied, followed by the output layer with one neuron and a linear activation function.

The actor network is simpler, using as inputs the pixel representation of one preprocessed image, followed by 2 convolutional layers with 16 and 32 filters, both with receptive fields of size 8x8 and ReLU activation functions. Following that two hidden layers are added, with 400 and 200 neurons, both of them using ReLU as an activation function. From this point on there are three separate output layers for each component of the action. Actions are of the form [steering, acceleration, brake], with steering taking values in [-1, 1] and the other two in [0,1]. Therefore, for the steering the hyperbolic tangent activation function is used (tanh(x)) and for the other the sigmoid ($\frac{1}{1+e^{-x}}$). Finally, the 3 values are concatenated. After each fully connected layer batch normalization is applied.

3.5. Evolutionary Algorithm

As it is the case also for deep learning, evolutionary algorithms have benefited from renewed interest as the computational power has dramatically increased in the past few years. As I have previously mentioned, the approach I am going to use is called neuroevolution. At the moment, the research performed on neural network is generally focused on deep learning and deep reinforcement learning which use backpropagation to compute the loss functions gradients and an optimization function to update the weights in order to reduce the loss. Evolutionary algorithms come as an alternative to the backpropagation-optimization function combo, as the optimization is done through evolution. Even though at first researchers focused only on applying evolutionary algorithms on fixed topologies, an approach I am also taking in this thesis, most of the recent development is focused on using evolutionary algorithms to also improve the architecture of the neural networks (Stanley, Clune and Lehman, 2019). One domain in which evolutionary algorithms have been implemented successfully is robotics, where the first running gait was produced for the Sony Aibo robot (Hornby, Takamura, Yokono, Hanagata, Yamamoto and Fujita, 2000) and other in modifying the morphologies of 3D printed robots that were capable of movement in a real environment (Lipson and Pollack, 2010). Also reinforcement learning is used extensively when aiming to produce autonomous robots and vehicles (Abbeel et al, 2007),(Benbrahim and Franklin, 1997) so it comes naturally that whenever a problem can be tackled with reinforcement learning also an evolutionary algorithm should be tested on it.

A simplified version of the main code developed from scratch can be found below.

```
def play(model, weights, epsilon, species, share_a):
    models, rewards = [], []
    iter = 1
    for i in range(species):
        observation = env.reset()
        weights_new = mutation(weights, weights_index, shapes)
        model_gen = model.copy()
        model_gen.model.set_weights(weights_new)
        models.append(weights_new)
        totalreward = 0
        iter = 1
        done = False
        while not done:
            state = transform(observation)
            action = model_gen.act(state, epsilon)
            observation, reward, done, info = env.step(action)
            new_state = transform(observation)
            totalreward+=reward
            state = new_state
        rewards.append(totalreward)
    weighted_average, weights_final = crossover(rewards, models, share_a)
    return weights_final, weighted_average
```

The way in which I have designed this algorithm is as following: a range of artificial neural networks are generated using the weights of the parent, then for each individual a mutation is applied. I use a fixed mutation rate and generate a random number between 0 and 1 for each weight in the network. If the number is larger than the rate nothing changes and if its smaller I replace the weight with a number randomly sampled. Mutation can be viewed as an exploration method in evolutionary algorithms and a mutation rate is equivalent to the epsilon in an epsilon greedy policy. After that each individual goes through a full episode and the rewards and weights used are stored in lists. The crossover procedure is then applied. The crossover is ranking based, selecting the top 2 performers in terms of reward and then applying a uniform crossover. I have selected a variable named $share_a$ that would indicate the proportion in which the weights are taken form the top performer. A random number [0,1] is sampled and if it is greater than $share_a$ the weight of the parent network is taken from the second best performer and otherwise from the first best. The weights of the new parent network are returned by running the play function once and used as an input for the following generation. The inputs for this algorithm are handcrafted features and the architecture of the network is the one used for DQN with the same type of inputs. Actions are discretized in this algorithm as well, even though continuous actions could also be used by using 3 outputs for the neural network from which the steering had a hyperbolic tangential activation function and the braking and acceleration sigmoid activation functions.

4. Experiments

In order to determine which algorithm tackles reinforcement learning problems with continuous action spaces the best I have devised several experiments. I will perform comparisons between algorithms, but also within algorithms. For each algorithm I will explore their performance with changed parameters, inputs and methods, such as exploration rate and decay, types of inputs, rate of update, gamma values, replay buffers etc. I will also use other environments, albeit simpler, to see whether the algorithms perform good just in the CarRacing environment or also when faced with other challenges. In terms of inputs three approaches are used: feeding one image to the neural network, using handcrafted features and using 4 stacked images on order to obtain a better sense of direction for the way in which the car is going. I will start with the experiments specific for each algorithm and then compare the best performers.

