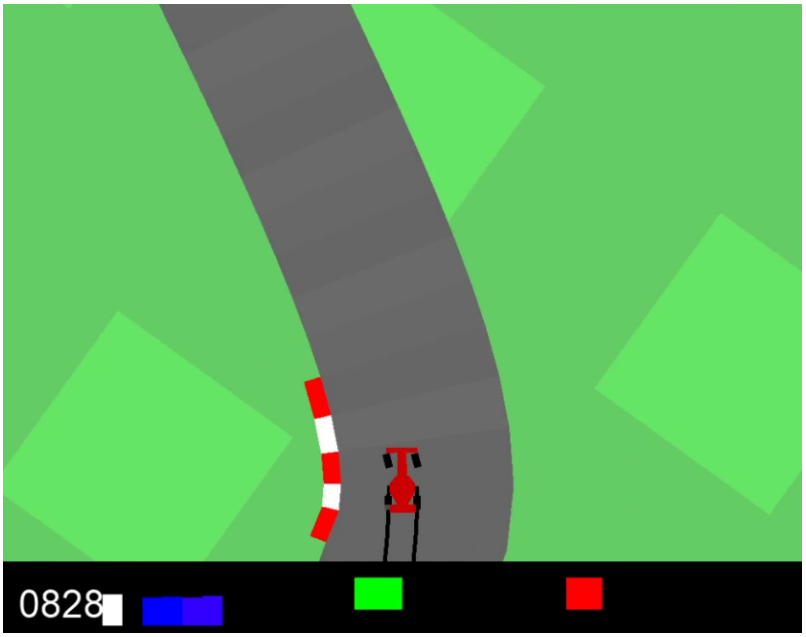
Developing an Intelligent Self-Driving Race Car using Deep Reinforcement Learning

Final Project - Intelligent Systems

1st Semester 2019/2020



Teachers
João Sousa
Susana Vieira

Written by:
Joaquim Pedro Rodrigues - 84282 - j.pedro.rodrigues@tecnico.ulisboa.pt
Manuel Adegas Henriques - 84295 - manuel.henriques@tecnico.ulisboa.pt
Vasco Mendes - 84359 - vm_vasco@hotmail.com

Date: December 23, 2019
Instituto Superior Técnico

Contents

1 Introduction

1.1 Problem Statement

In this project, a python based car racing environment is trained using a deep reinforcement learning algorithm to perform efficient self driving on a racing track. A deep Q learning algorithm is developed and then used 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. This score is given by the gym environment and can be seen on 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. In this project, this goal was surpassed having obtained an average score of 905 over the last 100 runs. Therefore, we successfully solved the environment.

1.2 Approach

In this report, we first explain the theory we learned and found relevant about reinforcement learning, Q-Learning and Deep Q-Learning and Neural Networks. Next, we explain our implementations of deep Q learning agents for autonomous driving. Finally, we explore different hyperparameters and network architectures combinations. The obtained results are analyzed.

A video of the self racing car's performance can be seen on youtube.

Video of the agent's final performance: `https://youtu.be/jbdjhoDT41M`

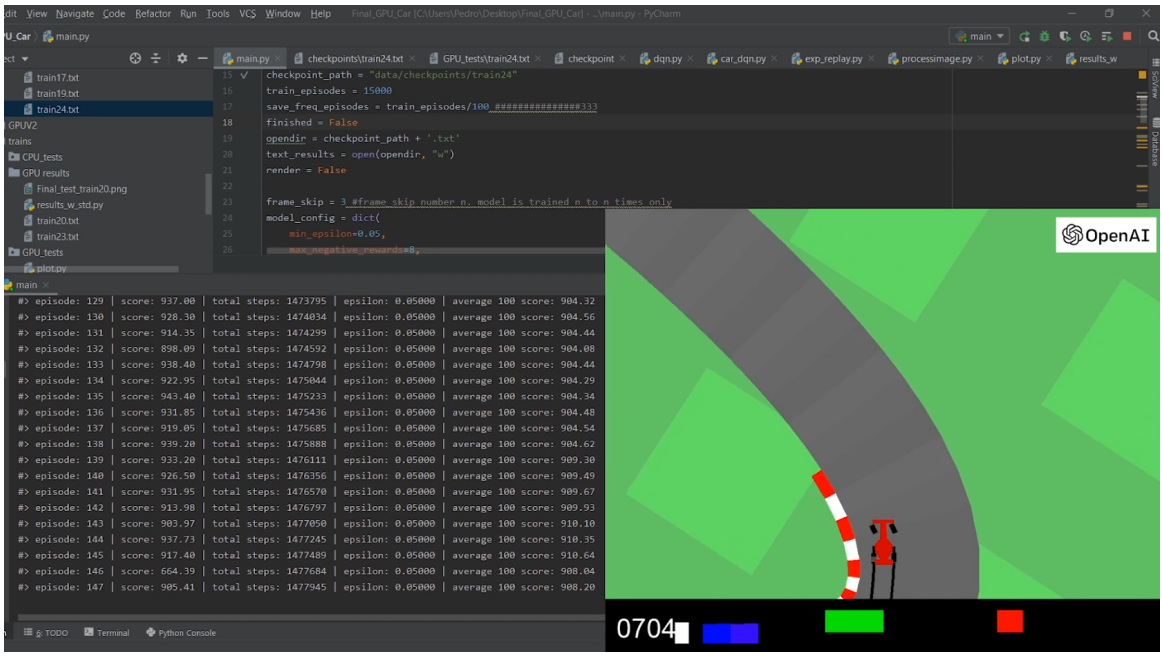


Figure 1: Screenshot of the video of the final test performance

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). [1]

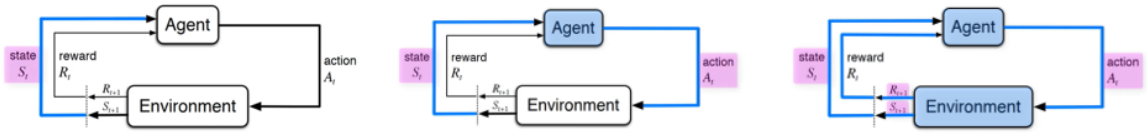


Figure 2: Reinforcement Learning Process

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 $\epsilon - greedy$ $(0 \leq \epsilon \leq 1)$ 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 $\pi(s)$ 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 $\pi(s)$. 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:

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

Where $a = \pi(s)$ is chosen (i.e. actions given by the policy). And the expectation is taken over: $s_{t+1} \sim P_{a_t}(s_t, s_{t+1})$. 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.

$$V(s) := \sum_{s'} P_{\pi(s)}(s, s')(R_{\pi(s)}(s, s') + \gamma V(s')) \tag{2}$$

$$\pi(s) := argmax_a \left\{ \sum_{s'} P(s'|s, a)(R(s'|s, a) + \gamma V(s') \right\} \tag{3}$$

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.

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a) \tag{4}$$

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:

$$Q(s, a) \rightarrow \gamma Q(s', a) + \gamma^2 Q(s'', a).....\gamma^n Q(s''^{\ldots n}, a) \tag{5}$$

Adjusting the value of γ will diminish or increase the contribution of future rewards.

