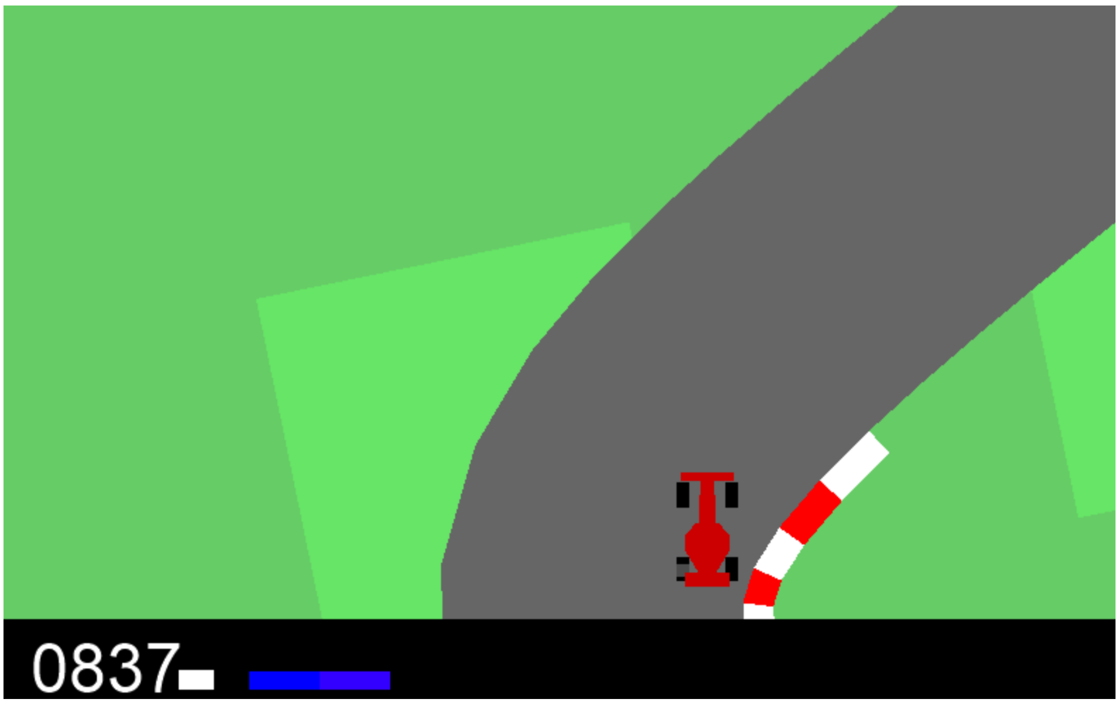
Self Driving Car in OpenAI Gym using Imitation Learning and Reinforcement Learning

INFO 7390 - FALL 2020
Final Project - Advance Data Science
Under Guidance of
Prof. Brown, Nicholas

Nisher, Rushabh
nisher.r@northeastern.edu

Sharma, Manali
sharma.ma@northeastern.edu

December 19, 2020



Contents

Abstract

We all know self-driving cars is one of the hottest areas of research and business for the tech giants. What seemed like a science-fiction, a few years ago, now seems more like something which is soon to become a part and parcel of life. The reason, I am saying "soon to be" is because of the fact that even though companies like Tesla, Nissan, Cadillac do have self-driving car assistance software, but, they still require a human to keep an eye on the road and take control when needed. However, it is fascinating to see how far we have come in terms of innovation and how fast technology is advancing. So much so, that now, with the help of basic deep learning, neural network magic, we can build our own pipeline for autonomous driving.

Our idea to try and build our very own self driving car emerged from here. In order to understand the basics of the process , we did this project in two parts.

- Self Driving Car using Supervised Learning
- Self Driving Car using Reinforcement Learning

PS- To make you understand the structure for the same, We have done this project in 3 parts, and all 3 parts are divided into separate notebooks. And these individual notebooks contain the whole code and documentation of the entire part.

1 Introduction

1.1 Problem Statement

In this project, a python based car racing environment is trained using two approaches Imitation Learning approach and Deep Q-Learning approach. The aim of the trained model is to perform efficient self driving on a racing track in the OpenAI Gym Environment. This project aims at making a comparison between the two approaches and tries to lay out the advantages and disadvantages involved in both the processes. The measure of goodness of the model is given by the gym environment at the bottom left corner.

According to OpenAI Gym, this environment is considered solved when the agent successfully reaches an average score of 900 on the last 100 runs.

1.2 Approach

For the Imitation Learning approach we have applied a supervised learning algorithm (convolution networks), to control the direction of a car in a 2D simulation. We create the dataset by playing in the environment and use it for training our network, then we use gym to retrieve the output of the neural network in order to control the simulation.

The general idea is that we use supervised classifier and train a convolutional neural network to classify images in the game, according to three labels: left, right and straight ahead. We then convert these commands into instructions for the simulator, which will execute them.

For the Deep Q-Learning approach, we train a deep Q learning algorithm and then use it to train an autonomous driver agent. Different configurations in the deep Q learning algorithm parameters and in the neural network architecture are then tested and compared in order to obtain the best racing car average score over a period of 100 races.

2 Reinforcement Learning

2.1 Reinforcement Learning in Machine Learning

Reinforcement learning is the training of machine learning models to make a sequence of decisions. Formally the environment in RL is modeled as a Markov decision process (MDP): it involves an agent, a set of states S, and a set A of actions per state. By performing an action a $\in$ A, the agent transitions from state to state. Executing an action in a specific state provides the agent with a reward r (a numerical score).

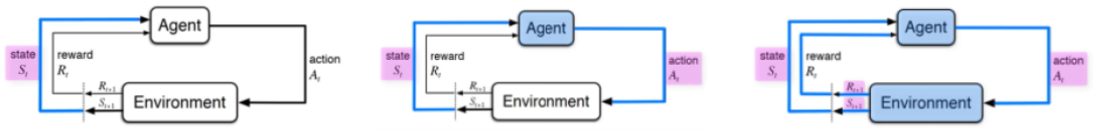


Figure 2: Reinforcement Learning Process

The goal of the agent is to maximize its total (future) reward across an episode. This episode is anything and everything that happens between the first state and the last or terminal state within the environment. It does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward. This potential reward is a weighted sum of the expected values of the rewards of all future steps starting from the current state. We reinforce the agent to learn to perform the best actions by experience. This is the strategy or policy.

In reinforcement learning, the aim is to weight the network (devise a policy) to perform actions that minimize long-term (expected cumulative) cost. At any juncture, the agent decides whether to explore new actions to uncover their costs or to exploit prior learning to proceed more quickly. We can define ϵ-greedy ($0 \leq \epsilon \leq 1$) which is a parameter controlling the amount of exploration vs. exploitation. ϵ is usually a fixed parameter but can be adjusted to make the agent explore progressively less.

2.2 Markov Decision Process

A Markov Process is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. Future states of the process (conditional on both past and present states) depends only upon the present state, not on the sequence of events that preceded it. The next state s' depends on the current state s and the action a. But given s and a, it is conditionally independent of all previous states and actions. In a Markov decision process:

- S is a finite set of states;
- A is a finite set of actions (also, A_s is the finite set of actions available from state s);
- P_a(s; s') is the probability of transition from s to s' under action a;
- R_a(s; s') is the immediate reward (or expected immediate reward) received after transition from state s to state s' with action a.

The core problem of MDPs is to find a “policy” for the decision maker: a function π that specifies the action π(s) that the decision maker will choose when in state s. Once a Markov decision process is combined with a policy in this way, this fixes the action for each state: the action chosen in state s is completely determined by π(s). The goal is to choose a policy π that will maximize some cumulative function of the random rewards, typically the Expected Discounted Reward sum over a potentially infinite horizon:

$$Ex[\sum_{t=0} \gamma^t R_{a_t}(s_t, s_{t+1})]$$

Where a = π(s) is chosen (i.e. actions given by the policy). And the expectation is taken over: $s_{t+1} \approx P_{a_t}\ (s_t;\ s_{t+1})$. And where γ is the discount factor ($0 \leq \gamma \leq 1$).

The standard family of algorithms to calculate this optimal policy requires storage for two arrays indexed by state: value V, which contains real values, and policy π which contains actions. At the end of the algorithm, π will contain the solution and V(s) will contain the discounted sum of the rewards to be earned (on average) by following that solution from state s.

$$V(s) := \sum_{s'} P_{\pi(s)}(s, s')(R_{\pi(s)}(s, s') + \gamma V(s'))$$

$$\pi(s) := argmax_a \left\{ \sum_{s'} P(s'|s, a)(R(s'|s, a) + \gamma V(s') \right\}$$

2.3 Q-Learning

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model of the environment (hence the connotation ”model-free”), and it can handle problems with stochastic transitions and rewards, without requiring adaptations. For any Finite Markov Decision Process (FMDP), Q-learning finds an optimal action-selection policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. “Q” names the function that returns the reward used to provide the reinforcement and can be said to stand for the “quality” of an action taken in each state.

