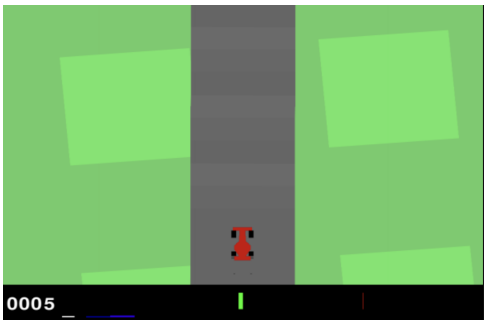
# Playing Car Racing with Deep Reinforcement Learning

Alexandrescu Nicolae, Adrian Onofrei

January 2024

## 1 Abstract

In this article we present a Python project designed to provide students with a hands-on introduction to reinforcement learning through the development of an autonomous car racing simulation using the environment CarRacing-v2[1] environment from Gymnasium. We inspired from paper Playing Atari with Deep Reinforcement Learning[2] and implemented the DQN model for this project and in the end of this article we present our results and conclusion.




## 2 Introduction

The given problem encapsulates the essence of decision-making under uncertainty, as the car must dynamically adapt to the conditions of the race track to achieve optimal performance. The observation space will be a top-down 96x96 RGB image of the car and the race track, meanwhile the reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track. For example, if you have finished in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points. The episode finishes when all the tiles are visited. The car can also go outside the playfield - that is, far off the track, in which case it will receive -100 reward and die.

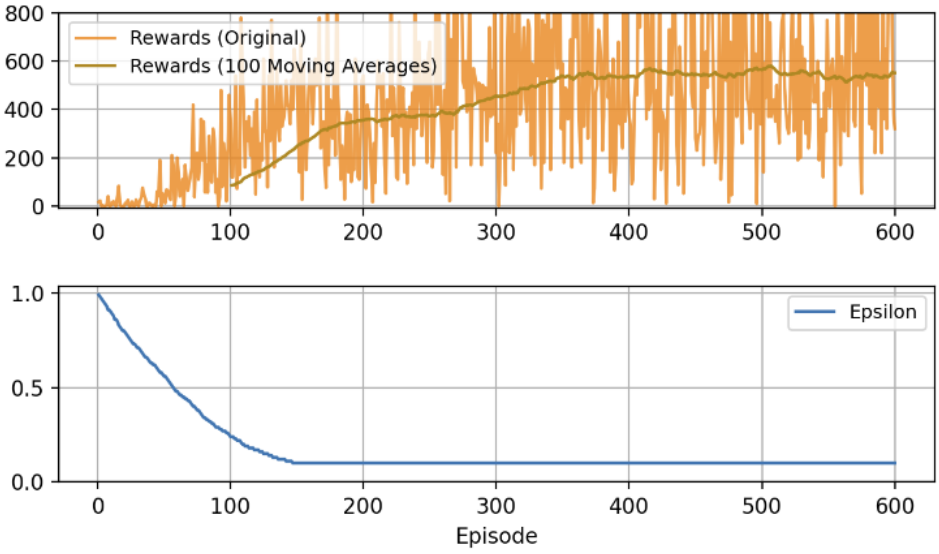
**Action Space** There are 5 actions :

- **0:** do nothing
- **1:** steer left
- **2:** steer right
- **3:** gas
- **4:** brake

The most optimal solution achieved a score of 850 with 2 million steps. Because of our hardware, we couldn't train the model with more steps. With this configuration, a single training session took about 16 hours to complete with a RTX 3080 Ti laptop GPU and a 12th Gen Intel(R) Core(TM) i9-12950HX CPU.