4.1. Q Learning

As it would have been expected, tabular methods do not fare well when applied in problems with large state spaces. Considering that each slight movement of the car results in a new state and therefore almost every step taken by the agent results in a new state the state-value pairs go very quickly in the tens of thousands (an episode lasts for 1600 time steps) which makes the algorithm memory and computationally heavy. The experiments run for this algorithm have not been finalized, as even after 50 episodes (approximately 80 000 state-value pairs) progression was very slow and not much improvement was observed, rewards r belonging to [-80, -140]. This bottleneck is called the curse of dimensionality in the literature, which makes such algorithms infeasible in environments with large action and state spaces.

Therefore, Q-learning can not be applied in this environment, but represents the basis for algorithms that tackle the same problem really well.

4.2. Double Deep Q Learning

For DDQN I have performed a series of experiments, using as inputs images as well as handcrafted features, modifying the epsilon decay function and using several values for the parameters that determine the soft updates. From all the tests that I have run, DDQN

has performed the best in terms of convergence and stability.

One of the first issues I have run into was the discretization of the action space such that the algorithm would not diverge or get stuck in a loop. At first I have devised a predefined set of actions the algorithm could choose from, such as [[-0.1, 0.1, 0.2], [-0.2, 0.4, 0],]. After using one action each for braking, accelerating, steering left and right, I have observed that the agent, after 200 episodes, will simply choose to go straight ahead. In order to promote a more diverse behavior, when choosing a random action under the epsilon greedy policy, I have attributed higher weights to the steering actions, making them more likely to be chosen. Unsurprisingly, this has led to the agent being stuck in loops taking left or right. By adding more actions to the action space nothing has changed, as this has had a similar effect to increasing the weights for some actions. Therefore, I settled using the discretization function used by Luc Prieur. Using this method the Q-network outputs 11 elements with values ranging from [0, 10] and the maximum value form this array of outputs is chosen. Then, based on the value of the maximum output certain values are attributed to steering, gas or brake, as it can be observed below:

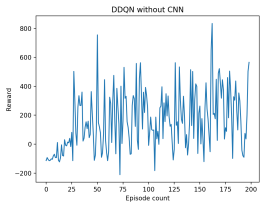
```
if output_value <= 8:
    output_value -= 4
    steering = float(output_value)/4
elif output_value >=9 and output_value <=9:
    output_value -= 8
    gas = float(output_value)/3
elif output_value >= 10 and output_value <= 10:
    output_value -= 9
    brake = float(output_value)/2
```

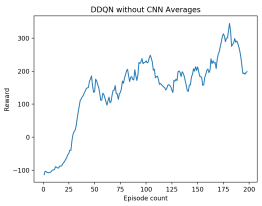
I will first run the algorithm 3 times, with the same epsilon decay function and parameters for soft updates, the only thing that will be different being the inputs, using the three types of inputs I have previously mentioned.

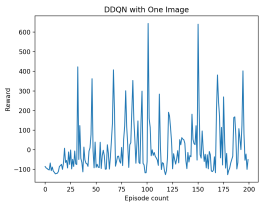
In the first experiment that I run with a DDQN network I have used handcrafted features as inputs, therefore using the second architecture that I presented in 3.2 Q-learning, a τ value for the soft update of 0.8 and the following decay function for epsilon:

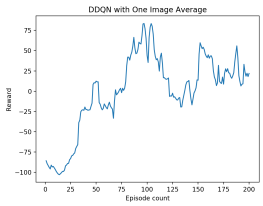
```
if n <= 10:
    eps-=0.05
else:
    eps = max(0.05, 1/np.sqrt(n))
```

Where n is the number of episodes. By using this function I aimed to promote exploration in the early stages and then use a nonlinear function later in the episodes. This first went surprisingly well, achieving decent results even after 40 episodes.







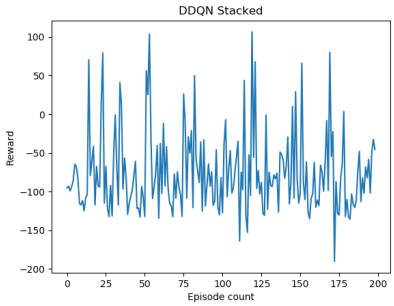


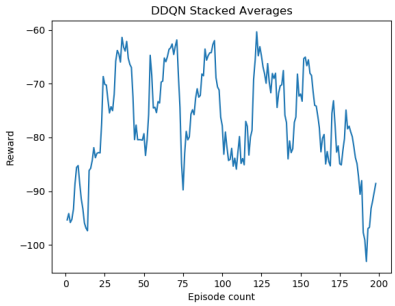
As it can be observed, the algorithm almost solves the environment once at around the 160 episodes mark and reaches some very good scores early on in the training. However, this algorithm proves to be highly unstable, going from rewards of 800 to -100 in 2 episodes. This is mainly due to the fact that the circuit on which the car is spawned is quite large and there is no full map knowledge. So if the car trains very well in a portion of the map and then is spawned in an unexplored corner it is more than expected for it to perform poorly. As it can be seen in the average computation, which takes the average of the last 20 episodes, a increasing trend can be observed. The training took about 5 hours and the average of the last 100 episodes is 204.3.