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

The above equation states that the Q-value yielded from being at state s and performing action a is: the immediate reward r(s,a) plus the highest Q-value possible from the next state s'.γ is the discount factor which controls the contribution of rewards further in the future.

Q(s',a) again depends on Q(s",a) which will then have a coefficient of gamma squared. So, the Q-value depends on Q-values of future states as shown here:

$$Q(s,a) \rightarrow \gamma Q(s',a) + \gamma^2 Q(s'',a)..........\gamma^n Q(s''^{..n},a)$$

Adjusting the value of γ will diminish or increase the contribution of future rewards. The weight for a step from a state $\triangle$t steps into the future is calculated as $\gamma^{\triangle t}$, where γ (the discount factor) is a number between 0 and 1, and has the effect of valuing rewards received earlier higher than those received later (reflecting the value of a "good start"). γ may also be interpreted as the probability to succeed (or survive) at every step $\triangle$t The algorithm, therefore, has a function that calculates the quality of a state-action combination:

$$Q : S \times A \rightarrow R$$

Before learning begins, Q is initialized to a possibly arbitrary

xed value (chosen by the programmer). Then, at each time t the agent selects an action a_t , observes a reward r_t, enters a new state s_{t+1} (that may depend on both the previous state s_t and the selected action), and Q is updated. The core of the algorithm is a simple value iteration update, using the weighted average of the old value and the new information:

$$Q^{new}(s_t,a_t) \longleftarrow (1-\alpha) \cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}} \tag{7}$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and α is the learning rate $(0 < \alpha \leq 1)$.

3 Imitation Learning

Generally, imitation learning is useful when it is easier for an expert to demonstrate the desired behaviour rather than to specify a reward function which would generate the same behaviour or to directly learn the policy. The main component of IL is the environment, which is essentially a Markov Decision Process (MDP). This means that the environment has an S set of states, an A set of actions, a P(s—s,a) transition model (which is the probability that an action a in the state s leads to state s) and an unknown R(s,a) reward function. The agent performs different actions in this environment based on its policy. We also have the experts demonstrations (which are also known as trajectories) = (s0, a0, s1, a1,) , where the actions are based on the experts (“optimal”) π * policy. In some cases, we even “have access” to the expert at training time, which means that we can query the expert for more demonstrations or for evaluation. Finally, the loss function and the learning algorithm are two main components, in which the various imitation learning methods differ from each other.

3.1 Why Imitation Learning over Reinforcement Learning?

Reinforcement learning (RL) is one of the most interesting areas of machine learning, where an agent interacts with an environment by following a policy. In each state of the environment, it takes action based on the policy, and as a result, receives a reward and transitions to a new state. The goal of RL is to learn an optimal policy which maximizes the long-term cumulative rewards.

To achieve this, there are several RL algorithms and methods, which use the received rewards as the main approach to approximate the best policy. Generally, these methods perform really well. In some cases, though the teaching process is challenging. This can be especially true in an environment where the rewards are sparse (e.g. a game where we only receive a reward when the game is won or lost). To help with this issue, we can manually design rewards functions, which provide the agent with more frequent rewards. Also, in certain scenarios, there isnt any direct reward function (e.g. teaching a self-driving vehicle), thus, the manual approach is necessary.

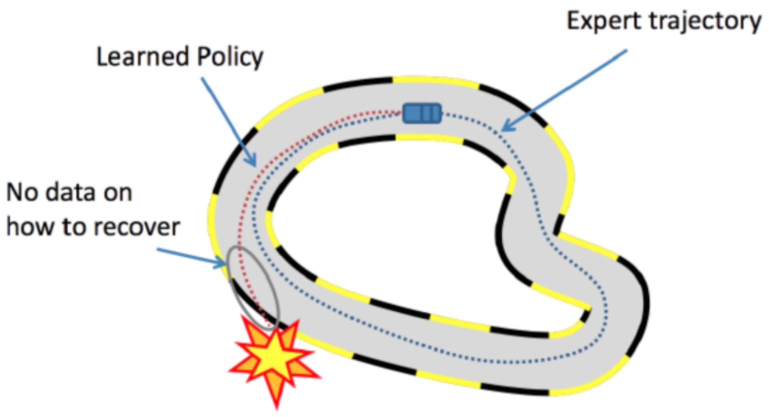
However, manually designing a reward function that satisfies the desired behaviour can be extremely complicated. A feasible solution to this problem is imitation learning (IL). In IL instead of trying to learn from the sparse rewards or manually specifying a reward function, an expert (typically a human) provides us with a set of demonstrations. The agent then tries to learn the optimal policy by following, imitating the experts decisions.

3.2 Behavioral Cloning

The simplest form of imitation learning is behaviour cloning (BC), which focuses on learning the experts policy using supervised learning. An important example of behaviour cloning is ALVINN, a vehicle equipped with sensors, which learned to map the sensor inputs into steering angles and drive autonomously. This project was carried out in 1989 by Dean Pomerleau, and it was also the first application of imitation learning in general. The way behavioural cloning works is quite simple. Given the experts demonstrations, we divide these into state-action pairs, we treat these pairs as i.i.d. examples and finally, we apply supervised learning. The loss function can depend on the application. Therefore, the algorithm is the following:

1. Collect demonstrations (τ^* trajectories) from expert
2. Treat the demonstrations as i.i.d. state-action pairs: $(s_0^*, a_0^*), (s_1^*, a_1^*), \ldots$
3. Learn π_θ policy using supervised learning by minimizing the loss function $L(a^*, \pi_\theta(s))$

In some applications, behavioural cloning can work excellently. For the majority of the cases, though, behavioural cloning can be quite problematic. The main reason for this is the i.i.d. assumption: while supervised learning assumes that the state-action pairs are distributed i.i.d., in MDP an action in a given state induces the next state, which breaks the previous assumption. This also means, that errors made in different states add up, therefore a mistake made by the agent can easily put it into a state that the expert has never visited and the agent has never trained on. In such states, the behaviour is undefined and this can lead to catastrophic failures.



Still, behavioural cloning can work quite well in certain applications. Its main advantages are its simplicity and efficiency. Suitable applications can be those, where we dont need long-term planning, the experts trajectories can cover the state space, and where committing an error doesnt lead to fatal consequences. However, we should avoid using BC when any of these characteristics are true.

4 Deep Reinforcement Learning

4.1 Deep Learning

Unsupervised pre-training and increased computing power from GPUs and distributed computing allowed the use of larger networks, particularly in image and visual recognition problems, which became known as "deep learning"

The word "deep" in "deep learning" refers to the number of layers through which the data is transformed. More precisely, deep learning systems have a substantial credit assignment path (CAP) depth. The CAP is the chain of transformations from input to output. CAPs describe potentially causal connections between input and output. Deep models have a CAP ¿ 2 and they are able to extract better features than shallow models and hence, extra layers help in learning the features effectively.

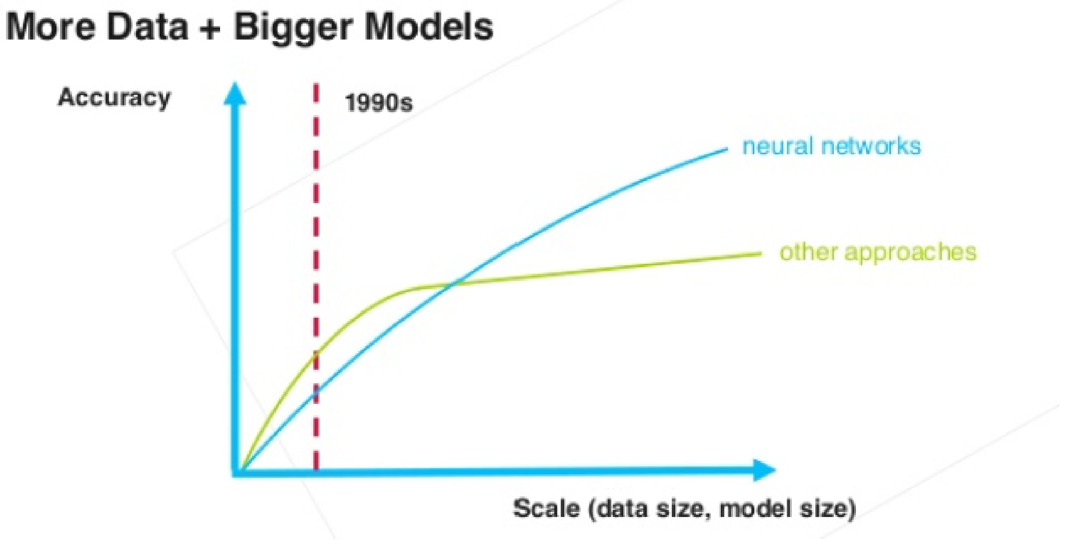


Figure 3: Deep Learning VS Older Algorithms

4.2 Deep Q-Learning

Q-learning at its simplest stores data in tables. This approach falters with increasing numbers of states/actions.

