Reinforcement Learning- Final Course Project

Sharon Mordechai, Amit Huli

Submitted as final project report for the RL course, IDC, 2022

1 Introduction

In this project, we covered the theoretical and practical parts according to the discussed main topics in the Reinforcement Learning course. We provided several variations agents, such as DQN, DDQN, and A3C including ICM (Curiosity-Driven Learning) models, for solving the Highway environment, which is an environment collection for autonomous driving and tactical decision-making tasks.

Our goal is to aid the user car to overtake the bot cars on its own in a roadway environment. The agent controls a vehicle with a finite set of maneuvers implemented by low-level controllers, Actions = no-op, right-lane, left-lane, faster, slower. It is driving on a lane road populated with other traffic participants. The vehicles in front of the agent drive slowly, and there are incoming vehicles in the opposite lane. Their behaviors are randomized, which introduces some uncertainty concerning their possible future trajectories. The task consists of driving as fast as possible, which is modeled by a reward proportional to the velocity. It motivates the agent to try and overtake its preceding vehicles by driving fast in the opposite lane. This optimal but overly aggressive behavior can be tempered through a cost function that embodies a safety objective.

We trained our agent, the user car, with the asset of deep reinforcement learning. The reward function will penalize the agent every time it slows down, each time it crashes into the bot car, and if there are any bot cars in front of it. In the following environments, we will use the raw pixels (4x128x128) as our state space, thus, it will allow us to train CNN Neural Networks.

It comprised of the "Highway-Env", "Merge-Env", and "Roundabout-Env" environments. In the "Highway-Env" environment (which is constructed with three or six lanes), the ego-vehicle is driving on a multi-lane highway populated with other vehicles. In the "Merge-Env" environment, the ego-vehicle starts on the main highway but soon approaches a road junction with incoming vehicles on the access ramp. In the "Roundabout-Env" environment, the ego-vehicle is approaching a roundabout with flowing traffic. It will follow its planned route automatically but has to handle lane changes and longitudinal control to pass the roundabout as fast as possible while avoiding collisions.

1.1 Related Works

Multi-Agent Reinforcement Learning with Application on Traffic Flow Control. Dueling Deep Q Network for Highway Decision Making in Autonomous Vehicles: A Case Study.

2 Solution

2.1 General approach

As mentioned, we would need to train our agent to control a vehicle to drive on a lane road populated with other traffic bot cars while preventing collisions and gaining optimal rewards.

In order to achieve this goal, we used neural networks, which are effective in learning patterns, to replace the Q-table approach. Each input is comprised of a sequence of the last four gray scale images (4x128x128) to obtain the agent's movements and observations respectively. Additionally, to learn about the images' features, we used CNN architecture, which is very powerful in understanding those data types.

The benefits of this approach are that we do not need to memorize separate policies for each state in the environment. This means we can now perform Reinforcement Learning in larger environments without memory constraints.

2.2 RL algorithms

We used three different algorithms to achieve this goal: DQN, DDQN, and A3C using ICM (for Curiosity-Driven Learning) models.

Deep Q-Network (DQN): DQN approximates a state-value function in a Q-Learning framework with a neural network. We take in several frames of the game as an input and output state values for each action as an output. It is usually used in conjunction with the Experience Replay buffer for storing the episode steps in memory for off-policy learning, where samples are fetched from the replay memory at random. In addition, the Q-Network is optimized towards an on-hold target network that is periodically updated with the latest weights every k steps, which makes training more stable by preventing short-term changes from a moving target. (Published paper: Playing Atari with Deep Reinforcement Learning paper)

Double Deep Q-Network (DDQN): The idea of Double Q-learning is to reduce over estimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. Therefore, we evaluate the greedy policy according to the online network but use the target network to estimate its value. (Published paper: Deep Reinforcement Learning with Double Q-learning paper).

Asynchronous Advantage Actor-Critic (A3C) Algorithm: A multi-threaded asynchronous method using the advantage actor-critic algorithm.

A3C is a policy gradient algorithm in reinforcement learning that maintains a policy and an estimate of the value function. It operates in the forward view and uses a mix of n-step returns to update both the policy and the value-function.