The next experiment performed uses the same parameters but takes as an input one image. The results can be observed in the above graphs.

This algorithm performs worse than the one with handcrafted features, but it was to be expected as it is harder to get a sense of the direction and speed of a car only from one image. A bigger learning curve can be observed for this algorithm, as the one which used handcrafted features required only about 3-4 episodes to recover from being faced with a new part of the map, whereas this one requires about 10-12 episodes or even more. Graphically, this can be observed by looking at the distance between the peaks. For the aforementioned reasons this algorithm is more unstable and its increasing trend is not as accentuated as for the former. The mean reward of the last 100 episodes is 23.248. The training for this algorithm took about 8 hours.

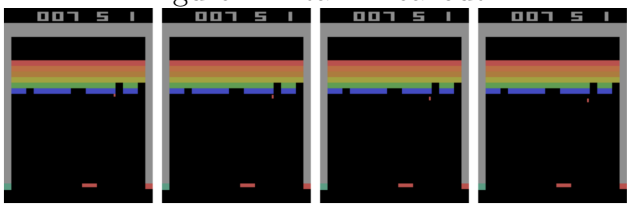
Another input that I have used is the 4 last frames observed in the environment stacked. According to Mnih et al. (2013). this should have provided a better sense of the direction and speed of the car.





As it can be easily observed, this was not the case. This method worked the worst of them all, having peaks that reach only 100 points of reward. Also, the average reward is the lowest from any experiment, reaching only -77.23. It seems that stacking images is not beneficial for this environment, offering unnecessary information that output features that are not contributing to a better understanding of the environment and the behavior of the agent. However, Mnih et al. (2013) have applied this algorithm in different environments, more specifically Atari 2600 games, where it is next to impossible to determine the direction in which the agent was going, such as in Breakout, where in a single frame there is only a plain bar and the ball appears to be a stationary object. In the CarRacing environment on the other hand, the car has wheels which point the direction in which it was going and also braking marks are present if the car braked too hard, providing sufficient information to determine the inertia of the car.

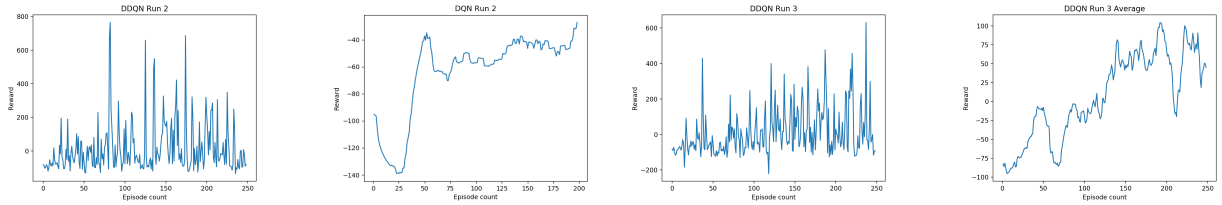
Figure 2: Atari Breakout



Even though the algorithm that used only one image as an input did not perform as well as the one with the handcrafted features, I am going to be using it as a benchmark to observe how performance varies as parameters change. There are several reasons why I am proceeding this way. First of all, the algorithm that uses 4 images has a very long training time (24 hours) and I would also like to use raw images as inputs, in the spirit of the paper written by Mnih et al. (2013). On top of that, the development of autonomous cars and robotics in general will use a form of computer vision, as it better generalizes and extracts features than a human. Another reason is that in this algorithm there is a large room for

improvement as it performed poorly initially. The following algorithms have been run for 250 episodes and all of them took about 8 hours to train and each batch of samples for the replay experience consisted in 8 samples, due to computational restrictions.

The second run with the algorithm that took a single image as input used a linear decay function for epsilon, decreasing it by 2% at every iteration, using the following formula: eps = max(0.05, eps-0.02), and stop decreasing at 5%.

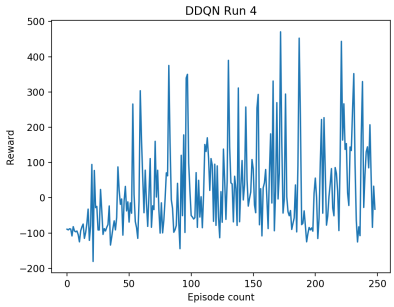


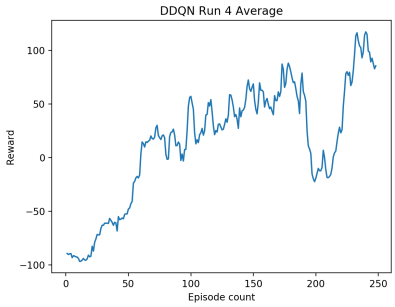
It can be observed that this algorithm peaks higher, reaching rewards of 800 almost 3 times. It also seems that in the initial 75 episodes the faster decreasing exploration rate used in the first algorithm fares better, reaching a mean value of -31, whereas the linear decay reaches -37. The increase in average reward follows a similar pattern to the previous algorithm. However, the median reward of the last 100 episodes is higher by 9 points, reaching 32.79.