The weight for a step from a state Δt steps into the future is calculated as $\gamma^{\Delta t}$, 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 Δt

The algorithm, therefore, has a function that calculates the quality of a state-action combination:

$$Q : S \times A \rightarrow R \tag{6}$$

Before learning begins, Q is initialized to a possibly arbitrary fixed 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 Deep Reinforcement Learning

3.1 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. [2]

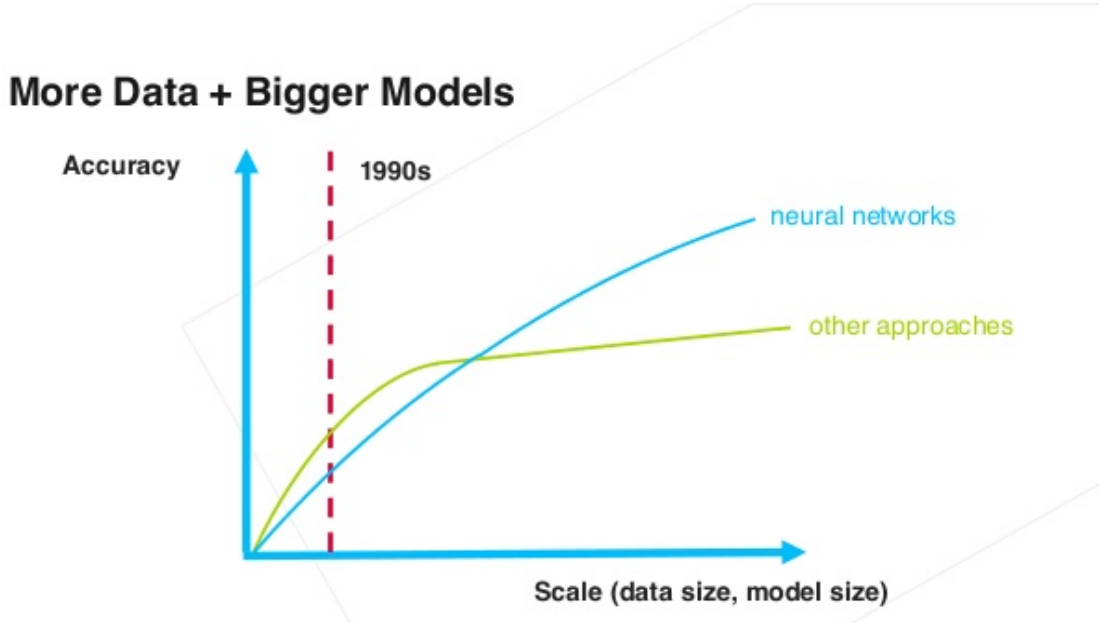


Figure 3: Deep Learning VS Older Algorithms

3.2 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 in finite 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).

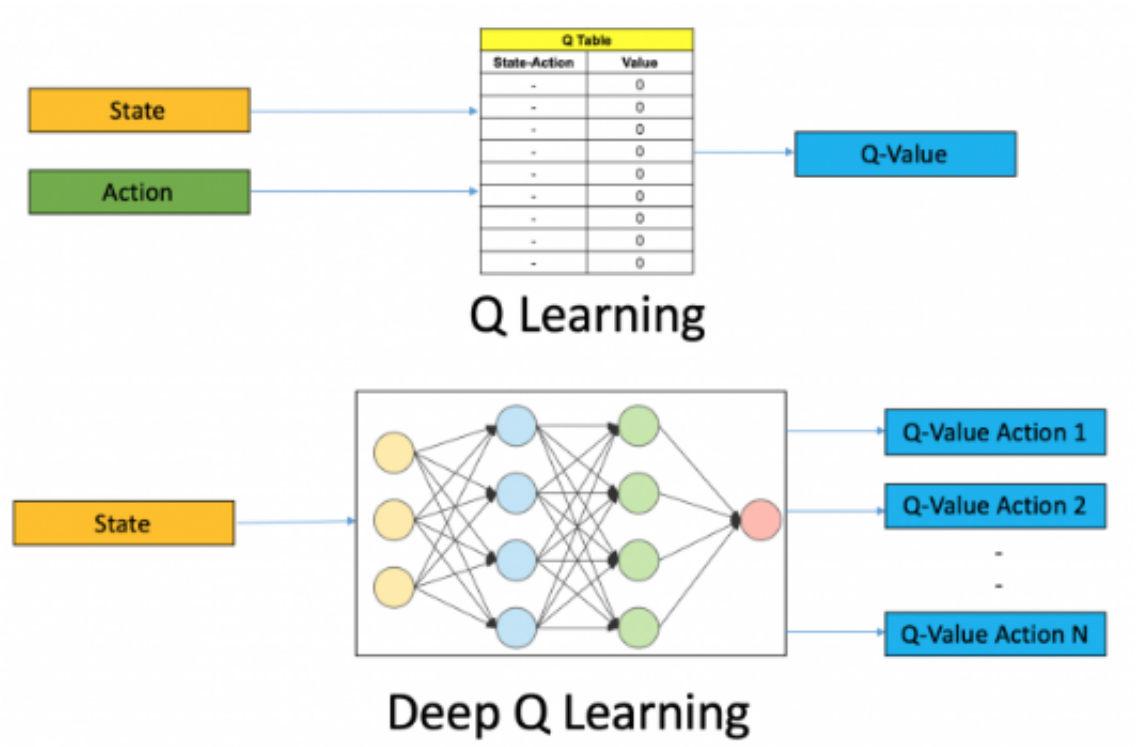


Figure 4: Q-Learning VS Deep Q-Learning

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:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{8}$$

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')+$\gamma \max_{a'} Q_{k(s',a')}$
    $\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta E_{s' \ P(s'|s,a)}[(Q_\theta(s,a) - target(s'))^2]|_{\theta=\theta_k}$
    $s \leftarrow s'$

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):

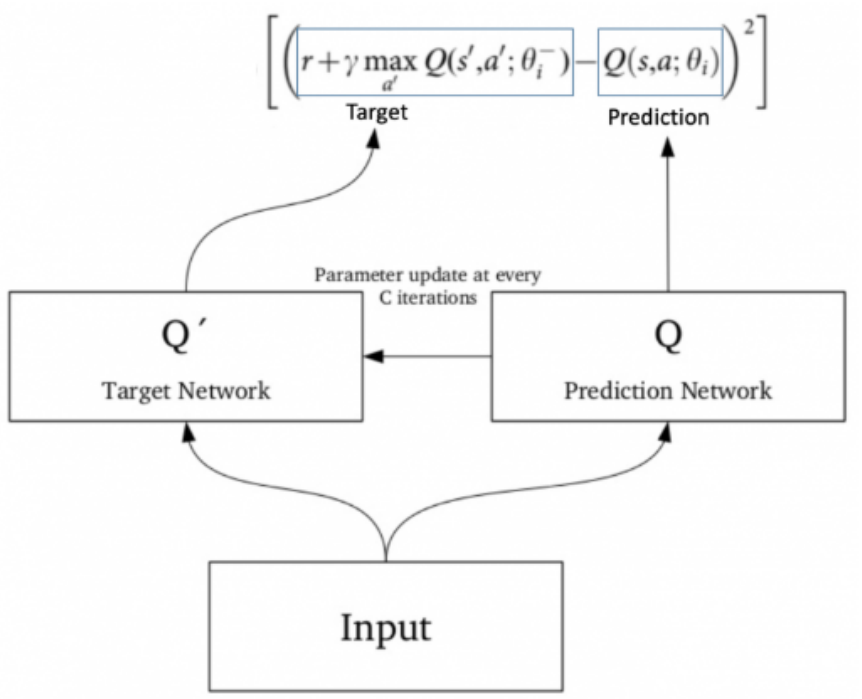


Figure 5: Two Neural Networks system implementation

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

4 Deep Q Neural Networks

4.1 Hyperparameters of a Deep Q Learning Network

In any deep Q learning neural network, specific deep Q learning parameters are important to tune. On this project, we experimented with some parameters within the following value ranges.