# 3 State of The Art

As mentioned in abstract, we followed the approach from Playing Atari with Deep Reinforcement Learning[2] paper published in 2013, which presents a convolutional neural network model, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. The authors applied their method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm and they fond that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.
For comparation with other results we manged to get better results than the ones we fond in OpenAI-GYM-CarRacing-DQN[3] repository from Github with the following image that shows the reward for each episode (an episode has 1000 steps and we used the step number in our graphs) and epsilon decay, the factor that determinate the chance of exploration:


Rewards (Original)
Rewards (100 Moving Averages)
800
600
400
200
0
0
100
200
300
400
500
600
Epsilon
1.0
0.5
0.0
0
100
200
300
400
500
600
Episode

# 4 Our approach

We used the neural network model structure from the Playing Atari with Deep Reinforcement Learning[2] paper:

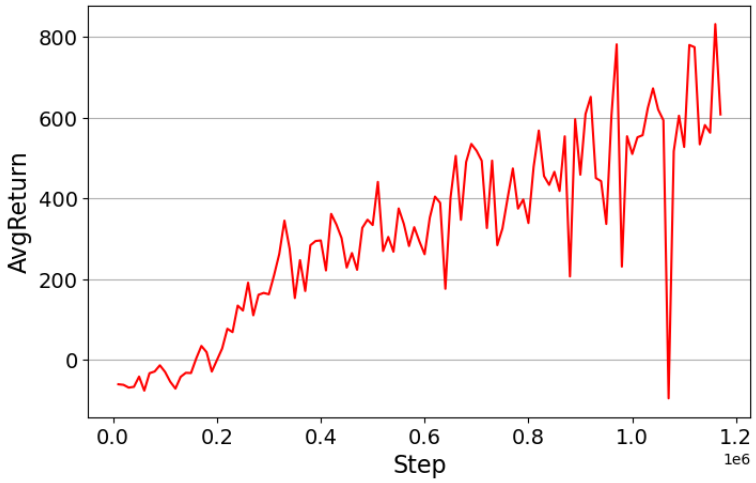
- The input to the neural network consists is an 84 × 84 × 4 image ( 4 consecutive states of the environment without the pixels from the bottom margin).
- The first hidden layer convolves 16 8 × 8 filters with stride 4 on the input image.
- The second hidden layer convolves 32 4 × 4 filters with stride 2.
- The final hidden layer is fully-connected and consists of 256 rectified units.
- The output layer is a fully connected linear layer with a single output for each valid action, five in our case.

```
class CNNActionValue(nn.Module):
    def __init__(self, state_dim, action_dim, activation=F.relu):
        super(CNNActionValue, self).__init__()
        self.conv1 = nn.Conv2d(state_dim, 16, kernel_size=8, stride=4)  # [N, 4, 84, 84] -> [N, 16, 20, 20]
        self.conv2 = nn.Conv2d(16, 32, kernel_size=4, stride=2)  # [N, 16, 20, 20] -> [N, 32, 9, 9]
        self.in_features = 32 * 9 * 9
        self.fc1 = nn.Linear(self.in_features, 256)
        self.fc2 = nn.Linear(256, action_dim)
        self.activation = activation

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = x.view((-1, self.in_features))
        x = self.fc1(x)
        x = self.activation(x)
        x = self.fc2(x)
        return x
```

We observed that the game screen gradually zooms in for the first 50 steps. You can move the car during this zoom-in phase, but this zoom-in phase is a very small part of the overall game, which may hinder our agent from learning to control the car. Thus, we are not useing the first 50 steps of the game. And because we are given only one current game frame for each step and the agent can not guess if the car is moving forward or backward from only one frame, which means we cannot predict the next frame so we need to stack the previous four frames. As other image preprocess we cut the bottom margin with score and other metrics from images and applied a grayscale filter to the images. The replay buffer had maximum size of 100000 because it requires a big amount of RAM memory. For epsilon greedy policy, we used the initial epsilon of 0.99 and linearly decreased it to 0.01 for the the first million steps, and fixed it at 0.01 after.