The third run uses a decay function similar to the third run, but the decay is accentuated, using instead of np.sqrt(n) np.sqrt(n+10).

Even though this was not a major change it appears that the more aggressive decay function has benefited the algorithm, rendering the best result of the 3. Even though it does not reach the peaks the others did, this algorithm seems more stable and reliable, having a mean value of the last 100 episodes of 56.34, an increase of over 24 points over the former best performer. Also the average increase seem to be the most consistent of all three, the only major depression being at the around 210 episodes mark. The question now becomes which is more important, consistency or peaks. This can be found out running the algorithms for longer, say 700 episodes, which I could not do due to the computer that I was using timing out after about 13 hours when using the GPU intensely.

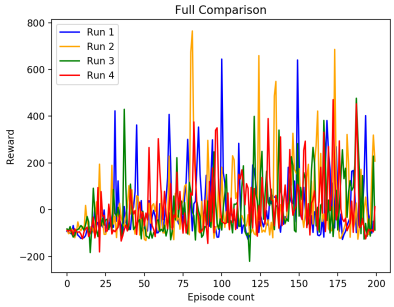
The fourth run builds op on the last, as it was the best performer. Now I am looking to find out whether a more aggressive update of the weights is beneficial. As a reminder the soft updates are performed in the following way: $\theta^{'} \leftarrow \tau\theta + (1-\tau)\theta^{'}$. Until now θ was 0.2, now I will set it to 0.1.

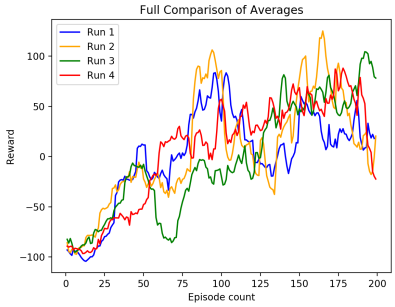




The performance is similar, no notable difference being found. One thing to notice is that both algorithms have depressions at the 200 episodes mark, this being due to the fact that the starting point of the car is repositioned. The mean reward for the last 100 episodes is 52.04, a slight decrease from the previous version.

A full comparison of the four algorithms can be found below:





Graphically it is hard to determine the best performer but if we consider the average reward Run 3 is the winner.

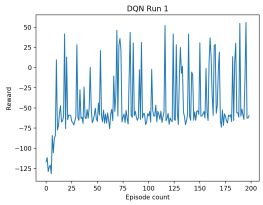
4.3. Deep Q-Learning

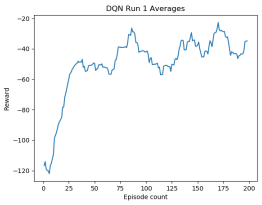
The Deep Q-Learning has not proven to be as performant as I expected, rendering results that were less than satisfactory. The training time for each algorithm took about 7 hours and used 200 episodes.

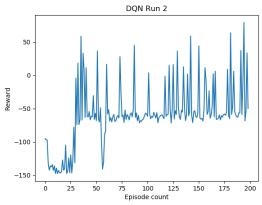
In some of the experiments that I ran with this algorithm I have used a replay buffer as it was suggested in the paper by Mnih et al. (2013) so that I could obtain a smoother training and get rid of the correlations between the subsequent states. This might me due to the overestimation of actions that Q learning is prone to.

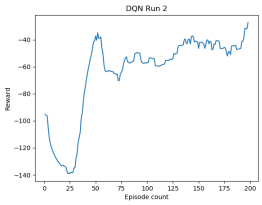
In the first run of this algorithm I have used a single image as an input, batch size of 8,

a γ (the constant that determines how much Q-values are changing each episode) of 0.8, no replay buffer and the following epsilon decay function:



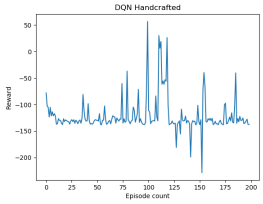


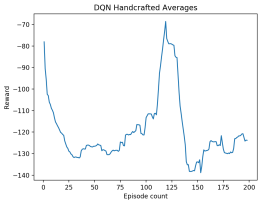


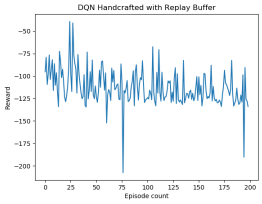


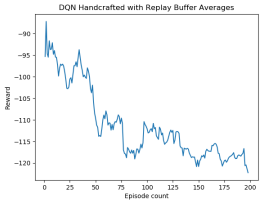
The agent has reached several times values approaching 50 and the algorithm has proven once again very unstable with results lower considerably worse than DDQN. The average of the last 100 episodes is -39.885. For the second run I have used the same decay function but a different value for gamma, 0.99. A similar pattern to the first run can be observed with a slightly worse mean reward of only -43.26.