Parameter	Value Range	Description
min ϵ	[0.05, 0.1]	Minimum value of epsilon, the final exploration rate of actions
ϵ	[1]	Initial value of epsilon, the initial 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.

4.2 Input Data Preprocessing

Originally, the observations obtained from the environment were colored RGB images with a 96x96 shape. Since these observation images are the training inputs for the neural network, they were preprocessed 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.

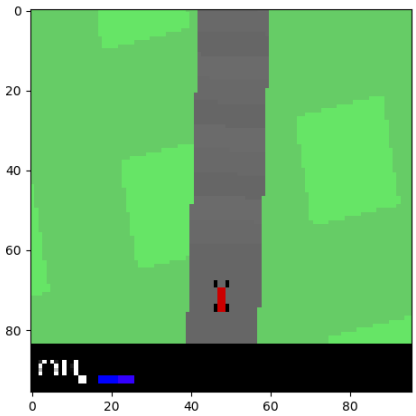


Figure 6: Original observation image

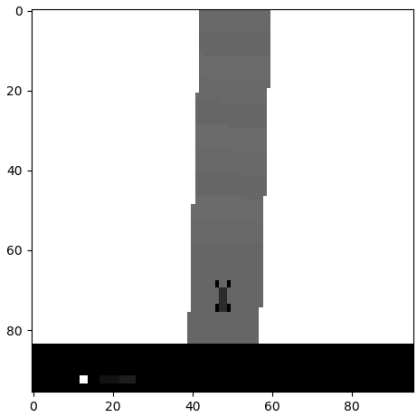


Figure 7: Observation image after preprocessing

4.3 Neural Network Architecture for Deep Q Learning

4.3.1 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.

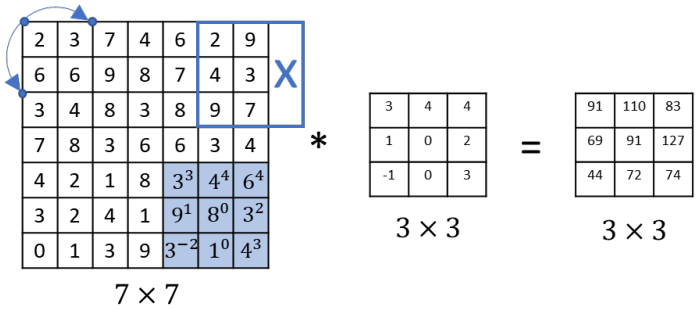


Figure 8: Example of a convolution using a 3x3 kernel with stride 2x2 [3]

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...

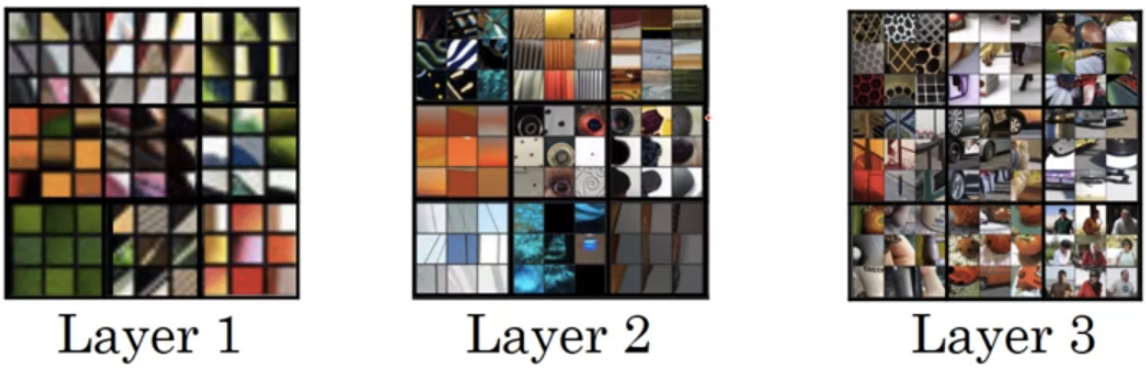


Figure 9: Visualization of CNN layers [3] [4]

4.3.2 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. [5]

4.3.3 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. [6]

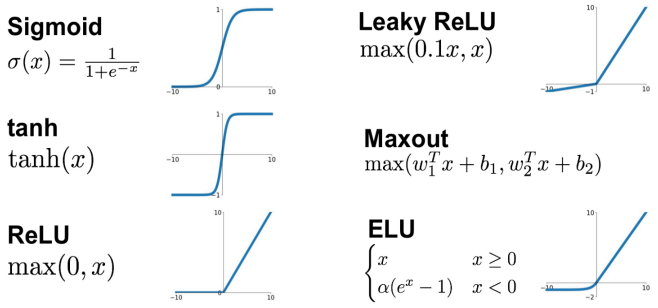


Figure 10: Commonly used activation functions

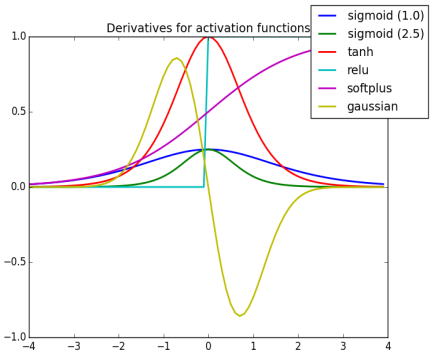


Figure 11: Derivative of common activation functions

For this implementation, the ReLU (Rectified 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. [6]

Considering a Neural Network with the following form:

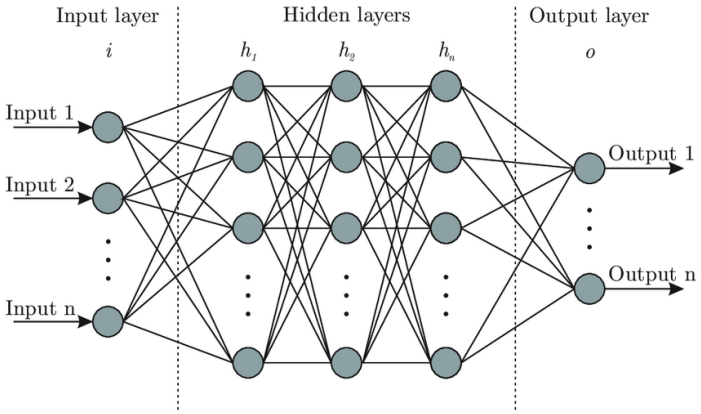


Figure 12: Example of a Deep Neural Network

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

$$a_0^{(1)} = ReLU(w_{0,0}a_0^{(0)} + w_{0,1}a_1^{(0)} + ... + w_{0,n}a_n^{(0)} + b) \tag{9}$$