Q-learning can be combined with function approximation, such as an (adapted) artificial neural network.

This makes it possible to apply the algorithm to larger problems and also to speed up learning infinite problems, due to the fact that the algorithm can generalize earlier experiences to previously unseen states.

Q-learning is a a powerful algorithm to help the agent to figure out exactly which action to perform. But imagine now an environment with 10,000 states and 1,000 actions per state. This would create a table of 10 million cells, which presents two problems: Large amount of memory required (to save and update that table) and large amount of time required to explore each state to create the required Q-table. So, DeepMind proposed to approximate these Q-values with machine learning models such as a neural network. This led to its acquisition by Google and in 2014, Google DeepMind patented the “deep reinforcement learning” or “deep Q-learning” (that can play Atari 2600 games at expert human levels).

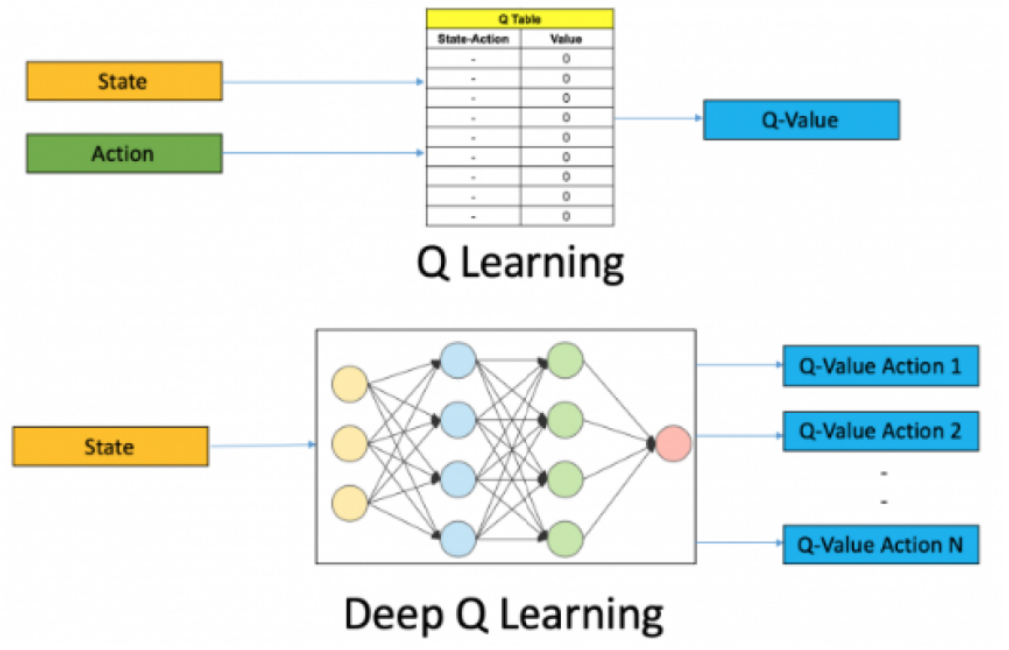


Figure 4: Q-Learning VS Deep Q-Learning

In Deep Q-Learning, the state is given as the input and the output generated is the Q-value of all possible actions

Steps involved in reinforcement learning using deep Q-learning networks (DQNs):

1. All the past experience is stored by the user in memory;
2. The next action is determined by the maximum output of the Q-network;
3. The loss function here is mean squared error of the predicted Q-value and the target Q-value -Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning

problem. Going back to the Q-value update equation derived from the Bellman equation (4). we have:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \underset{a}{max} Q(s_{t+1}, a) - Q(S_t, A_t)]$$

Pseudocode for deep Q-learning:

```
Start with $Q_0$(s,a) for all s,a.
Get initial state s
For k = 1,2,... till convergence
        Sample action a, get next state s'
        If s' is terminal:
            target = R(s,a,s')
            Sample new initial state s'
        else:
            target = R(s,a,s')+$\gamma \underset{a'}{\max} Q_{k(s',a')}$
        $\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta E_{s' \ P(s'|s,a)}[(Q_\theta(s,a) - target(s'))^2]|_{\theta=\theta_k}$
        $s \leftarrow s'$
```

Note that the target is unstable / non-stationary: it is continuously changing with each iteration.

So, we try to learn to map for a constantly changing input and output. But then what is the solution?

Since the same network is calculating the predicted value and the target value, there could be a lot of divergence between these two. So, instead of using one neural network for learning, we can use two.

We could use a separate network to estimate the target. This target network has the same architecture as the function approximator but with frozen parameters. For every C iterations (a hyperparameter), the parameters from the prediction network are copied to the target network. This leads to more stable training because it keeps the target function fixed (for a while):

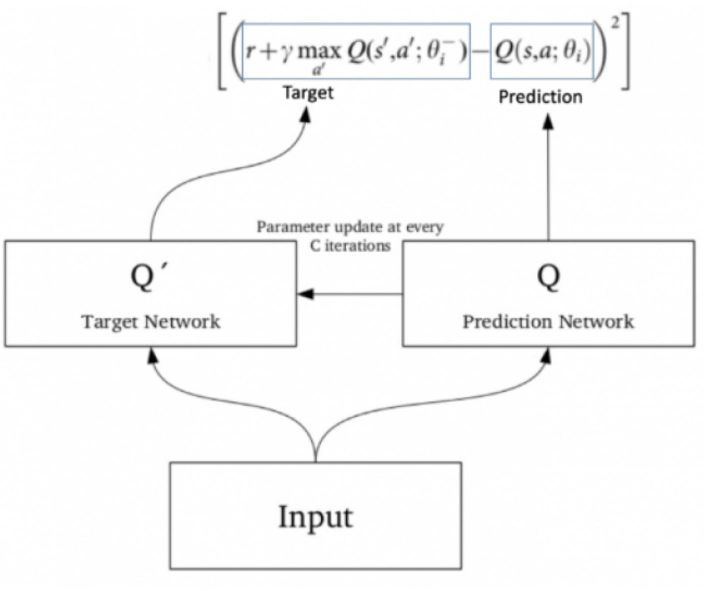

$\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-)\right) - Q(s,a;\theta_i)\right]^2$
Target
Prediction
Parameter update at every C iterations
Q′
Target Network
Q
Prediction Network
Input

Figure 5: Two Neural Networks system implementation

In deep learning, the target variable does not change and hence the training is stable, which is just not true for RL.

Deep Q Network

1. Pre-process and feed the game screen (state s) to the DQN, which will return the Q-values of all possible actions in the state
2. Select an action using the epsilon-greedy policy. With the probability epsilon, we select a random action a and with probability 1-epsilon, we select an action that has a maximum Q-value, such as a = argmax(Q(s,a,w))
3. Perform this action in a state s and move to a new state s' to receive a reward. This state s' is the pre-processed image of the next game screen. We store this transition in our replay buffer as ¡s,a,r,s'¿
4. Next, sample some random batches of transitions from the replay buffer and calculate the loss
5. It is known that: $Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$ which is just the squared difference between target Q and Predicted Q
6. Perform gradient descent with respect to the actual network parameters in order to minimize this loss
7. After every C iterations, copy the actual network weights to the target network weights
8. Repeat these steps for M number of episodes

5 Model training with Pytorch

Pytorch is a python matrix computing and deep learning library. It consists on the one hand in an equivalent of numpy, but usable both on CPU and on GPU. And on the other hand, in a library which allows to calculate the gradient of each operation performed on the data, so as to apply the backpropagation algorithm, At the base of the training of neural networks Pytorch also has a set of modules to assemble, which makes it possible to create neural networks very simply.

In pytorch, the basic object is the module. Each module is a function, or an assembly of pytorch functions, which takes as input Tensors (matrices containing data), and emerges another tensor. All the operations performed in this module will be recorded, because the operation graph is necessary for the backpropagation algorithm.