After having observed the results using images as inputs I have run the experiments the third time with the same parameters as in the first experiment using handcrafted features as inputs. I have also run the algorithm a fourth time, using also a replay buffer with a batch size of 8.







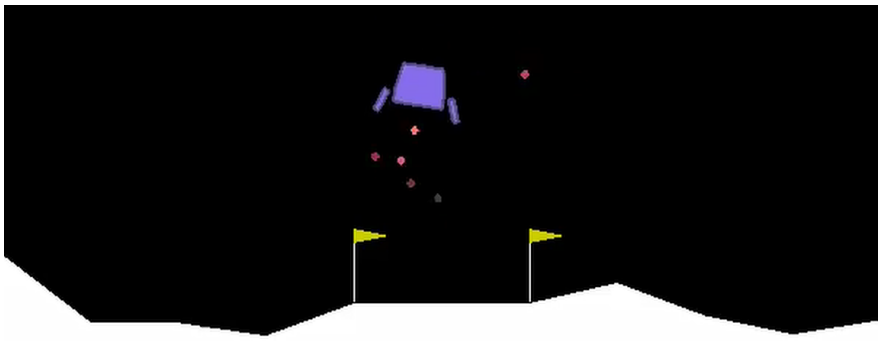


Surprisingly, using handcrafted features has significantly worsened the performance of the algorithm, especially when considering that the Double Deep Q Learning algorithm has produced far superior results when using handcrafted features. The agent reaches only once values over 50 and very seldom positive values. There is also a slight increase in average values from the 25^{th} episode mark up to 100, but from thereafter rewards decrease significantly. The preprocessing techniques are the same as the ones in the DDQN algorithm and the discrepancies between the performances of the 2 algorithms is quite strange. The mean value of the last 100 episodes is -181.72, which shows that this type of input is not informative enough when using online learning. I have run several experiments with these parameters in order to make sure that the result is indicative of the algorithm's actual performance. The results when using the replay buffer worsen, as there is no episode where the reward is higher than 0. Even though in the paper presented by Mnih et al. (2013) it is stated

that a replay buffer improves performance by diminishing the effect of serial correlation of consecutive states, it proves that in this environment this technique has the opposite effect. Some further investigation is required in order to determine why this is the case.

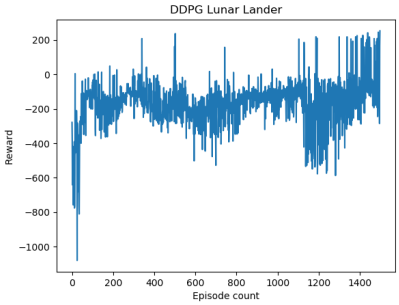
4.4. Deep Deterministic Policy Gradient

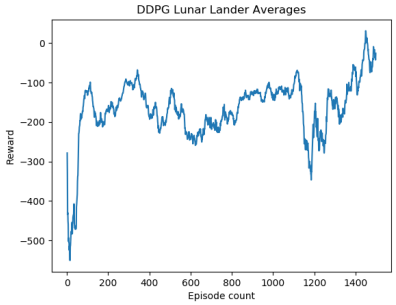
Using the algorithm that I have initially developed (a heavily adapted version of the code that can be found in Lau (2020)) I could not obtain any improvement of the policy in the CarRacing environment, the car being stuck running in circles. In order to test whether the algorithmic approach was sound I applied the same algorithm in a simpler environment provided by OpenAI gym called LunarLanderContinuous-v2. This an environment with 2 continuous actions with values ranging in [-1,1] and the goal of the agent is to land safely on a landing pad located at the (0,0) coordinates at all times. Unlike the CarRacing environment the reward function is linear only up to a certain point, as landing on the pad grants an additional 100 points and anywhere outside of it the reward decreases by 100. The fuel of the agent, which is a spaceship, is infinite, meaning that it can fly for as long as it does not crash. One other difference between the 2 environments is that the states are represented by vectors of length 8, indicating the position, velocity and angle of the ship in the LunarLander environment.