This asynchronous method aims to train deep neural network policies reliably (reduce bias) and without large resource requirements. The critics in A3C learn the value function while multiple actors are trained in parallel and synced with global parameters every so often.

Each thread interacts with its replication of the environment and at each step computes a gradient of the Q-learning loss. In addition, we accumulate the gradients over multiple time-steps before they are applied, which is similar to using mini-batches.

This reduces the chances of multiple actor learners overwriting each other's updates. Accumulating updates over several steps also provides some ability to trade off computational efficiency for data efficiency.

(Published papers: Actor-Critic Reinforcement Learning for Control with Stability Guarantee paper, Asynchronous Methods for Deep Reinforcement Learning paper).

The Intrinsic Curiosity Module (ICM) - Curiosity-Driven Learning: Agents use rewards to alter their behavior and would be stuck without them. To overcome this problem we use an intrinsic reward mechanism. The idea of intrinsically-driven learning, mostly known as curiosity, enables the agent to self-learn and be rewarded for discovering new states.

ICM is a general mechanism for learning feature representations such that the prediction error in the learned feature space provides a well intrinsic reward signal.

It presents the integration of an intrinsic reward function that guides the agent into choosing a better action. This intrinsic reward function reflects the prediction error of the next state, given the current state. Thus, maximizing this reward function means that it is harder to predict the next state, i.e., exploring unknown trajectories.

The Intrinsic Curiosity Module is significant since it provides the ability of RL to scale with big and complex environments without the dependency on human-made reward functions. Using curiosity, the agent will favor actions with high prediction error (i.e, less-visited transition, or more complex ones) and so it will explore the environment better. (Published paper: Curiosity-driven Exploration by Self-supervised Prediction paper).

In this project, we applied these three algorithms in the first experiment (*highway-env easy*). We implemented DQN, DDQN, and A3C (including ICM, to encourage exploration of our agent). Here, *highway-env easy*, the model uses a stochastic mechanism where the output action will run correctly in 85 percent of the cases, and in the remaining 15 percent, it will perform a different action.

For the second experiment (*highway-env medium*), we used DQN and DDQN algorithms, and we excluded A3C due to resource limitations.

Lastly, for the third experiment (*super-highway agent*), we compared both DQN and DDQN algorithms for our super agent. The super-agent requires to be trained in the different environments: highway-env, merge-env, and roundabout-env.

In all of the steps, the agent applies the Exploration vs. Exploitation approach. For every render of the environment, our agent decides to randomly explore the action space or choose one from the predicted network actions.

2.3 Design

The project was built using Keras API with the following code structure:

Project algorithms, which include several utilise functions (for installation, display, and configuration settings), and our Reinforcement Learning algorithms: DQN, DDQN, and A3C with ICM network.

Environments, which include the steps of the experiments according to the project's instructions (highway-env easy, highway-env medium, and highway-super agent). For each experiment, we plotted the results using the rewards and loss history for each performed episode.

For the final step (highway-super agent), we also implemented a hyper-parameters tuning section for choosing the best hyper-parameter for our agent. We chose to use the *batch-size*, *discount-factor*, and *learning-rate* parameters for each algorithm, and we reduced the number of episodes due to resource constraints.

The total time for training our agents in these various environments, took several days. We have mostly struggled with the implementation of the A3C and ICM models (which added additional complexity in terms of building neural networks), selecting the proper hyper-parameters for the DQN and DQDN algorithms, and improving our super agent in three different environments.

For the **DQN** and **DDQN**, we chose to build a CNN for learning the task, which is better at processing and handling images. Four sequential gray scale images of the environment are taken and given as the input. This contains our agent moving through the environment. Our network architecture is comprised of layer blocks of multiple Convolutional layers and finalizing with Fully connected layers.

We used Huber as our loss function, and Root Mean Squared Propagation (RMSprop) as our optimizer, which is initiated with the fixed hyper-parameters: learning-rate=0.00025, and rho=0.95. In these experiments, we used mini-batches of size 32. The policy during the training was epsilon-greedy with epsilon decreasing linearly from 1 to 0.1 over the first thousand frames and fixed at 0.1 thereafter. Additionally, We used a replay memory of one 15 thousand most recent frames and a discount factor of 0.99.