5.1 Functions

The __init__ function :

Here we define the network architecture. Our network is made up of two parts: self.convnet and self.classifier. The part convnet is the convolutional part: it is she who is responsible for analyzing the image, and recognizing the shapes. It is made up of two layers of convolution (pattern recognition), followed by a non-linearity (ReLU), and a pooling layer (which makes the output invariant to translations).

The second part is the 'classifier', it takes the output of the convolution network, and a size vector emerges num_classes = 3 which represents the score of each action to be performed.

The call nn.Sequential creates the layers in succession. The input will pass successively through all these layers, the input of one layer being the output of the previous one.

The forward function:

This function will be called by pytorch when our module is called. We notice the passage from a 2D input to a 1D input between the two parts convnet and classify thanks to the function input = input.view(input.size(0), -1)(the first dimension being the number of images in a batch). It's a shortcut for input = input.view(input.size(0), input.size(1) * input.size(2) * input.size(3)

The entry will indeed have 4 dimensions: the first for the batch, the next 2 for the x and y dimensions of the image, and the last for the number of channels of the image: It will be 3 for the 3 colors to entering the network, then each convolution will create new channels while reducing the x and y size. Thus, as the layers progress, the 1st dimension will remain fixed (the number of images in the batch), but the following two will decrease, and the 3rd (channels) will increase.

The __len__ : This function should return the length of the dataset. Here is the total number of images.

The __getitem __ (self, index) function : This function should return the index object index. Here, we load the image corresponding to this index, we apply the transformations to it, then we return the matrix as well as the labels (in the form of Tensor).

Directions : We have encoded the directions in three variables LEFT, RIGHT and GO, which will be used in the different modules.

6 Deep Q Neural Networks

6.1 Hyperparameters of a Deep Q Learning Network

In any deep Q learning project involving the neural network, we need to specify certain hyperparameters. Tuning of these hyperparameters is crucial and can give varying results when we tune them. We have experimented with some parameters within the following value ranges.

Hyperparameters Table

Hyperparameter	Value Range	Description
ϵ	[1]	Initial value of epsilon, the initial exploration rate of actions
min ϵ	[0.05, 0.1]	Minimum value of the epsilon, the final exploration rate of actions
ϵ decay steps	[10000, 100000]	The number of total time steps it takes to reduce ϵ to its final value min ϵ
min exp size	[1000, 10000]	Minimum number of experiences saved, these are done at the beginning before the training
exp capacity	[50000, 200000]	Maximum number of experiences saved, when this capacity is full, old experiences are erased for new ones.
target network update freq	[500, 1000]	How often the prediction network parameters are copied to the target network in time steps
max negative rewards	[5, 20]	Maximum number of consecutive negative rewards before the script considers it an early ending
batchsize	[32, 128]	Number of training cases over which each loss function minimization is computed
num frame stack	[1, 4]	Number of frames used in sequence input to the neural network, also corresponds to how often the network is trained
gamma	[0.95, 0.99]	The discount factor, it is multiplied by future rewards as discovered by the agent in order to dampen the rewards' effect on the agent's choice of action

In the end of this report, a final table is attached with the final parameter values used along with other deep learning hyperparameters.

6.2 Input Data Preprocessing

For Reinforcement Learning

Originally, the observations obtained from the environment were colored RGB images with a 96×96 shape. Since these observation images are the training inputs for the neural network, they were pre-processed to remove unwanted information and improve the training results.

First, the three channel image was converted to a single channel grayscale. Then, all the grass colors were replaced with white. Finally, the score number information was also removed by placing a black square on the bottom left corner.

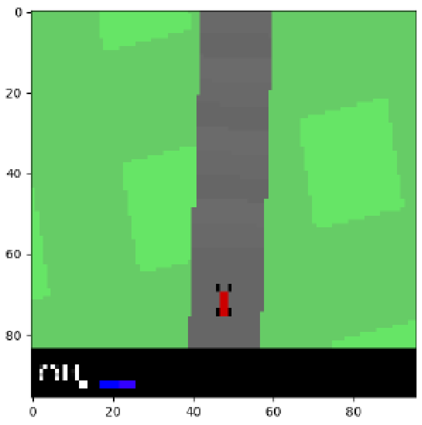


Figure 6: Original observation image

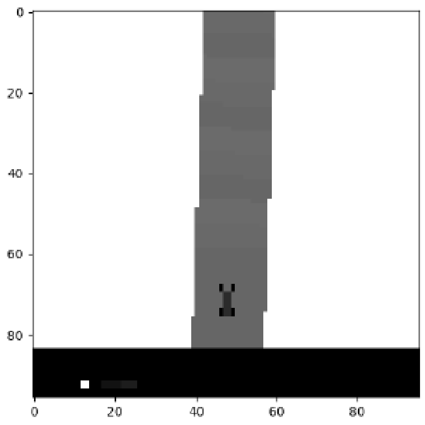


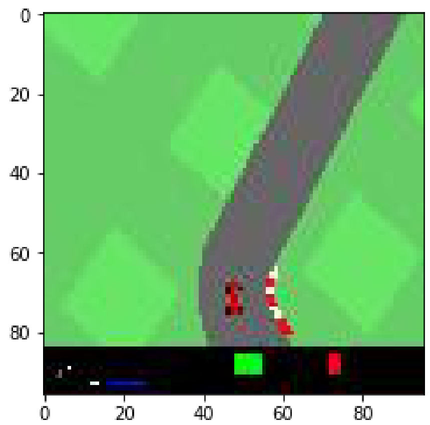
Figure 7: Observation image after preprocessing

For Imitation Learning

First of all, we will define the transformations that will be used to preprocess the images, in order to give them as input to the neural network.

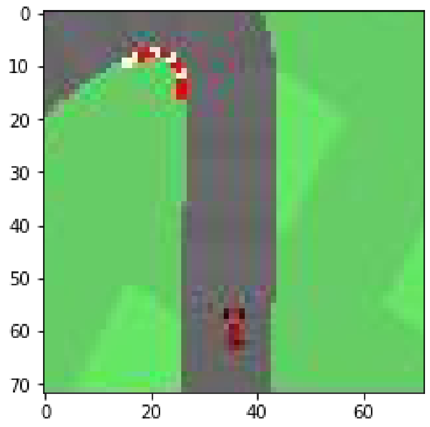
These transformation performs the following actions:

Crop the image: transforms.CenterCrop(72)
To keep only a square of size 72 pixels, centered in the same way as the image. Indeed, the image we get of the environment is like this:



We can see that the screen displays an indication bar on the speed and the direction and acceleration controls. If we do not hide it, CNN may learn to associate the commands we give it, with these indications (this is indeed the best indicator to deduce the command to be made from the screen).

After cropping, the resulting image is below. CNN will be forced to analyze the road and the position of the car in order to



Note: The images provided to CNN are of much lower quality than those displayed by the environment during the game. They are indeed only 96 pixels apart. This will be enough for the neural network to analyze the shapes, and will make the training much faster (because much less neurons will be needed).

Transform the matrix into Tensor pytorch: transforms.ToTensor()

The tensor is the base object in pytorch to store data. It is analogous to a numpy matrix, except that it can be stored on CPU, or on GPU. We need to transform our image into a pytorch tensor before giving it to the neural network as input.

We could also use the function tensor.from_numpy(numpy_array)to transform a numpy array into Tensor.

Standardization : transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
The images provided by PIL have data between 0 and 1. Here we subtract 0.5, and divide by 0.5 in order to have data between -1 and 1, which is more efficient for training a network. neuron (data centered in 0 and variance close to 1).

6.3 Neural Network Architecture for Deep Q Learning

6.3.1 Convolutional Layers

A convolutional layers is used in neural networks to extract features from image inputs. Image data usually has a shape of (Number of images) x (Width) x (Height) x (Channels (ex: RGB)), and by convolving it with a multiplication or other dot product through a kernel, patterns in the image can be extracted. Stride is a measure of the translation step in the convolution operation. Together with padding, and kernel size, they are the main parameters when developing a convolutional layer.