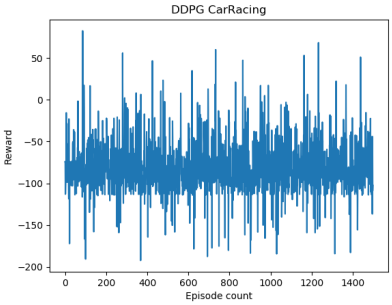
However, even when using the latter environment the algorithm did not seem to improve. The loss would decrease but only up to a certain point until which it reset. One other potential issue would be how the gradients are computed using keras in tensorflow 2.0. The algorithmic approach that I have implemented was sound but the optimization process would not work as expected. Therefore I have decided to take a DDPG framework developed in TensorFlow 1.4 and retry the experiments. I have used the framework provided by Tabor (2020) and adapted it for the LunarLander environment. The architecture of the network

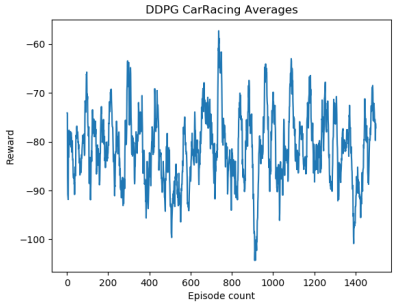
stays the same as in the first model, the only difference being the number of outputs and the fact that both outputs have a tangential activation function, thus accommodating the fact that the output ranges from -1 to 1.





The algorithm performed pretty well, considering that after 1400 episodes it solved the environment constantly. This proves that the algorithm does indeed work, improving the policy constantly over the episodes. One thing to observe is that this algorithm is not particularly sample efficient, requiring a large number of episodes before it solves the environment. Now, moving on to the CarRacing environment we observe the following results:

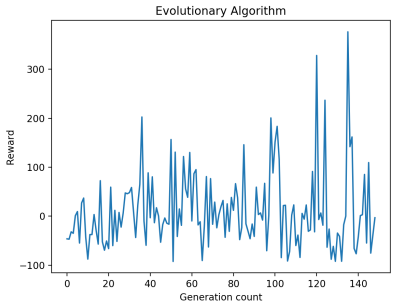


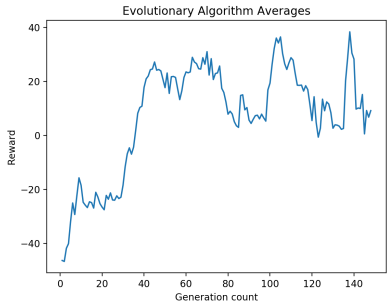


There is no real improvement in the policy over time when applying the same algorithm on this environment, the algorithm reaching very seldom positive values, averaging a reward of -81.63 over the last 300 episodes. In order to make sure that the issue with this algorithm is not that it has a slow convergence I have run the same experiment for 5000 episodes achieving similar results. The lack of improvement may be due to the way in which the reward function is built, considering that in the CarRacing environment there is no extra/deducted points for reaching/missing a particular destination, unlike the other environment, where the states associated to the (0,0) position are completely overshadowing the other states, having the value 100 and the ones when the lander crashes outside the pad of -100. One other aspect to consider is that the state-space is considerably smaller and less diverse in the LunarLander environment, allowing for devising a policy that has to generalize less.

4.5. Evolutionary Algorithm

Training deep neuroevolutionary algorithms has proven trickier to train than I have initially anticipated. There are several factors which come into play when creating an evolutionary algorithm. From my experience, the most important is the mutation rate. Initially I have set it at 10%, which rendered an algorithm that was not creating any usable agent. The car would simply go straight. Having seen that in the literature most of the algorithms use low mutation rates I have also decreased them, which only worsened my results. Mutation rates this low might be efficient in the long run, but I was not able to run the simulations for too long due to computational restrictions and decided to increase the rate to 40%. Another very important aspect in training my network was also the value range of the weights. Initially, when performing mutations I was replacing weights with values randomly sampled from np.random.random(), which samples values no smaller than 0.1. It turns out that this was a major issue because the weights in the network were intialized using a LeCun Uniform function, taking values in (-0.1, 0.1), and replacing weights with values that large would render the network useless, as the network would output higher and higher values, which, given the way in which the discretization function is designed, translate to full acceleration. Therefore I accommodated this issue by replacing the values with np.random.random()*np.random.choice([-1,1])/10. In the final algorithm I have used a population of 12 individuals, a crossover rate of 80% and a mutation rate of 40%. The results can be found below:





The results obtained using this algorithm are better than the ones obtained when using Deep Q Networks and even some versions of the DDQN algorithm. The average reward is 27.05 and an overall increasing trend can be observed, although it fluctuates quite heavily around the 90 and 130 episodes mark. However, this algorithm is sample inefficient, requiring 1800 episodes to achieve rewards that DDQN would achieve in 150. Training time was

also quite long, about 36 hours, but such an algorithm is not computationally heavy, the optimization function achieving O(n) complexity, with n being the number of weights in the network.

5. Conclusions

After performing all the experiments there are several conclusions that can be drawn. These will highlight the best performing algorithms and the variables that improve the most the learning of an optimal policy.