## 4.1 Benchmark Performance

After some trial and error, we discovered that the gray filter leads to the best results. Below you can see what are the images that were sent to the agent and the results:



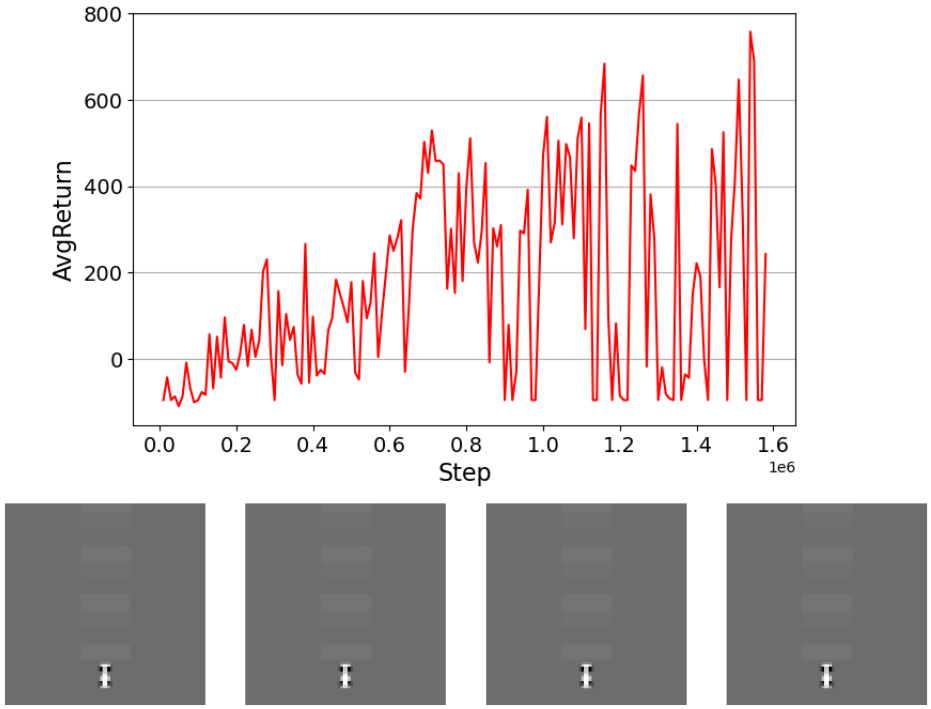

```
]: %%time

eval_env = gym.make('CarRacing-v2', continuous=False, render_mode='rgb_array')
eval_env = ImageEnv(eval_env)

total_reward = 0
number_of_episodes = 30

for x in range(number_of_episodes):
    frames = []
    episode_reward = 0
    (s, _), done = eval_env.reset(), False
    while not done:
        frames.append(eval_env.render())
        x = torch.from_numpy(s).float().unsqueeze(0).to(device)
        q = model(x)
        a = torch.argmax(q).item()
        observation, reward, terminated, truncated, info = eval_env.step(a)
        s = observation
        episode_reward += reward
        done = terminated or truncated
    total_reward += episode_reward

env.close()
print(f"Average reward dqn_grayscale_895.2486.pt model after {number_of_episodes} episodes: {total_reward / number_of_episodes}")
```

```
Average reward dqn_grayscale_895.2486.pt model after 30 episodes: 862.1551194890594
CPU times: total: 15.9 s
Wall time: 4min 41s
```

