**University of Tripoli**
**Faculty of Engineering**
**Department of Electrical and Electronic Engineering**

Fall-2024
EE569 - Deep Learning

**Assignment #2 - RL**

الاسم: محمد محمد محمد النعاس
رقم القيد:2200209040

# 1. Abstract:

Reinforcement  Learning is a type of machine learning where a computer controls an agent, learns how to make the best decision by taking actions in an environment and trying to achieve the best reward for each action. In this assignment, two agents (cars) were trained to race effectively in the MultiCarRacing-v0 environment using Deep Q-Learning (DQN). It was found that, for complex environments such as this one, the effects of target network update frequency is great for achieving better performance, and that lower loss does not necessarily correlate to better performance.

# 2. Introduction:

Reinforcement learning aims to learn to take the best decision to achieve the best results by way of improving on cumulative reward. The MultiCarRacing-v0 environment is an OpenAI Gym environment designed for multi-agent car racing. It is a multiplayer variant of the original CarRacing-v0 environment. The key concepts of RL learning, as present in this environment, are as follows:

- **Agents**: Make decisions in the environment, the goal RL is to improve tiger decision-making. In this case, they are the cars to be driven through the track.
- **Environment**: Everything the agent interacts with, providing states and rewards based on the agent's actions.
- **State**: A representation of the current situation the agent is in, here it is a series of images of the cars racing through the track rendered by the environment.
- **Action**: Moves or decisions the agent can take. Include acceleration, steering and breaking.
- **Reward**: Feedback indicating the quality of the agent's actions. Meaning, how far the cars progressed through the track, who got where first. And whether they drove out of bounds or not.
- **Policy**: The strategy the agent uses to decide actions based on states. In this assignment, an e-greedy policy is used.

The goal of this assignment was to use RL to allow two cars to drive through the track, achieving a higher reward per step than that of random decision making.

# 3. Key Features:

## a. States:

In this assignment, the state consists of the last four frames rendered by the environment for each agent; these frames are then converted to grayscale, normalised and stacked channel-wise.

## b. Actions:

The actions of each car have been discretized into a set of five actions. These actions are:

1. Steer Left
2. Steer Right
3. Accelerate
4. Brake
5. Do nothing

Using a discrete set of actions allows for easier implementations of the RL model.

## c. Experience Replay:

For effective learning, a series of transitions (experiences) need to be recorded. Starting with the 2nd state, each state, the decision taken from analysing that state, and the resulting state for taking that actions are recorded in a replay buffer to be used for learning the ideal decisions by the agents.

## d. Policy:

The RL model used the epsilon greedy policy for decision making, this policy allows for higher exploration (random action taking) at the begging of training, that slowly reduced in favour of exploitation (decisions using the model's output) the decay rate of the epsilon term is what determines for how long the exploration period lasts.

## e. Reward:

The code in this assignment modifies the reward function used by the base environment, adding more rewards for speed and forward progress, while increasing the penalty for going outside of the tarmac.

## f. Early Stopping:

In order to not spend time computing when there's no learning to be done, early stopping was implemented to stop the training loop when no improvement in the loss occurs after a certain amount of time.

## g. Premature Episode Termination:

Ensures the quality of training experiences by terminating episodes that are likely to be unproductive.

## h. Metrics Recording:

Alongside frequent print statements, this code leverages TensorBoard for recording and visualising the loss and average reward per step for both agents.

## i. Video Recording:

At regular intervals, an episode is recorded which allows for runtime evaluation of the model.

## j. The DQN model:

This assignment uses a Deep Q-Learning Network model for reinforcement learning, these models leverage the power of deep neural networks for the purposes of estimating the ideal decision (Q-values) desired in RL.
The model uses both convolutional and fully connected layers in order to extract relevant information (features) from the state images and fully connected layers to produce the desired decisions.

## k. Weight Initialization:

A method was introduced to ensure proper weight initialization for all of the different layers used in this model, with each layer requiring its unique initialization method to ensure a chance to converge to the right solution.

## l. Sampling and Mini-Batching:

Experiences are sampled from the replay buffer, breaking the temporal correlation between consecutive experiences. And allowing more diverse learning. Samples are then further broken down into smaller (but configurable) mini-batches. This was done due to the hardware constraints of not having GPU on the training device, limiting the amount of parallel processing the device is capable of executing, and thus the catch size.

## m. The Target Network:

To ensure learning stability a target network used during learning to allow for a more stable reference point for loss calculation. The frequency with which the target network proved to be one of the most important factors in learning stability. With the chosen value being one of the closest estimates for optimal performance.