First of all, it seems that the Double Deep Q Network works the best for the environment that I have tested it on, a continuous action space with a linear reward function. It has proven to be the most reliable and sample efficient algorithm, providing a relatively stable policy and being able to solve the environment in several instances throughout the experiments. In terms of inputs, the handcrafted features are superior to the raw inputs, or at least given the architecture of the convolutional neural network that I have used. Using four stacked images proved to be detrimental for this environment, extracting features that are not relevant for the task at hand. For this algorithm using a more aggressive exploration rate increases noticeably stability, while also sacrificing peak performances, which were higher in the other versions of the algorithm. Furthermore, the rate at which the updates are performed on the target network does not seem to affect the performance of the network too much.

On the other hand, the Deep Q Learning algorithm did not fare too well in these experiments, achieving results well below the ones obtained when using DDQN or Evolutionary Algorithms. This might be due to the overoptimistic estimations that DQN is prone to, as it often times defaulted to either going straight ahead or circling. One unexpected result was that when using handcrafted features the results were worse than when using images as inputs, which might indicate that online learning benefits more from processing the environment through convolutional neural networks. However, this result needs further investigation.

The evolutionary algorithm represented a surprise as it managed to achieve reasonable results after spawning 150 generations of 12 species. Given enough time these methods represent a viable alternative to the reinforcement learning algorithms and even better in some

cases, outperforming the DQN algorithm in several instances. This method is not sample efficient, but it has the advantage that it is very simple from an algorithmic standpoint and can be applied to continuous and discrete action spaces alike.

The last algorithm that I have implemented is the Deep Deterministic Policy Gradient. This algorithm worked well in the LunarLander environment where the reward function was non-linear and the state space was quite small. This environment was solved after about 1400 episodes, making the algorithm not particularly sample efficient. When applying the same algorithm on the CarRacing enviornment there was no improvement recorded, which goes to show that this method is highly sensitive and most likely the way in which the reward function is built is heavily affecting the performance recorded.

Concluding, the Double Deep Q Learning algorithm has managed this task the best and could represent a solution for taking on a problem with a continuous action space and linear reward function, if an appropriate discretization function is used. One further improvement to the algorithms that benefit from using a replay buffer, DDQN and DDPG (and possibly DQN, needs further investigation), would be to use prioritized experience replay (Schaul, Quan, Antonoglou, Silver, 2016), which samples transitions for the mini-batch update based on the frequency with which they were originally experienced. This technique has the potential to boost the learning performance by feeding the agent only the most relevant transitions that provide the most important information regarding the behavior of the environment and the key states in an episode.

Bibliography

Darwin, Charles (1859). *On the origin of species by means of natural selection, or preservation of favoured races in the struggle for life.* London :John Murray

Richard S. Sutton and Andrew G. Barto. (2018) *Reinforcement Learning: An Introduction.* A Bradford Book, Cambridge, MA, USA.

Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N.M., Erez, T., Tassa, Y., Silver, D. and Wierstra, D. (2015). *Continuous control with deep reinforcement learning.* CoRR, abs/1509.02971.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. Riedmiller, M.A. (2013). *Playing Atari with Deep Reinforcement Learning.*, In Neural Information Processing Systems.

Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. (2012) *Imagenet classification with deep con- volutional neural networks.* In Advances in Neural Information Processing System, pages 11061114

Bellman, Richard Ernest (1957). *Dynamic programming.*; Rand Corporation, Princeton University

Bhatnagar, S., Sutton, R. S., Ghavamzadeh, M. and Lee, M. (2007). *Incremental natural actor-critic algorithms.* In Neural Information Processing Systems.

Degris, T., White, M. and Sutton, R. S. (2012b). *Linear off-policy actor-critic.* In 29th International Conference on Machine Learning.

Peters J. and Bagnell J.A. (2011) Policy Gradient Methods. In: Sammut C. and Webb G.I. (eds) *Encyclopedia of Machine Learning.* Springer, Boston, MA

A. M. Turing, (1952), The Chemical Basis of Morphogenesis, *Philosophical Transactions of the Royal Society of London.* Series B, Biological Sciences, Vol. 237, No. 641.

J. V. Neumann and A. W. Burks, (1966), *Theory of Self-Reproducing Automata*, Published by University of Illinois Press; First Edition

Sloss, Andrew Gustafson, Steven. (2019) *Evolutionary Algorithms Review.*, ArXiv, abs/1906.08870.

J.R. Koza, (1994), *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*, Published by A Bradford Book

D.E. Goldberg, (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Published by Addison-Wesley Professional

F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley and J. Clune, (2018), *Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*, Uber.

Signorelli, M., Vinciotti, V. and Wit, E. C. (2016). NEAT: an efficient network enrichment analysis test., BMC Bioinformatics, 17:352. DOI: 10.1186/s12859- 016-1203-6.

Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala and Yann LeCun. (2013), *Pedestrian detection with unsupervised multi-stage feature learning.* In Proc. International Conference on Computer Vision and Pattern Recognition (CVPR 2013)

Fridman, A. (2018). *Human-Centered Autonomous Vehicle Systems: Principles of Effective Shared Autonomy.*, MIT, Massachusets.