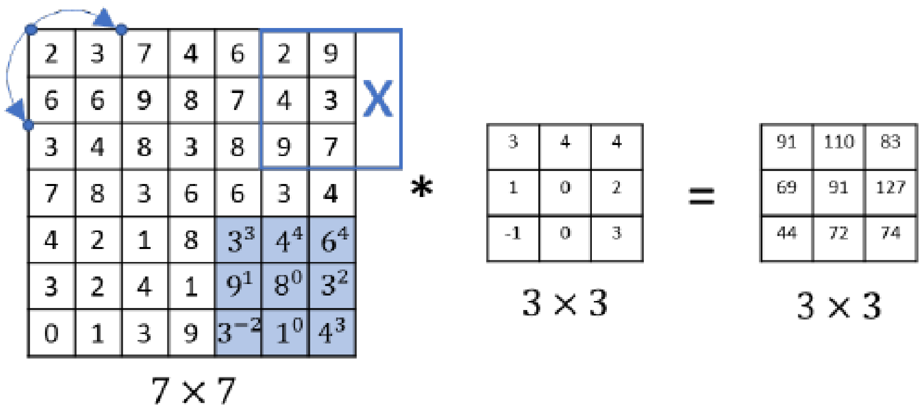


Figure 8: Example of a convolution using a 3x3 kernel with stride 2x2 [3]

When adding multiple convolutional layers, the depth of these features is increased. For example, in figure 9, the first layer recognizes different types of edges, the second layer uses these edges to recognize some basic shapes like circles, stripes, more specific features. Finally, layer 3 is now able to detect more complex shapes based on the features extracted on layer 2 like car wheels, textures, human silhouettes, etc...

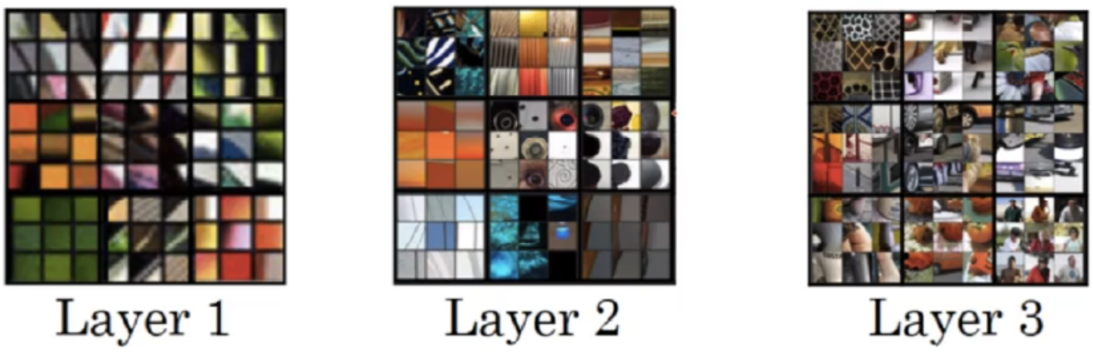


Figure 9: Visualization of CNN layers [3] [4]

6.3.2 Dense Layers

The dense layer, also referred to as fully connected layers, is a regular layer of neurons. Each neuron unit receives input from all the neurons in the previous layer, so, it is densely connected. It is usually used after convolutional layers, preceded by a flattening operation where the multidimensional output of the convolutional is converted into a one dimensional array.

6.3.3 Activation Functions

Nonlinear activation functions are preferred as they allow the nodes to learn more complex structures in the data. Two widely used nonlinear activation functions are the sigmoid and hyperbolic tangent activation functions. However, a general problem with both the sigmoid and tanh functions is that they saturate. This means that large values snap to 1.0 and small values snap to -1 or 0 for tanh and sigmoid respectively. Furthermore, the functions are only really sensitive to changes around their mid-point of their input, such as 0.5 for sigmoid and 0.0 for tanh.

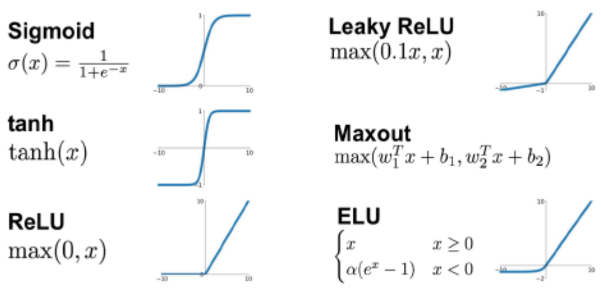


Figure 10: Commonly used activation functions

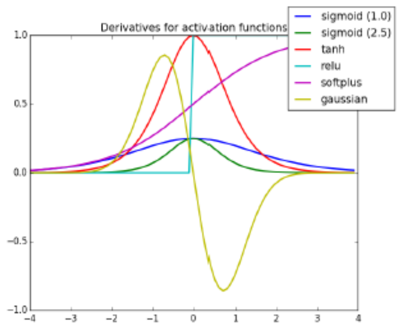


Figure 11: Derivative of common activation functions

For this implementation, the ReLU (Recti ed Linear Unit) activation function was chosen since it is the one which generally provides the best results. This is true because the derivative of ReLu [Figure 11] is either 0 or 1, so multiplying by it won't cause weights that are further away from the end result of the loss function to suffer from the vanishing gradient problem. Also, ReLU is more computationally efficient to compute than others such as the sigmoid. Overall, ReLU provides a faster converging model.

Considering a Neural Network with the following form:

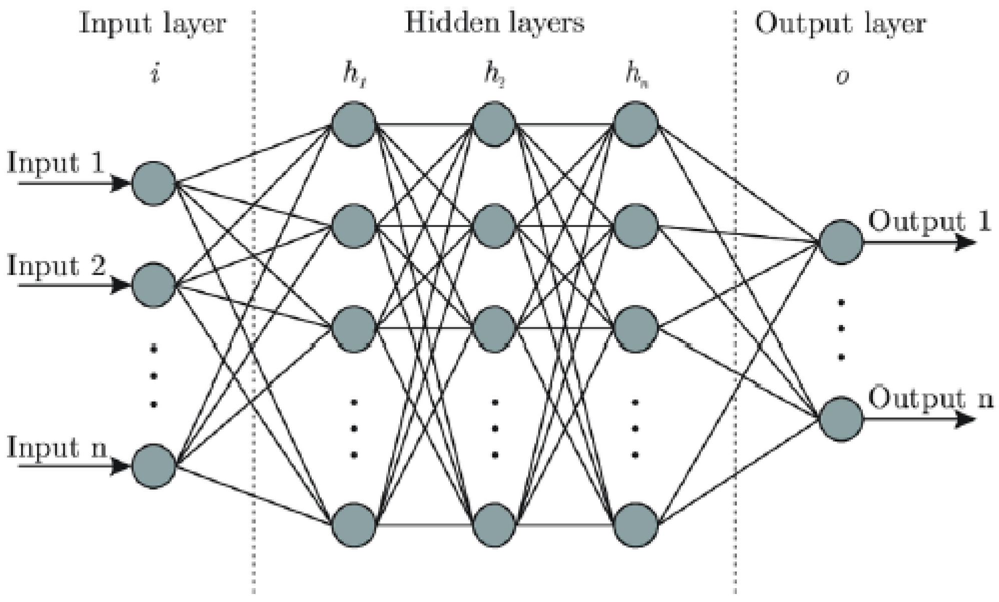


Figure 12: Example of a Deep Neural Network

The activation of the first neuron in the first hidden layer (layer 1), after the input layer (layer 0), is given by:

$$a_0^{(1)} = ReLU(w_{0,0}a_0^{(0)} + w_{0,1}a_1^{(0)} + ... + w_{0,n}a_n^{(0)} + b)$$

Where w represent the weights and b are the biases.

To represent all the activation functions between layers 0 and 1, we write it in matricial form:

$$\begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_n^{(1)} \end{bmatrix} = \begin{bmatrix} w_0, 0 & w_0, 1 & \dots & w_0, n \\ w_1, 0 & & & \vdots \\ \vdots & & & \vdots \\ w_k, 0 & & & w_k, n \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \tag{10}$$

6.3.4 Loss Function

At each step, after obtaining the Predicted Q, this value is going to be compared to the Target Q*. This way we calculate the Loss function:

$$L = (Q_{a_1} - Q^*_{a_1}) + (Q_{a_2} - Q^*_{a_2})^2 + ... + (Q_{a_n} - Q^2_{a_n})$$

Where n represents the number of possible actions a at each state s (i.e. the number of outputs)

The Loss function is what is intended to be minimized. So we calculate the negative of the Gradient of the Loss Function: $-\nabla L$, representing the 'direction' where the function decreases the most

This gradient vector contains all the partial derivatives of the Loss Function in respect to the weights and biases of all Activation Functions:

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w^{(1)}} \\ \frac{\partial L}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial L}{\partial w^{(n)}} \\ \frac{\partial L}{\partial b^{(n)}} \end{bmatrix}$$

Where each of these partial derivatives are obtained by the chain rule:

$$\frac{\partial L}{\partial w^{(k)}} = \frac{\partial z^{(k)}}{\partial w^{(k)}} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial L}{\partial a^{(k)}}$$

$$\frac{\partial L}{\partial b^{(k)}} = \frac{\partial z^{(k)}}{\partial b^{(k)}} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial L}{\partial a^{(k)}}$$

After minimizing the Loss function, all weights and biases are updated. [7]

6.3.5 Optimizers: Implementation and Learning Rate Decay

For a neural network, the curve/surface that that is attempted to optimize is the loss surface. Since we are trying to minimize the prediction error of the network, we are interested in finding the global minimum on this loss surface, which is the main aim of training a neural network.