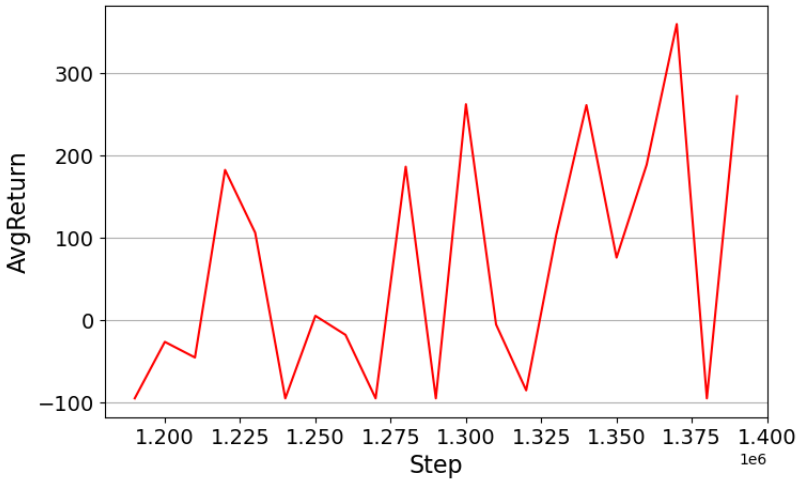
## 4.2 Ablation Study

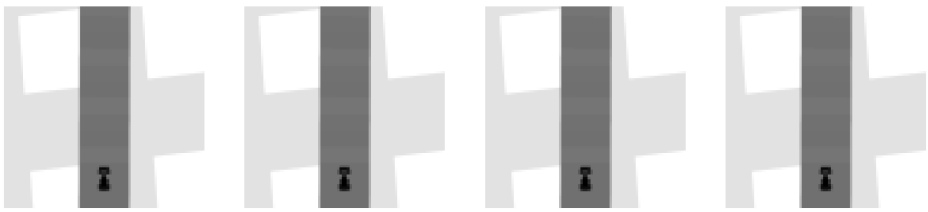
We tried to train the model only on a specific color channel:

- **Red** channel average result plot

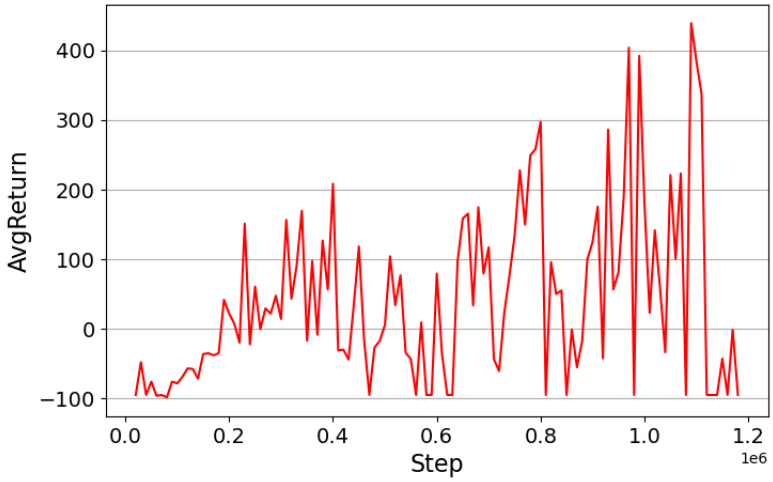



- **Green** channel average result plot

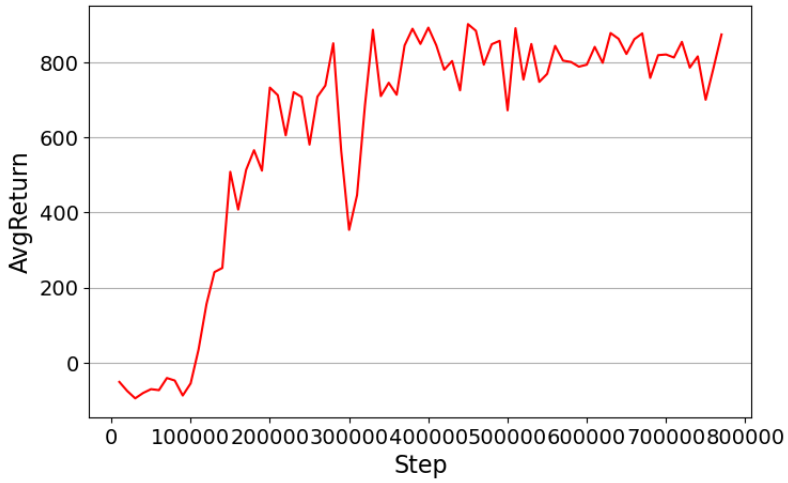




- **Blue** channel average result plot




- Grayscale filter with increasing gamma




starting gamma = 0.01, starting epsilon = 0.99
gamma = min(max_gamma, 1 - self.epsilon)
max gamma = 0.99, min epsilon = 0.01

We reach max gamma and min epsilon after 100k step.

## 5 Conclusion

This paper has presented a comprehensive solution to the problem of finding a reinforcement learning model that obtains best scores in CarRacing-v2[1] environment from Gymnasium using Deep Q-Network.

## References

[1] Gymnasium environment: https://gymnasium.farama.org/environments/box2d/car$_r acing/$

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[3] OpenAI-GYM-CarRacing-DQN https://github.com/andywu0913/OpenAI-GYM-CarRacing-DQN