Y. LeCun, B Boser, J Denker, D. Henderson, R. Howard, W. Hubbard and L Jackel (1989), Backpropagation Applied to Handwritten Zip Code Recognition in *Neural Computation*, vol. 1, no. 4, pp. 541-551

LeCun Y., Haffner P., Bottou L. and Bengio Y. (1999) Object Recognition with Gradient-Based Learning. In: *Shape, Contour and Grouping in Computer Vision. Lecture Notes in Computer Science*, vol 1681. Springer, Berlin, Heidelberg

Rumelhart, D., Hinton, G. and Williams, R. (1986), Learning representations by back-

propagating errors, *Nature*, 323, 533536 (1986). https://doi.org/10.1038/323533a0

Leemon Baird (1995), *Residual algorithms: Reinforcement learning with function approximation.* In Proceedings of the 12th International Conference on Machine Learning (ICML 1995), pages 3037. Morgan Kaufmann, 1995

Gerald Tesauro, (1995) *Temporal difference learning and td-gammon.* Communications of the ACM, 38(3):5868

Lipson, Hod and Pollack, Jordan. (2010), Automatic design and manufacture of robotic life forms, *Nature* 260-267. 10.1017/CBO9780511730191.024.

Long-Ji Lin (1993) *Reinforcement learning for robots using neural networks. Technical report* DTIC Document

Hubel, D. H., and Wiesel, T. N. (1962), Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1), p 106154

LeCun Y.A., Bottou L., Orr G.B. and Mller KR. (1998) *Efficient BackProp* In: Montavon G., Orr G.B., Mller KR. (eds) *Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science*, vol 7700. Springer, Berlin, Heidelberg

Ruder, S. (2016) *An overview of gradient descent optimization algorithms*, ArXiv, abs/1609.04747.

Hasselt, H.V., Guez, A. and Silver, D. (2016). *Deep Reinforcement Learning with Double Q-Learning*, AAAI

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D. and Riedmiller, (2014) Deterministic policy gradient algorithms. In International Conference on Machine Learning (ICML)

Ian H. Witten (1977) *An adaptive optimal controller for discrete-time Markov environments. Information and Control*, Volume 34, Issue 4, Pages 286-295

S. Thrun and A. Schwartz (1993) *Issues in using function approximation for reinforcement*

learning In M. Mozer, P. Smolensky, D. Touretzky, J. Elman, and A. Weigend, editors, Proceedings of the 1993 Connectionist Models Summer School, Hillsdale, NJ, 1993. Lawrence Erlbaum.

Uhlenbeck, George E and Ornstein, Leonard S (1930) On the theory of the brownian motion, in *American Physical Society*, 36(5):823

Stanley, K.O., Clune, J. and Lehman, J. (2019), Designing neural networks through neuroevolution.*Nat Mach Intell*, 2435

G. S. Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto and M. Fujita, (2000) *Evolving robust gaits with AIBO* Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065), San Francisco, CA, USA, 2000, pp. 3040-3045 vol.3.

Benbrahim, H and Franklin, JA (1997) Biped dynamic walking using reinforcement learning In Benbrahim, H and Franklin, JA (1997) *Robotics and Autonomous Systems*, Springer, 22(34): 283302.

Abbeel, P, Coates, A, Quigley, M adn Ng, AY (2007) *An application of reinforcement learning to aerobatic helicopter flight.* In: Advances in Neural Information Processing Systems (NIPS).

Kober, J., Bagnell, J. A., and Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11), 12381274.

Ian H Witten. (1977) An adaptive optimal controller for discrete-time markov environments. *Information and control*, Essex, England, 34(4):286295

Schaul, Tom, John Quan, Ioannis Antonoglou and David Silver. (2016) *Prioritized Experience Replay* CoRR abs/1511.05952

Ioffe, Sergey and Szegedy, Christian. (2015) *Batch normalization: Accelerating deep network training by reducing internal covariate shift.* In Proceedings of the 32nd International

Conference on International Conference on Machine Learning - Volume 37 (ICML15)

Watkins, C.J.C.H. (1989) *Learning from Delayed Rewards.* PhD Thesis, University of Cambridge, England.

Radu Burtea - ThesisRaduBurtea - https://github.com/raduburtea/ThesisRaduBurtea

Luc Prieur, CarRacing-v0, https://gist.github.com/lmclupr/b35c89b2f8f81b443166e88b787b03abfile-race-car-cv2-nn-network-td0-15-possible-actions-ipynb [last access: 25/06/2020]

Yan Panlau, DDPG Keras Torcs, https://github.com/yanpanlau/DDPG-Keras-Torcs [last access: 25/06/2020]

Phil Tabor, BipedalWalker,https://github.com/philtabor/Youtube-Code-Repository/tree/master/Reinforc
[last access: 25/06/2020]

OpenAI Gym, https://gym.openai.com, [last access: 25/06/2020]