## n. Smooth L1 Loss Function:

The Huber Loss function was chosen for its ability to combine the best of both MSE and MAE loss functions and robustness against outliers.

## o. ADAM Optimizer:

The ADAM optimizer was used in this assignment in compliance with specification in the assignment paper.

## p. Weight Saving:

As the training process is completed, the model's weight is saved in a file and can be used to evaluate the model later.

## q. Evaluation:

A separate environment instance is created where the saved parameter weights can be loaded and used to evaluate the performance of the model.

# 4. Challenges & Experimentation:

## a. Episode Abortion:

One of the primary experimental additions to this code include the early episode abortion functionality, which at first was not present, this caused the replay buffers to be filled with experiences where no significant progress occurs. The initial idea was to terminate the episode once the cars go off into the grass, but that meant denying them the opportunity to learn how to get back on track, and terminated many episodes without sufficient exploration. Eventually, the current system, which terminates the episode after the episode reward drops below a certain threshold, was implemented. This method strikes a balance between both termination of bad episodes without filling the buffer with just grass, and allowing the model just enough time to learn how to course-correct.

## b. Sample Mini-Batching:

At first, The learning process entailed sampling many smaller samples from the replay buffer over a series of iteration due to the GPUless constraints. But since such a method would require many costly sampling operations where the entire replay buffer would need to be loaded multiple times. The idea was modified until the current method, which uses one big sample and iterates over it, was implemented. With hopes that it allows less learning time by reducing downtime from experience replay sampling.
It should be noted that such a method of learning causes many learning iterations per episode which means that a lower learning rate should be employed to avoid divergence.

## c. Reward Function:

At first, the custom rewards function was set to include much heavier penalties for going off track and driving slowly, but such penalties were found to cause suboptimal behaviour for the model. Which lead to both rewards and penalties being reduced to much smaller values whilst allowing for driving at lower speeds without being penalised.

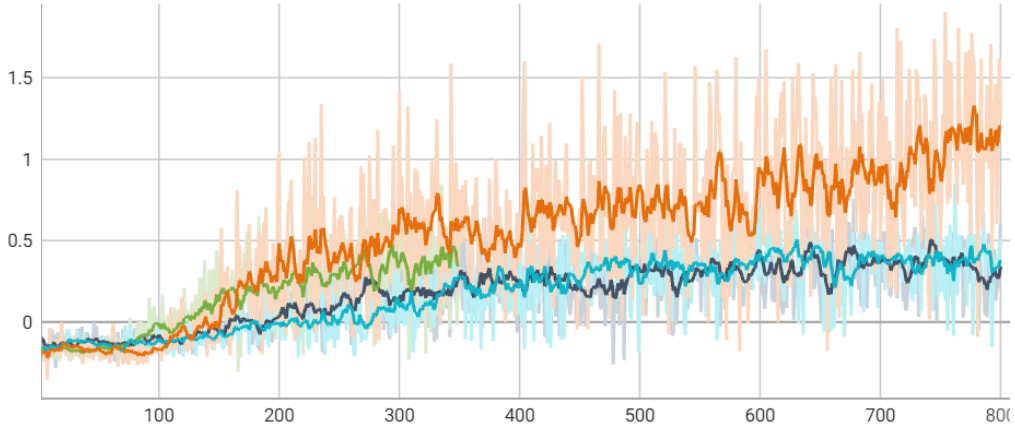
## d. Monitoring:

The nature of machine learning requires that the training process be left running for long periods of time until the desired result is achieved. This however can alose cause the user to leave a diverging model running, they can’t observe if said model is reaching the desired

result. At first, the usage of extensive print statements was implemented in order to monitor the model's performance throughout each episode and during the episode itself. Such a method flooded the terminal and made it hard to actually monitor the model's progress. So it was ultimately replaced with much less print statements and the usage of PyTorch's TensorBoard, this tool allows the user to view the model's progression through a simple graph and to determine whether or not the model is producing progress for the user to decide if they want to terminate the learning process before the intervention of early stopping.
Additionally, the training loop was modified to frequently record the environment every a set number of episodes. Allowing for visual proof of the progression of both models.

# 5. Results:

The graphs for the models can be accessed by running TensorFlow. Below is an overview of the average reward per agent per step for each episode with different, tweaked runs of the models:




It should be noted that the first two runs (green and yellow) also use a tweaked reward method that yields higher rewards for each action. Causing a higher reward even though that, when evaluated, they'd drive worse than the first two.

For test a of the eye evaluation, the eval.py file can be run provided the correct paths of the relevant weights. If the user wishes to, they can increase the number of agents in order to compare the different models together, as well as race them against a human driver.