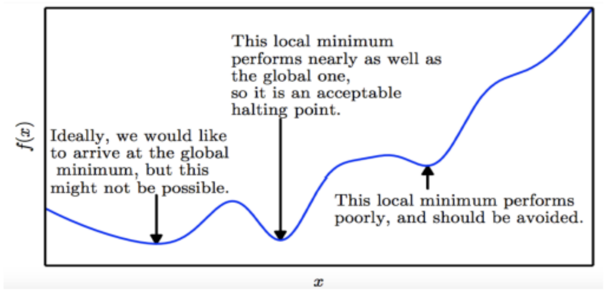


Figure 13: Optimization curve

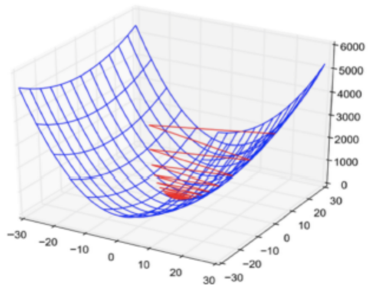


Figure 14: Optimization surface

For this project, it was explored the Adam optimizer, also known as Adaptive Moment Optimization. It was chosen for this project due to its simple code implementation. Instead of the classical stochastic gradient descent procedure, the Adam algorithm can be used to update network weights iterative based in training data. It is an optimizer that is known to generalize well and computationally efficient for different models, making it a safe choice.

- **alpha (α):** Commonly known as learning rate or step size. It is an indicator for the portion of weights that are updated. Larger values results in faster learning before the rate is updated. This is the main parameter to be tuned in any neural network application.
- **beta 1 ($\beta 1$):** The exponential decay rate for the first moment estimates. weighted average for dw
- **beta 2 ($\beta 2$):** The exponential decay rate for the second moment estimates. weighted average for dw^2
- **epsilon (ϵ):** Very small number to prevent division by zero in the implementation

Adam Optimization Algorithm:

Initialize $V_{dw} = 0$, $S_{dw} = 0$, $V_{db} = 0$, $S_{db} = 0$ where w are the weights and b represents the bias.
On iteration t:

```
    Compute $dw, db$ using current mini-batch
    Momentum update: $V_{dw} = \beta_1 V_{dw} + (1-\beta_1)dw$, $V_{db} = \beta_1 V_{db} + (1-\beta_1)db$
    RMSProp update: $S_{dw} = \beta_2 S_{dw} + (1-\beta_2)dw^2$, $S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2$
    $V_{dw}^{corrected} = \frac{V_{dw}}{(1-\beta_1^t)}, V_{db}^{corrected} = \frac{V_{db}}{(1-\beta_1^t)}$
    $S_{dw}^{corrected} = \frac{S_{dw}}{(1-\beta_w^t)}, S_{db}^{corrected} = \frac{S_{db}}{(1-\beta_w^t)}$
    Update $w$: $w = w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \epsilon}$ and update $b$: $b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$
```

Exponential Decay on Learning Rate

When training a model, it is often recommended to lower the learning rate as the training progresses. Therefore, exponential decay was implemented on the
nal tests according to equation:

$$\text{decayed } \alpha = \alpha * \text{decay } rate^{\frac{globalstep}{decayrate}}$$

with decay rate = 0.8 and decay steps = 200000. This was implemented within the tensorflow framework by following

6.3.6 Overfitting Prevention Methods: Weight Regularization

Overfitting occurs when a neural network is so closely fitted to the training set that it is difficult to generalize and make predictions for new data. Nowadays, many strategies exist to combat this problem, usually at the expense of the training error. One of these techniques, is weights regularization.

Weights regularization works by punishing the largest weights in a network during training. This makes room in the network and forces other neurons to learn to

t the data, thus making it more reliable on different neurons and not only on a specific set of neurons. This was the main regularization measure implemented in our project.

6.3.7 Overfitting Prevention Methods: Dropout Layers

Just like with weights regularization, the usage of dropout layers is a good solution to prevent overfitting. It works by disabling a random percentage of neurons in the hidden layers on the neural network, thus making it so that the training has to rely on multiple combinations to optimize the fit. In convolutional neural networks, it is usually used in between two convolutional layers or before the dense layers. Furthermore, dropout layers are known to work best in smaller network architectures.

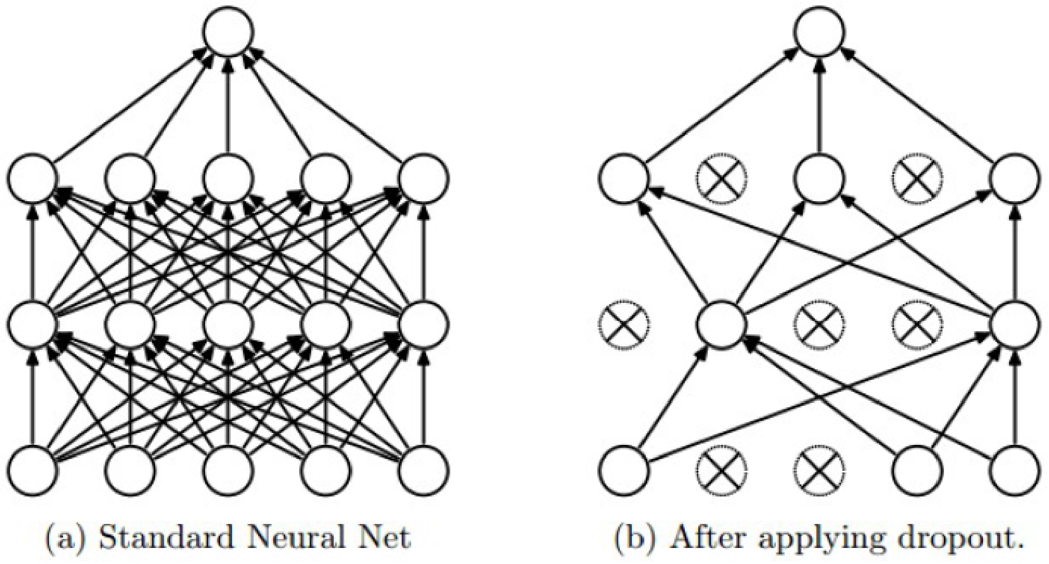


Figure 15: Example of three 40% dropout layers

7 Autonomous Racing Car Driver Agent

7.1 Open AI Gym: Car Racing Environment

The CarRacing-V0 environment available on the OpenAI Gym python library consists of a control task where the agent learns from pixels. A top-down racing environment. The state consists of 96 $\times$ 96 pixels. The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of visible tiles in track. For example, if you have finished in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points thus rewarding the agent for visiting as many tiles as possible while punishing him for taking a long time to do it. Episode finishes when all tiles are visited or when a certain time has passed. Some indicators shown at the bottom of the window and the state RGB buffer. From left to right: true speed, four ABS sensors, steering wheel position, gyroscope.

For the imitation learning bit, we used this library in a roundabout way. We analyze frames of thousands of preprocessed images on a CNN. The neural network then uses end-to-end learning. This means that the neural network will directly give us the commands to navigate the car. This is not a road detection module, which will then be analyzed by another program (most true autonomous driving systems are made this way). Here, the neural network takes the field matrix as input, and issues a command to be executed (turn left, turn right, continue straight ahead), without any intermediate program.

7.1.1 Data Collection for Imitation Learning

The first step in training our neural network is to create a dataset. It is about recording a set of images accompanied by their label . We will represent the possible actions with integers:

- 0 to indicate to go left
- 1 to indicate go right
- 2 to indicate to go straight

Thus, we will save a set of 3000 images in a folder, accompanied by a file labels.txt indicating on each line ¡Image path¿ label. We have 3 labels, so we save 1000 images of each label for the training set. For the testing set, we will save 600.

You have to create a train set , which will be used to train the network, and a test set , which will be used to evaluate its performance during training, to know when to interrupt it. Indeed, given the relatively low number of images that we use (3000), there is a risk of overfitting .That is to say that the network will lose in power of generalization to be better. in the special cases of the training set . This is a situation that we want to avoid, since we want to use our model subsequently in situations that it has not seen. The technique of interrupting training before convergence is called early stopping .