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:

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

4.3.4 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:

$$L = (Q_{a_1} - Q^*_{a_1})^2 + (Q_{a_2} - Q^*_{a_2})^2 + ... + (Q_{a_n} - Q^*_{a_n})^2 \tag{11}$$

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

This 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:

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w^{(1)}} \\ \frac{\partial L}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial L}{\partial w^{(n)}} \\ \frac{\partial L}{\partial b^{(n)}} \end{bmatrix} \tag{12}$$

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

$$\frac{\partial L}{\partial w^{(k)}} = \frac{\partial z^{(k)}}{\partial w^{(k)}} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial L}{\partial a^{(k)}} \tag{13}$$

$$\frac{\partial L}{\partial b^{(k)}} = \frac{\partial z^{(k)}}{\partial b^{(k)}} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial L}{\partial a^{(k)}} \tag{14}$$

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

4.3.5 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. [8]

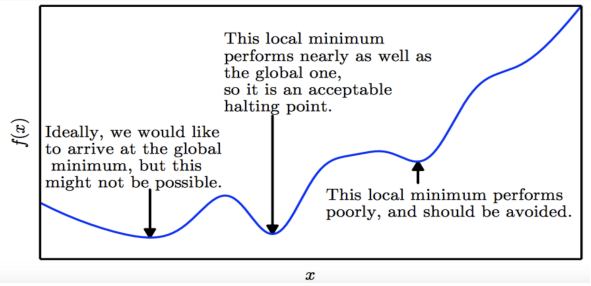


Figure 13: Optimization curve

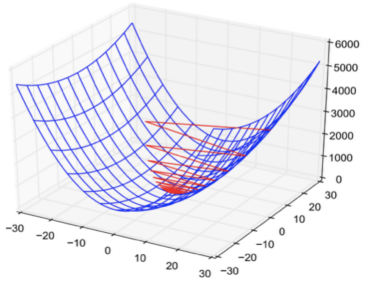


Figure 14: Optimization surface

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. [9]

- alpha: 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: The exponential decay rate for the first moment estimates. weighted average for dw

- beta 2: The exponential decay rate for the second moment estimates. weighted average for dw^2
- epsilon: Very small number to prevent division by zero in the implementation

Adam optimization algorithm [9]:
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 final tests according to equation:

$$decayed\ \alpha = \alpha * decay\ rate^{\frac{global\ step}{decay\ step}} \tag{15}$$

with decay rate = 0.8 and decay steps = 200000. This was implemented within the tensorflow framework by following [10].

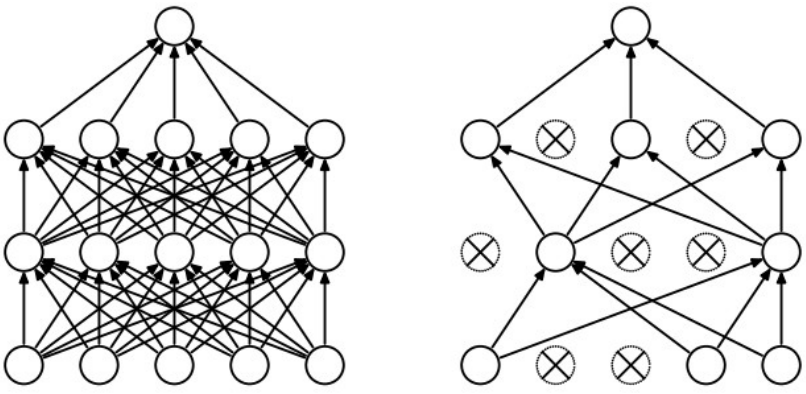
4.3.6 Overfitting Prevention Methods: Weight Regularization

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 fit 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. [9]

4.3.7 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.



(a) Standard Neural Net (b) After applying dropout.

Figure 15: Example of three 40% dropout layers

5 Autonomous Racing Car Driver Agent

5.1 Methods

Before approaching the racing car environment, some of the more simple open ai gym environments were tested such as the CartPole, MountainCar, Breakout and Lunar Lander [Figure 16]. After that, a similar code was made for the RacingCar environment. This helped learn how to code a python deep Q learning algorithm but this implementation was taking a very long time to train when changed to this new environment.

For this reason, the code developed for this project is based on [11] which is a simple tensorflow based implementation of the deep Q learning algorithm on the open ai gym car racing environment. This implementation followed the parameters shown on article [12] but the neural network architecture was reduced and it is easily able to run on a CPU. This helped solve the high computational demands.

However, to further improve the computation capacity, the tensorflow framework used before was updated to support tensorflow-gpu (version 1.14), a python library that enables a faster gpu computing. This was possible by using a Nvidia GTX 1050 Ti GPU. After this, the training speed became a lot faster and allowed for larger training sessions.

Afterwards, in order to test different configurations, several .txt files with the training information were written after each training session. Next, different configurations or parameters and neural network architectures were tested and the training information was plotted to try to obtain the best racing car performance measured as the average score over a period of 100 episodes.

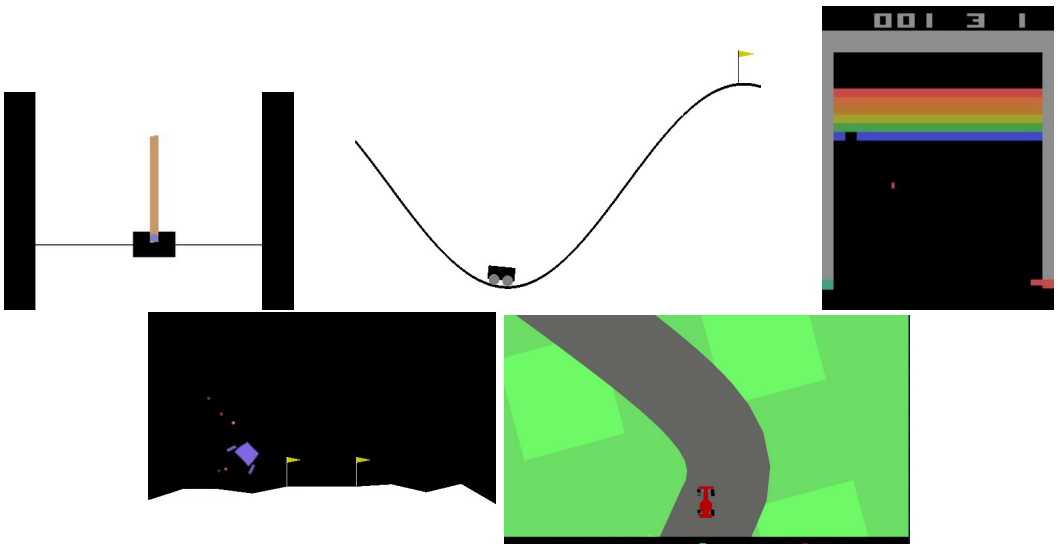


Figure 16: All the open ai gym environments tested for this project

5.2 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 96x96 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. [13]

5.3 Deep Q Learning on the Car Racing Agent

5.3.1 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.