The Huber loss function can be used to balance the Mean Absolute Error (MAE) for large values, and the Mean Squared Error (MSE) for low values. Therefore, we have found it very useful in our case, when we aim to minimize our errors using 2 neural networks when using DQN and DDQN.

Moreover, we have trained our model every four frames and updated our target model every 10 thousand frames. Using 10 thousand frames, for each update, improved our loss reduction during the training.

For the **A3C** and **ICM** models, we also chose to build a CNN for learning the task, using four sequential gray scale images of the environment as the input. Our network architecture is comprised of layer blocks of multiple Convolutional layers with Fully connected layers (based on the Actor-Critic Reinforcement Learning for Control with Stability Guarantee and the Curiosity-driven Exploration by Self-supervised Prediction papers).

The ICM model includes three neural networks: the **inverse-model** provides a sense of self-supervision by predicting the agent's action given the current and next states. The **forward-model** predicts the next state given the current state and the action (ICM model brings both these models together), and the ICM function uses the build-feature-map function that supports the vectorization of the states. It ensures the model only uses elements that can be controlled by or affect the agent.

We used Mean Squared Error (MSE) and Categorical Crossentropy as our loss functions (according to the value and policy error functions) and Root Mean Squared Propagation (RMSprop) as our optimizer, which is initiated with the fixed hyper-parameters: learning-rate=0.00001, rho=0.99, and epsilon=0.1. For the ICM parameters, we used lambda=1.0 and beta=0.01. Lambda controls the ratio of external loss against the intrinsic reward, and beta weights the inverse loss against the forward reward. Additionally, for the experiments, we use a batch size=16 and four asynchronous processes.

This algorithm is used as an integration of the intrinsic reward function (curiosity encouragement) as required in the first experiment. Later in the notebook, we decided to omit it and focus on the other algorithms for simplicity manner.

3 Experimental results

We have performed experiments on three different levels - Highway-env Easy (3 lanes), Highway-env Medium (6 lanes), and Super Highway Agent (highway, merge, and roundabout). For the DQN and DDQN algorithms, we used the same network architecture and hyperparameter settings across all three environments, demonstrating that our method is robust enough to work in diverse environments. For the A3C & ICM algorithm we used different network architecture and hyperparameter settings to encourage exploration of our agent.

We have evaluated our agents on the real and unmodified games, and we did not add any changes to the structure of the environment during the training.

Highway-env Easy: In this experiment, we used the DQN, DDQN, and A3C & ICM algorithms.

For DQN and DDQN, we trained our agents for 10,000 episodes (in the google-colab notebook you can find the last 1000 running episodes due to RAM size limitations).

Our evaluation metric is the total reward the agent collects in an episode or game averaged over several games, and we periodically computed it during the training. In addition, we added the average loss in an episode or game to observe our agent learning process.

According to the results below, the average total reward metric tends to be slightly noisy since small changes to the weights of a policy can lead to large changes in the distribution of states the policy visits. However, we can observe the increase in the rewards, meaning our agent has been improved during the training.

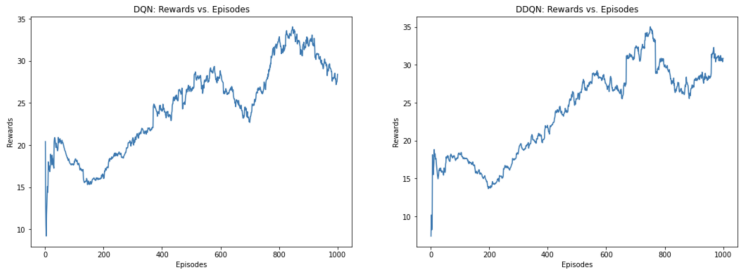

Figure 1: DQN and DDQN rewards vs. episodes plot (EX1 - Easy).

While comparing both algorithms based on their rewards, we can consider that the results are quite similar. Nevertheless, in the evaluation step (in the notebook), we found that the DQN performed better according to the videos.