7.2 Deep Q Learning on the Car Racing Agent

7.2.1 Action selection

The action type is chosen through the epsilon-greedy policy mentioned previously. If a random number is lower than epsilon, the agent will choose an action randomly (exploration). However, if the random number is greater or equal than epsilon, the agent will choose the best action (exploitation) by sampling a batch (usually 32 or 64) of experiences saved in the experience replay memory.

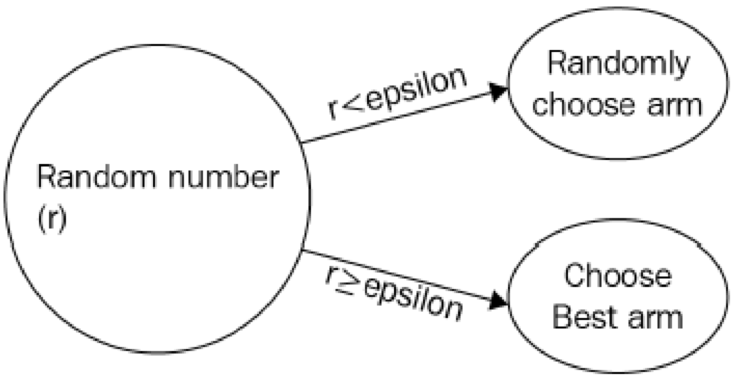


Figure 17: Epsilon greedy policy

7.2.2 Agent Simplification for Faster Training

Due to the difficulty in this project to obtain interesting results, the model was simplified to obtain faster convergence:

- An extra measure was implemented for the random actions obtained for exploration to give priority to accelerating actions. This decision was made because without it, the agent's random actions took a very long time to start learning the first steps. So, this initial "push" to the car makes it start of on a much better initial point and helps reduce the long training times.
- To further help reduce convergence time, and give the model only trains every n frames (n =3). This helps get more diverse experiences faster. This number is equal to the number of sequential frames used as input to the neural network.

- On the initial tests, if the car left the road and started driving randomly in the grass, the training would just continue for a long time before the episode was considered done wasting precious training time and saving a lot of unwanted experiences. For this reason, an early stop mechanism was implemented to detect early stops and begin a new training episode right away. It works by counting the consecutive number of frames (in this case, consecutive number every 3 frames) that the agent gets a negative reward. When this number reaches a certain maximum value, the episode is considered done early.

  This had the additional benefit of restricting the car to follow the road and stop skipping entire corners, a behavior that was observed previously.

- In the final model, the action space was also reduced from 12 to 5. This had a great impact in increasing the learning performance since it requires less parameters to be learned. This simpler action space was implemented based on [14].

- The car was observed to sometimes break when it almost reached the end of the track and therefore loose the rewards of the final tiles. To counter this behavior, a linear reward increment was done based on the number of frames in the current episode. This way, the last tiles give a higher reward than the initial tiles. This encourages the car to go further in the track, which also helps reduce the amount episodes with a lower score. *(This does not alter the output score, only the reward saved in the experience memory that is used for training the network)*

$$reward_{final} = reward_{initial} + 0.2 * num\_frames\_in\_episode$$

8 Conclusion

8.1 Imitation Learning

Our network recognizes the shapes to keep the car on the desired path. It's a sort of classifier that just indicates whether the car is in the right position, too far to the right or too far to the left. We then send this command to the simulator. All of this is done in real time.

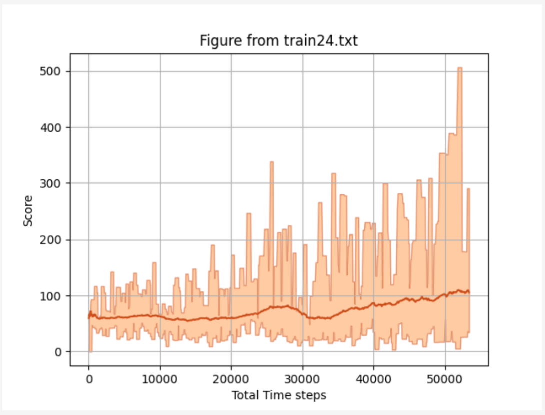
Behavioural Cloning though has a few disadvantages, and we can see them here in this notebook.

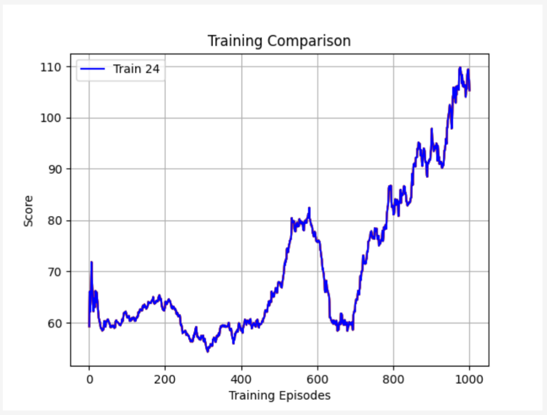
- We need to manually accelerate and decelerate, and we can only accelerate till a certain limit, because beyond that, the car will spin out of control and go outside in the patch of grass.
- Since while training we never leave the track, the car has no way of coming back to the road after it has left the track and is into the grass.
- Here we only have a train set of 3000 and validation set of 600, but we tried increasing the sizes of these by a magintude of 10 (30,000 and 6000), but because of the substantial increase in the size of the dataset, the error while generating the dataset also shot up, which turned out to be a very bad dataset for out neural net.
- Also, because we were well within the tracks, the car has no data on cases in which it goes out by accident.
- A possible remedy for this is preprocessing the data in such a way that the dataset has images of car coming in, but not going out.

8.2 Reinforcement (Deep Q) Learning

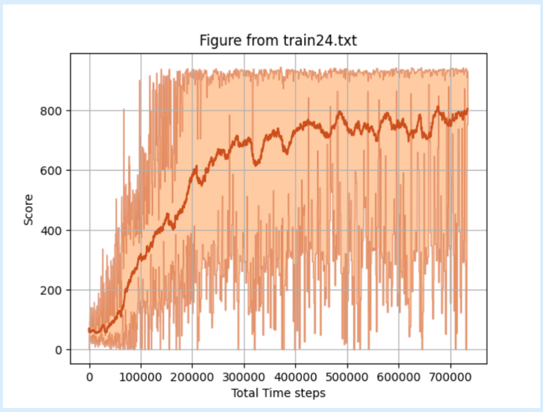
Below we see the progression of scores with each time-step and training episode

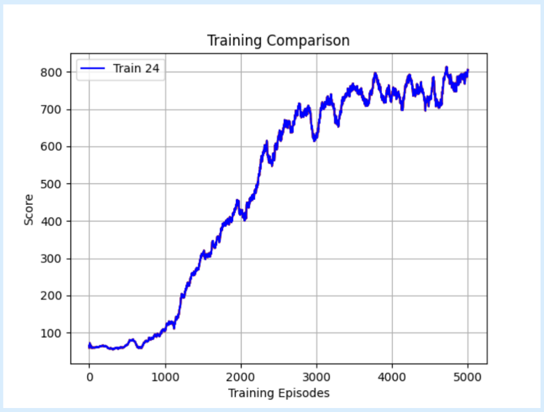
The first set of graphs show the average score across 1000 training episodes



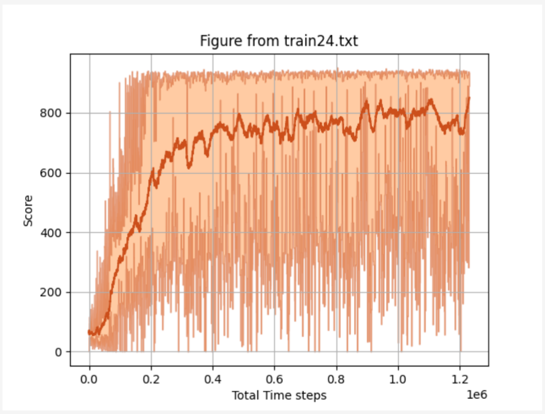


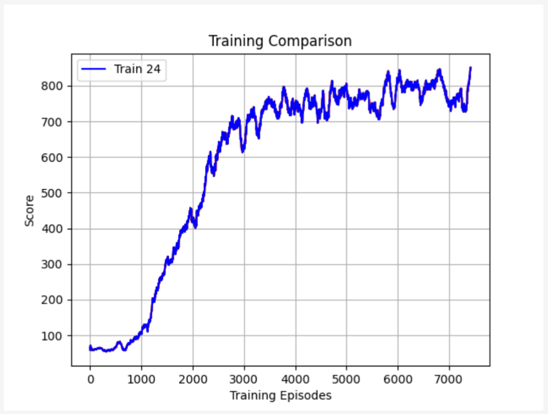
The next set of graphs show the average score across 5000 training episodes