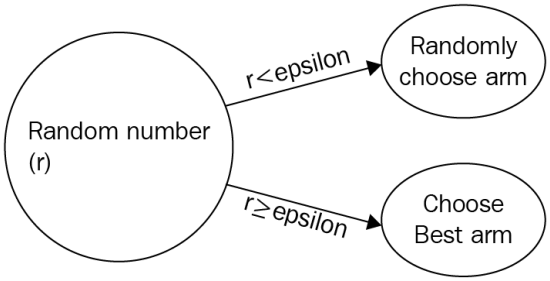


Figure 17: Epsilon greedy policy

5.3.2 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 behaviour 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 behaviour, 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*)

$$reward_{final} = reward_{initial} + 0.2 * num\_frames\_in\_episode \tag{16}$$

5.3.3 Simple Deep Q-Network (CPU trained)

For our initial implementation, the model used all possible 12 actions, was trained every 3 frames, already had the early stopping mechanism and the prioritized acceleration random action. The best 2 models on this initial implementation obtained a maximum training average over 100 episodes of 700. These configurations are shown on figures 28 and 29. The difference in the two here lies in the number of neurons of the first dense layer. The model in 28 had more neurons than its flattened input (400), so that might be the cause for the oscillatory behaviour when training. In comparison, the model in 29 was more consistent and displayed a more or less constant growing rate after the initial 150000 time steps. Both models seem to continue rising, however, the training was taking too long (30h+) to continue the training. Also, during the training, the variance of results was consistently high. Even when the average reached values near 700, the model was still consistently having low scores between the high scores.

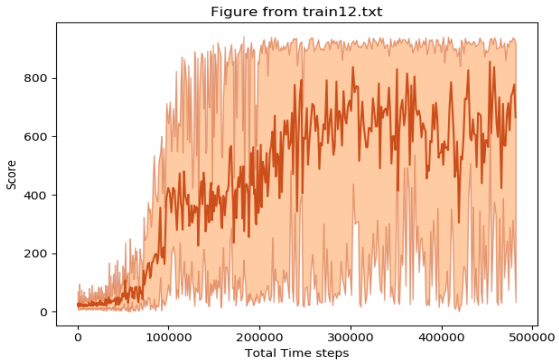


Figure 18: Second best model obtained with simple CPU deep Q-network: Dense layer neurons = 512

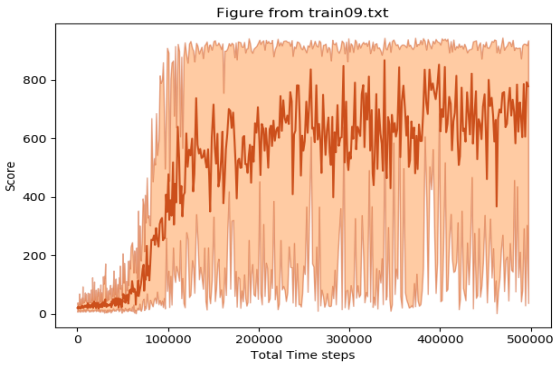


Figure 19: Best model obtained with simple CPU deep Q-network: Dense layer neurons = 256

Some of the tests done on this model, from worst to best results:

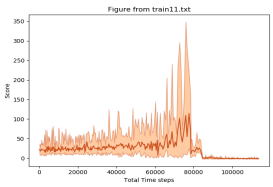


Figure 20: $\alpha = 0.01$

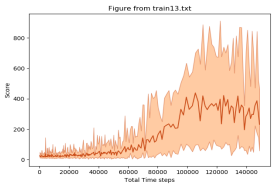


Figure 21: Dense layer neurons = 64

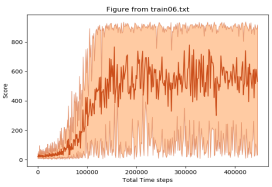


Figure 22: $\alpha = 0.0005$

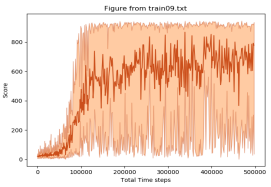


Figure 23: $\alpha = 0.001$, Dense layer neurons = 256

5.3.4 Final Deep Q-Network (GPU trained)

After the conda environment and the initial script was updated to support gpu training with the tensorflow framework using a CUDA GPU, the AI could train much faster. It allowed it to easily reach the 1 million time steps mark in less than 24 hours. The following improvements to the training have then been done:

Action space reduction from all 12 actions to selected 5:

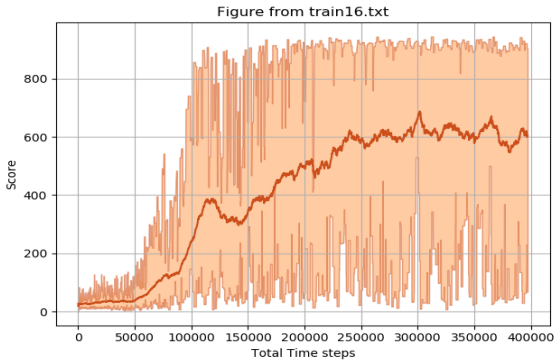


Figure 24: Using 12 actions (400000 time steps)

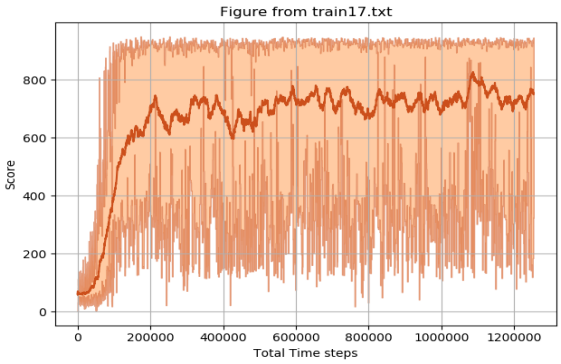


Figure 25: Using 5 actions (1200000 timesteps)

Learning rate decay using continuous 0.8 decay rate over 200000 steps:

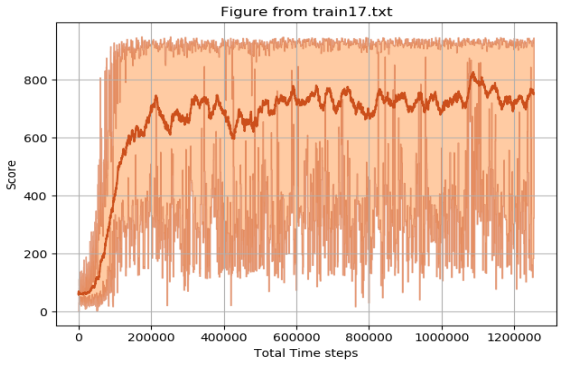


Figure 26: With constant learning rate (1200000 time steps)