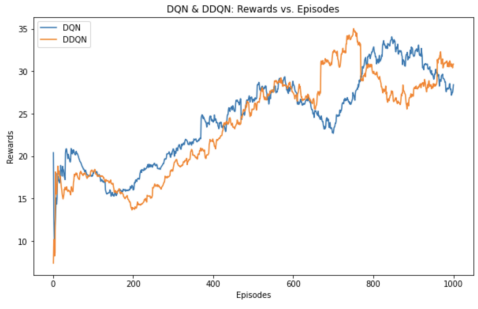

Figure 2: DQN and DDQN rewards vs. episodes merged plot (EX1 - Easy).

Below, we can observe the loss per episode plot, which tends to converge quite well during the training process.

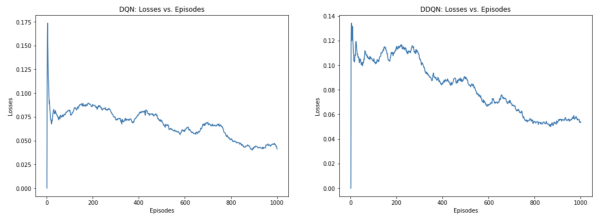

Figure 3: DQN and DDQN loss vs. episodes plot (EX1 - Easy).

For A3C & ICM, we trained our agent for 10,000 episodes with sixteen different processes (in the google-colab notebook you can find the last 1000 running episodes due to RAM size limitations). Our evaluation metric was also the total reward the agent collects in an episode averaged over several games, and we periodically computed it during the training for each process.

According to the results below, we can observe the increase in the rewards, meaning our agent has improved during the training. However, we have found that this algorithm has poor results compared to the other algorithms.

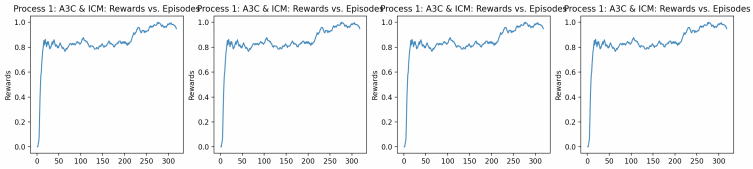

Figure 4: A3C & ICM rewards vs. episodes plot (four processes example).

Highway-env Medium: In this experiment, we used the DQN, DDQN algorithms.

We trained our agents for 10,000 episodes (in the google-colab notebook you can find the last 1000 running episodes due to RAM size limitations). Our evaluation metric is the same as in the easy mode, including the average loss metric.

According to the results below, the average total reward metric tends to be much noisy. However, we can observe the increase in the rewards, meaning our agent has been improved during the training.

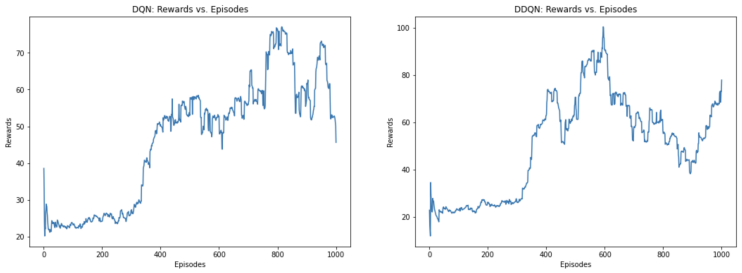

Figure 5: DQN and DDQN rewards vs. episodes plot (EX2 - Medium).

While comparing both algorithms based on their rewards, we have found that the DDQN algorithm had better results than DQN in both metrics (rewards and loss) and based on the evaluation step on the video.

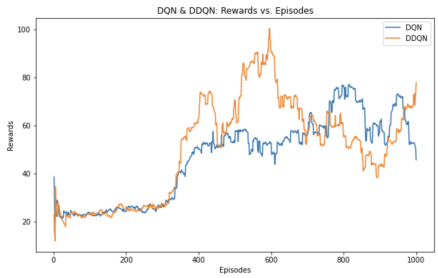

Figure 6: DQN and DDQN rewards vs. episodes merged plot (EX2 - Medium).

Below, we can observe the loss per episode plot, which tends to converge quite well, the DDQN in particular, during the training process.