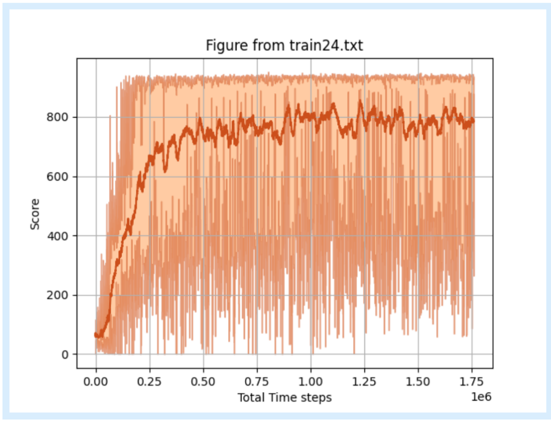


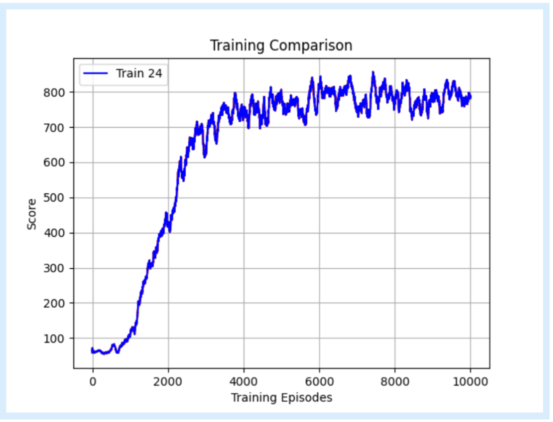
The next set of graphs show the average score across 7500 training episodes





The next set of graphs show the average score across 10K training episodes



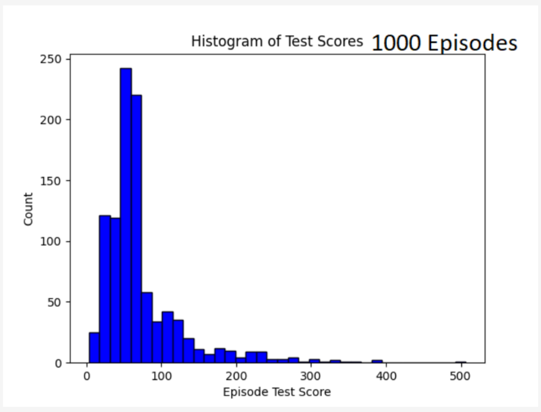


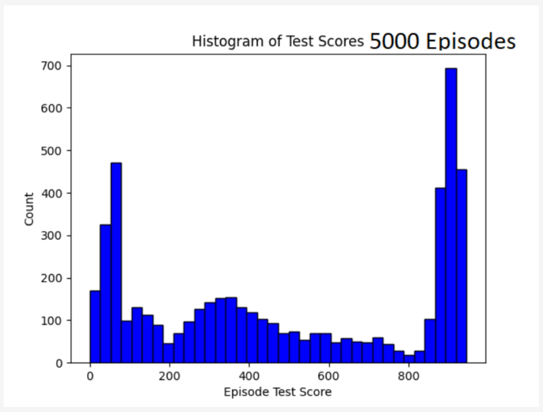
As we can clearly see, there is a gradual increase in the score

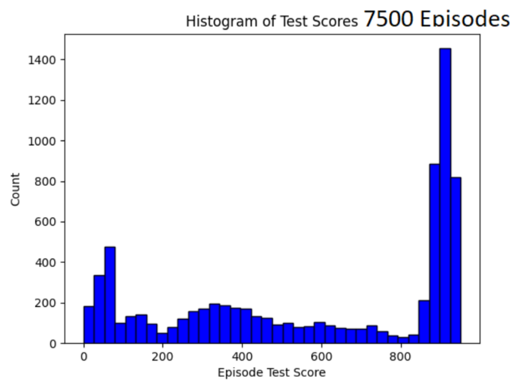
- The first 1000 episodes we barely touch the score of 110
- By the time 5000 episodes are run though, we just touched the score of 800, averaging around 700s
- By 7500 we are averaging around the 800 score
- By 10000 episodes we see that the score is slightly above 800
- If we would have let it run for 15000 epochs or so, we're confident we would've acheived the game winning condition i.e. (average of 900 score over the last 100 episodes

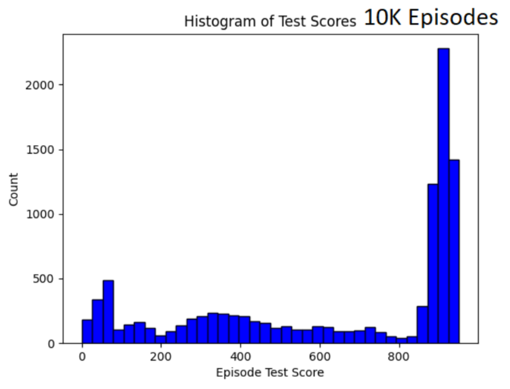
Below we see the histogram of test scores for 1000, 5000, 7500, and 10000 episodes each.

- For the first 1000 episodes the counts of scores are concentrated between 0 & 100
- For 5000 episodes this shifts to 3 areas, around 800, between 0 & 100 and between 350 and 450
- For 7500 episodes this shifts from previous and the counts near 800+ scores rising over 1400 and other scores between 0 & 100 and around 400 diminishing
- For 10000 episodes we clearly see that over 6000 of the 10000 episodes result in a score of 800+









8.3 Future Work

8.3.1 Imitation Learning

Acceleration Control :

The control of the car is not total here: the network only controls the lateral acceleration (the right / left direction) of the car, but does not control the acceleration (therefore the speed). The problem is that it is impossible to guess the speed of the car by looking at a single image, so it cannot control the acceleration to maintain a suitable speed.

- Use the speed bar which is under the image (the one that we have hidden). But the direction bar should be kept hidden, which misleads the direction classifier;
- Give the network several successive images, instead of just one. In this way, the network could deduce the speed of the car
- Ask the network to control only the speed, and not the acceleration (it is then necessary to code a feedback system that will maintain the requested speed): this approach is not really end-to-end but can be simpler if we has correct external data on the current speed (one could modify the environment to provide it in addition to the state).

Data Increase :

The best way to improve the performance of classifiers is to increase the amount of data. But here it is quite long because the data has to be saved while playing the game manually.

- One way to artificially increase the amount of data is called data augmentation.
- It is a question of carrying out transformations to the images, which will not modify the labels (or will modify it in a determined way). One can for example take the image symmetrical with respect to the vertical axis.
- The left / right labels will then be inverted, and the amount of data is multiplied by 2 immediately.
- Other possible transformations may be to distort the image a little or to modify the colors slightly (here the colors are fixed in the environment, so it will surely be less effective here than on real images).

8.3.2 Reinforcement Learning

Deep Q Learning :

Deep Q Learning is a very interesting and fairly recent area of study. Many new adaptations and techniques are surely going to happen for deep Q learning in the coming years. To further try and improve the results presented on this report, the following ideas could be explored:

- Testing Deep Q Learning variants such as Dueling Deep Q Learning or Double Deep Q.
- Testing different optimizers such as the RMSProp.
- Testing different network architectures. Maybe adding an extra convolutional layer or an extra dense layer can help the network to reach higher scores more consistently.
- Testing different hyperparameters, maybe adding having more memory available on the experience replay could help the model recall more experiences. Or, perhaps trying different learning rate exponential decay parameters could speed up the training more.
- Trying different weight regularization parameters.
- Exploring cloud computing, a technology that allows for models to be trained in the cloud using more powerful computation.

Bibliography

[1] https://github.com/openai/gym

[2] http://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

[3] https://github.com/jperod/AI-self-driving-race-car-Deep-Reinforcement-Learning

[4] https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3ca

[5] https://cdancette.fr/2018/04/09/self-driving-CNN/

[6] https://github.com/Anaconda-Platform/nb_conda_kernels

[7] https://www.analyticsvidhya.com/blog/2018/12/
guide-convolutional-neural-network-cnn/

[8] https://medium.com/@SmartLabAI/a-brief-overview-of-imitation-learning-8a8a75c44a9c

[9] https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add

[10] https://github.com/openai/gym/blob/master/gym/envs/box2d/car_racing.
py

[11] https://www.davidsilver.uk/teaching/

[12] https://github.com/pekaalto/DQN

[13] http://files.davidqiu.com//research/nature14236.pdf

[14]