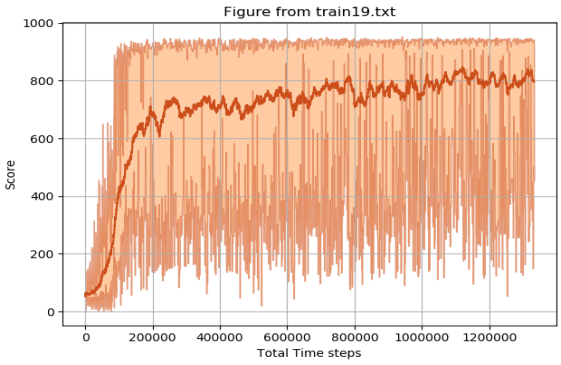


Figure 27: with decaying learning rate (1300000 time steps)

Implementing the linear reward increment according to the episode length

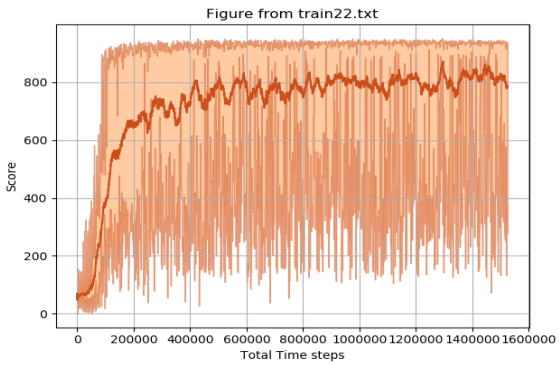


Figure 28: Without linear reward increase (1.6 million timesteps)

Figure 29: With linear reward increase implemented. $[reward = reward + 0.2 * num\_frames\_in\_episode]$ (1.6 million timesteps)

When testing configuration number 24, the training results were very satisfactory as seen on figure 30, thus it was the one chosen for the final testing. It displayed a solid learning curve that started to stagnate around the 850+ mark.

Figure 30: Final Training plot on 1.6 million timesteps (± 30 hours training on Nvidia GTX 1050 ti GPU). It has the linear increasing rewards implementation, an exponential learning rate decay of 0.7 and 400 dense layer neurons. The checkpoint with the maximum average 100 training score (872) was the one used for testing.

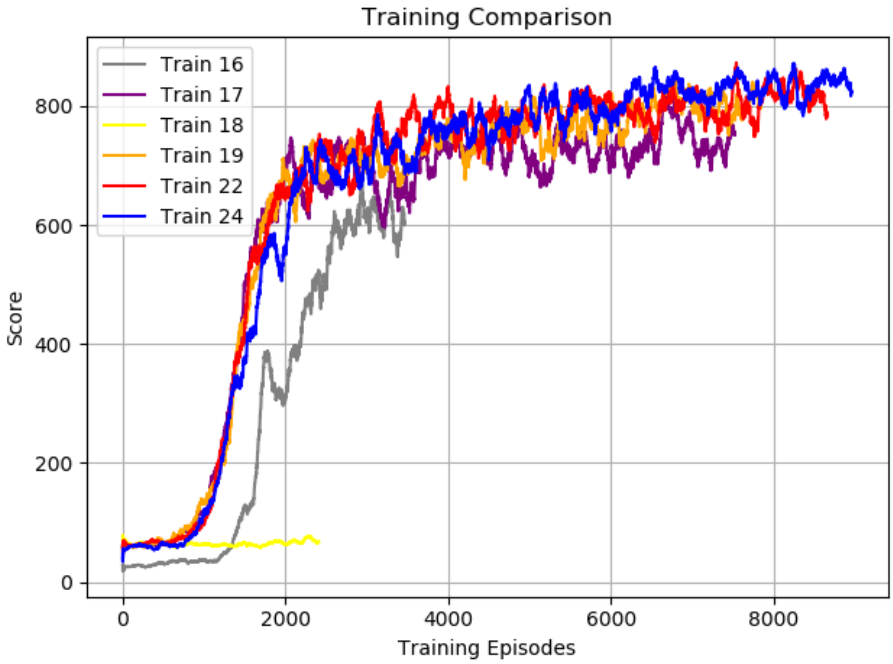


Figure 31: Comparison of some of the last training sessions. A table with all of the different training sessions done is attached at the end of this report.

6 Conclusion

6.1 Final Testing Results Discussion

Overall, in this report we tried a lot of different configurations and architectures. The different configurations, ideas and simplifications that we implemented over the last month have been continuously giving us higher and better performance scores.This made it possible for us to reach the score of 900 required to consider this problem solved by OpenAI's deep learning Gym library .

6.1.1 Final Test Results

To test the performance according to the OpenAI gym leaderboard [15], the trained model should be tested and its average over 100 episodes should be measured along with the standard deviation. Following these test guidelines on 150 episodes, a final score of 905.38 +- 23.82 was obtained by calculating the average of the last 100 episodes mean score right after the 100 episodes mark.

With this score, we are proud to have successfully solved the CarRacing-V0 environment. The group is very pleased with these results. Not only was the deep Q learning algorithm successfully implemented, but many different configurations and parameters were tested, which was a great learning opportunity for us.