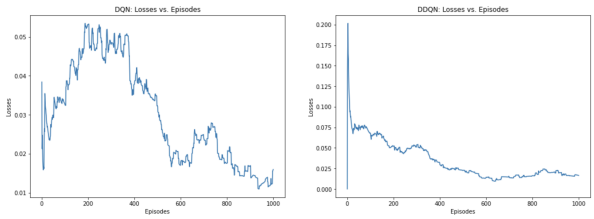

Figure 7: DQN and DDQN loss vs. episodes plot (EX2 - Medium).

Super Highway Agent: In this experiment, we used the DQN and DDQN algorithms in three different environments. We trained our agents for 10,000

episodes in the highway-env, and 2000 episodes in each of the other environments (in the google-colab notebook you can find the last 1000/200 running episodes due to RAM size limitations). Our evaluation metrics are the average total rewards and losses.

According to the results below, in the highway environment the average total rewards and losses metrics tend to converge quite well. In the merge environment the average total rewards and losses were highly noisy since we used 200 episodes. In the roundabout environment the average total rewards tends to be also noisy also since we used 200 episodes, whereas the losses were converged.

Nevertheless, in all of the environments, our agents have been improved during the training.

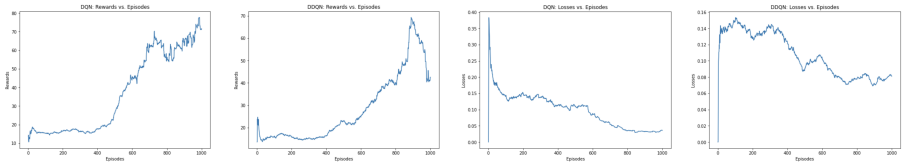

Figure 8: [Highway-env] DQN and DDQN rewards/losses vs episodes plot.

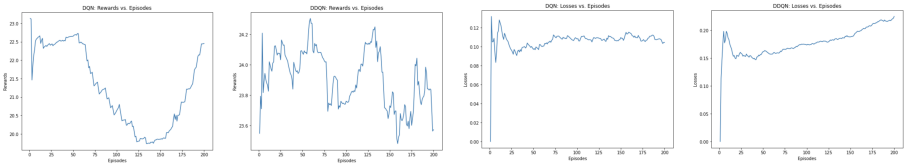

Figure 9: [Merge-env] DQN and DDQN rewards/losses vs episodes plot.

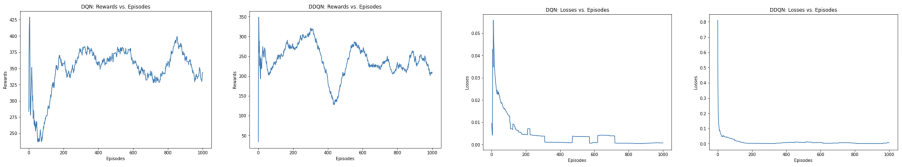

Figure 10: [Roundabout-env] DQN and DDQN rewards/losses vs episodes plot.

	highway-env easy	highway-env medium	highway-env super	merge-env super	roundabout-env super
Random	18.2	33.9	11.8	22.4	15.6
DQN	496.7	499.7	374.2	27.5	479.2
DDQN	491.1	488.6	242.6	23.3	401.6
A3C & ICM	354.4				

Figure 11: The upper table compares the average total reward for various learning methods.

While comparing the algorithms based on the table results [11], we can conclude that the DQN performs better than the others, and when evaluating the agents using the game videos, the DQN algorithm had better movements than the DDQN.

Moreover, we have seen that training our super agent in three different environments can affect the efficiency in each environment. For instance, completing the roundabout-env causes misleading movements in the highway-env and vice-versa.

4 Discussion

This project introduced several deep learning models for reinforcement learning, such as DQN, DDQN, and A3C & ICM models, and demonstrated its ability to learn control policies for different Highway environments games. Our focus was on applying the DQN and DDQN algorithms.

Based on the final results, the DQN algorithm, with the less complex architecture, had the best results compared to all the other algorithms in all the environments.

To conclude, our agents succeeded in learning and improving on these environments using a simple network architecture and using only raw pixels as input. Although the agents had some flawed movements in some games, the overall actions are good, and we definitely can use them to master these games.

5 Code

The google colab notebook can be found here.