Video of the agent's final performance: `https://youtu.be/jbdjhoDT41M`

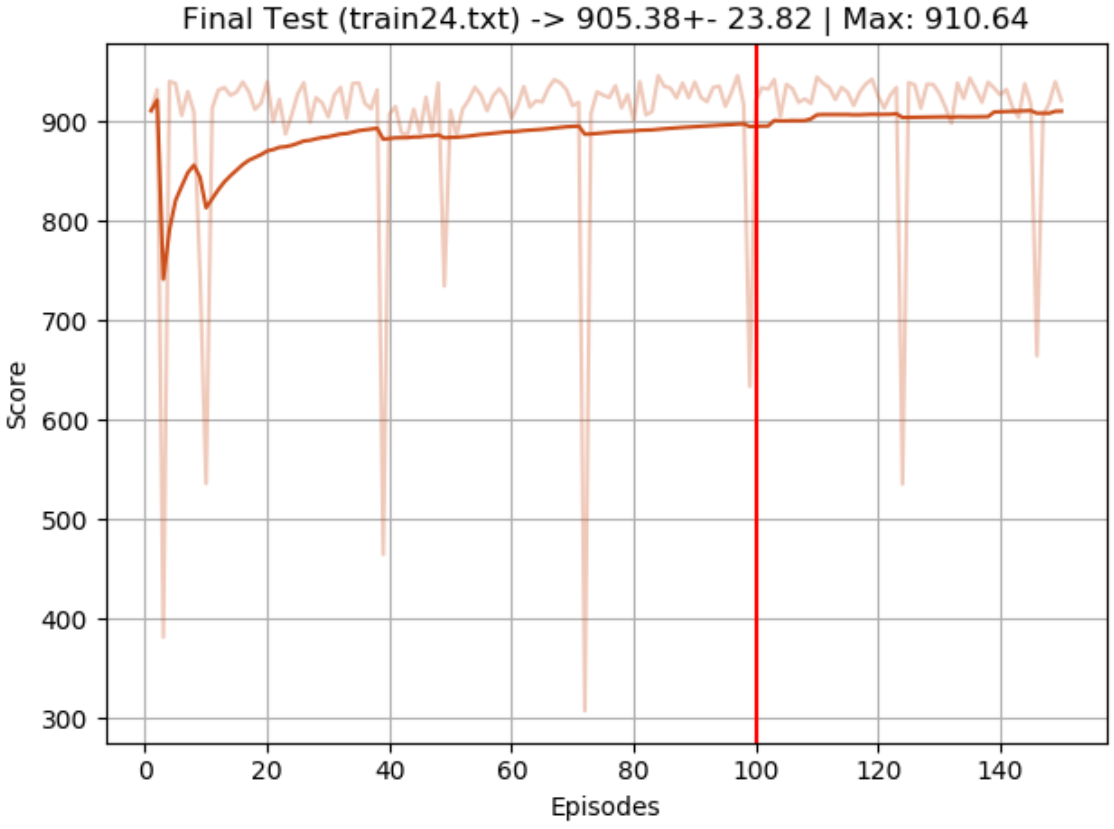


Figure 32: Final performance test on the GPU trained model (train24.txt). The more defined line represents the average over the last 100 episodes. The faded line represents the individual scores obtained in this 150 episodes test.

In this test, the agent also proved to be robust. The number of episodes where it obtained a score around the 900 mark is much higher than the number of episodes it reached a lower score as seen on figure 33.

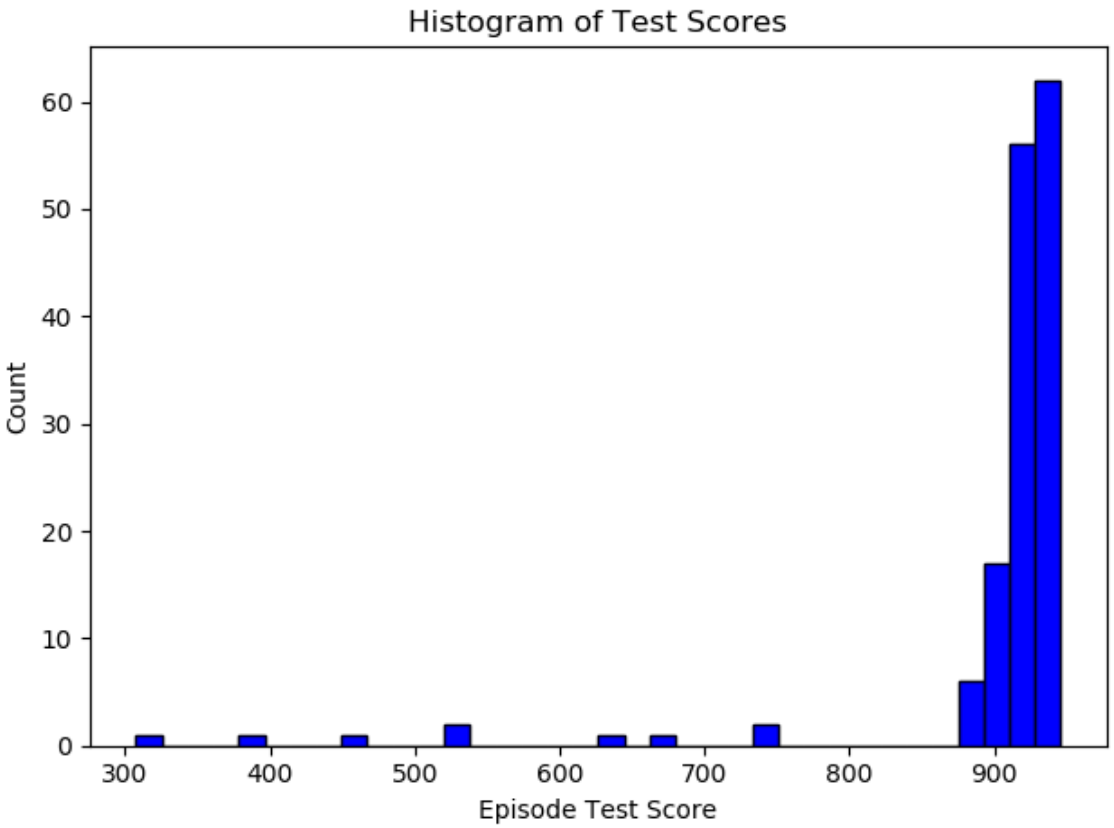


Figure 33: Histogram of the Test Scores

6.2 Future Work

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 [16]
- Testing different optimizers such as the RMSProp, the optimizer chosen in [12].
- 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.

A How to run code

To run this deep Q learning implementation on the racing car environment, after installing the necessary packages and the conda environment is ready, run *main.py* after selecting *load_checkpoint* true or false. When false, the agent will train and save checkpoints. A total of 7 python files were made:

1. *main.py*

   This is the main file. Allows to choose main parameters and choose to load previously trained checkpoints. Works by calling the other files and calling the main training loop.

2. *dqn.py*

   The dqn file is a script that implements a general deep Q learning agent that works on multiple openai environments. It creates both the target and estimate neural networks, defines the loss function and optimizer. It also handles the epsilon-greedy policy for action decision.

3. *car_dqn.py*

   This file works by adapting the dqn file to work to the specific car racing environment.

4. *exp_replay.py*

   This is the file responsible for the experience replay mechanism. It saves the states, actions, rewards and consequent states.

5. *processimage.py*

   This file is responsible for preprocessing the input 96x96 image and removing all the unwanted information to ensure the maximum training performance by enhancing input features.

6. *plot.py*

   Reads .txt training files and plots data.

7. *results_w_std.py*

   Reads .txt test files and plots data with final results and standard deviation.

B Actions

The actions used were discretized. Each actions is a vector with 3 elements.

Action Type	Column number	Range
Steering	1	[-1, 0 or 1] (-1 is left and 1 is right)
Accelerator	2	[0 or 1]
Brake	3	[0 or 0.5]

All Actions	
Action Number	Action Array
1	[-1, 1, 0.5]
2	[-1, 1, 0]
3	[-1, 0, 0.5]
4	[-1, 0, 0]
5	[0, 1, 0.5]
6	[0, 1, 0]
7	[0, 0, 0.5]
8	[0, 0, 0]
9	[1, 1, 0.5]
10	[1, 1, 0]
11	[1, 0, 0.5]
12	[1, 0, 0]

Selected Actions	
Action Number	Action Array
1	[-1, 0, 0]
2	[0, 1, 0]
3	[0, 0, 0.5]
4	[0, 0, 0]
5	[1, 0, 0]

C Deep Q Neural Network Architecture

Input Shape	Final DQN (GPU trained)
(64, 96, 96, 3)	Convolutional 2D, 8 7x7 kernels, stride 4, ReLU activation, Weight regularization
(64, 23, 23, 8)	Max Pooling 2D, 2x2 kernels, stride 2
(64, 12, 12, 8)	Convolutional 2D, 16 3x3 kernels, stride 1, ReLU activation, Weight regularization
(64, 10, 10, 16)	Max Pooling 2D, 2x2 kernels, stride 2
(64, 5, 5, 16)	Flatten
(64, 400)	Fully Connected, 400 neurons, ReLU activation, Weight regularization
(64, 256)	Output: Fully Connected, 5 neurons, Linear activation, Weight regularization

D Hyperparameters

Parameter	Value Range	Description
min ϵ	0.05	Minimum value of epsilon, the final exploration rate of actions
ϵ	1	Initial value of epsilon, the initial exploration rate of actions
ϵ decay steps	100000	The number of total time steps it takes to reduce ϵ to its final value min ϵ
min exp size	1000	Minimum number of experiences saved , these are done at the beginning before the training
exp capacity	150000	Maximum number of experiences saved, when this capacity is full, old experiences are erased for new ones.
target network update freq	1000	How often the prediction network parameters are copied to the target network in time steps
max negative rewards	8	Maximum number of consecutive negative rewards before the script considers it an early ending
batchsize	64	Number of training cases over which each loss function minimization is computed
num frame stack	3	Number of frames used in sequence input to the neural network, also corresponds to how often the network is trained
gamma	0.95	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
Learning rate	0.001	how much we are adjusting the weights of our network with respect to the loss gradient (Initial value)
Learning rate decay	0.7	70% reduction of the learning rate's starting value every 200000 steps (exponential decay)
Learning rate decay steps	200000	Steps needed to reduce the learning rate to 70% of it's starting value (exponential decay)

E Training Sessions Log

Train N	Learning Rate	TNUF	BS	exp cap	MNR	ϵ decay N	γ	Regularization	Convolution	Dense
1	0.0005	500	32	50000	12	50000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 128 (ReLU) and 12 (Linear)
2	0.0005	500	32	50000	12	**100000**	**0.99**	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: **384** (ReLU) and 12 (Linear)
3	0.0005	500	32	50000	12	100000	0.99	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: **128** (ReLU) and 12 (Linear)
4	0.0005	500	**64**	50000	12	100000	0.99	1e-6 Weight Regularization	**3** (ReLU and w max pooling))	2: 128 (ReLU) and 12 (Linear)
5	0.0005	500	64	50000	**5**	100000	0.99	1e-6 Weight Regularization	**2** (ReLU and w max pooling))	2: 128 (ReLU) and 12 (Linear)
6	0.0005	**5000**	64	50000	**8**	100000	**0.95**	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 128 (ReLU) and 12 (Linear)
6	0.0005	5000	64	**40000**	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: **256** (ReLU) and 12 (Linear)
7	0.0005	**1000**	64	**100000**	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 256 (ReLU) and 12 (Linear)
8	**0.1**	1000	64	**150000**	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 256 (ReLU) and 12 (Linear)
9	**0.001**	1000	64	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 256 (ReLU) and 12 (Linear)
10	**0.005**	1000	64	**200000**	8	**150000**	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 256 (ReLU) and 12 (Linear)
11	**0.01**	1000	64	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 256 (ReLU) and 12 (Linear)

TNUF - Target Network Update Frequency; BS - BatchSize; MNR - Maximum Negative Rewards;

Training Sessions Log - Continuation

Train N	Learning Rate	TNUF	BS	exp cap	MNR	ϵ decay N	γ	Regularization	Convolution	Dense
12	**0.001**	1000	64	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: **512** (ReLU) and 12 (Linear)
13	0.001	1000	64	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: **64** (ReLU) and 12 (Linear)
14	0.001	1000	**128**	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: **256** (ReLU) and 12 (Linear)
15	0.001	1000	**1000**	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 256 (ReLU) and 12 (Linear)
16 **(GPU)**	0.001	1000	64	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 256 (ReLU) and 12 (Linear)
17 (GPU)	0.001	1000	64	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 256 (ReLU) and **5 (Linear)**
18 (GPU)	0.001 **trying AdaDelta optimizer**	1000	64	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 256 (ReLU) and 5 (Linear)
19 (GPU)	0.001 **(w exp decay 0.8 on 200000 steps)**	1000	64	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: 256 (ReLU) and 5 (Linear)
20 (GPU)	0.001 (w exp decay 0.8 on 200000 steps)	1000	64	150000	8	100000	0.95	1e-6 Weight Regularization	2 (ReLU and w max pooling))	2: **400** (ReLU) and 5 (Linear)
21/22(GPU)	0.001 (w exp decay 0.8 on 200000 steps)	1000	64	150000	8	100000	0.95	**Dropout 0.5**	2 (ReLU and w max pooling))	2: 400 (ReLU) and 5 (Linear)
23 (GPU)	0.001 (w exp decay **0.7** on 200000 steps)	1000	64	150000	8	100000	0.95	Dropout 0.5	2 (ReLU and w max pooling))	2: 400 (ReLU) and 5 (Linear)
24 (GPU)	0.001 (w exp decay 0.7 on 200000 steps)	1000	64	150000	8	100000	0.95	**1e-6 Weight Regularization**	2 (ReLU and w max pooling))	2: 400 (ReLU) and 5 (Linear)

TNUF - Target Network Update Frequency; BS - BatchSize; MNR - Maximum Negative Rewards;

References

[1] Błażej Osiński. What is reinforcement learning? the complete guide. https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/, Jul 2019.

[2] Jason Brownlee. What is deep learning? https://machinelearningmastery.com/what-is-deep-learning/, Oct 2019.

[3] Coursera online course: Convolutional neural networks. https://www.coursera.org/learn/convolutional-neural-networks?specialization=deep-learning.

[4] Matthew Zeiler and Rob Fergus. Visualizing and understanding convolutional neural networks. volume 8689, 11 2013.

[5] Assaad MOAWAD. Dense layers explained in a simple way. https://medium.com/datathings/dense-layers-explained-in-a-simple-way-62fe1db0ed75, Oct 2019.

[6] Jason Brownlee. A gentle introduction to the rectified linear unit (relu). https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/, Aug 2019.

[7] Backpropagation calculus — deep learning, chapter 4. https://www.youtube.com/watch?v=tIeHLnjs5U8.

[8] Matthew Stewart. Neural network optimization. https://towardsdatascience.com/neural-network-optimization-7ca72d4db3e0, Jun 2019.

[9] Coursera online course: Improving deep neural networks: Hyperparameter tuning, regularization and optimization. https://www.coursera.org/learn/deep-neural-network?specialization=deep-learning.

[10] Tensorflow documentation: Exponential decay. https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/exponential_decay, Jun 2019.

[11] Pekaalto. Deep q learning. https://github.com/pekaalto/DQN, 2017.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[13] OpenAI. A toolkit for developing and comparing reinforcement learning algorithms. https://gym.openai.com/envs/CarRacing-v0/.

[14] Amd-Rips. Amd-rips/rl-2018 - an openai's carracing-v0 solution. https://github.com/AMD-RIPS/RL-2018.

[15] OpenAI. Openai gym leaderboard. https://github.com/openai/gym/wiki/Leaderboard, 2019.

[16] Mohit Sewak. *Deep Q Network (DQN), Double DQN, and Dueling DQN*, pages 95–108. Springer Singapore, Singapore, 2019.