# Reinforcement Learning using the CarRacing-v0 environment from OpenAI Gym

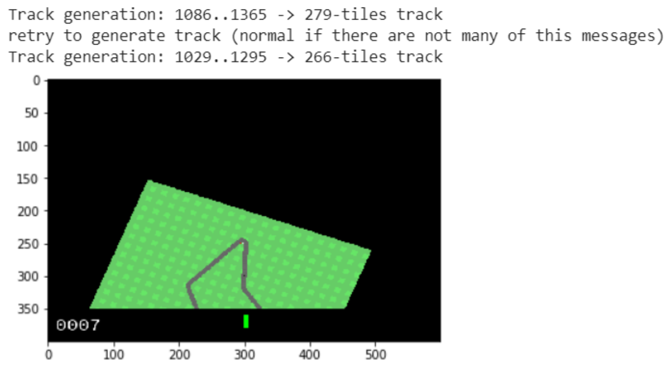
Nikhil Ramesh[1] and Simmi Mourya[1]

University of Pennsylvania

**Abstract.** In this project we implement and evaluate various reinforcement learning methods to train the agent for OpenAI- Car Racing-v0 game environment. Our current method explores Fully connected Deep Q-network and achieves an average reward of 210.92 for 10 evaluation steps. We train our best performing model which contains 70,475 paramaters for 570 episodes. We also explored methods like vanilla DQNs, DQNs with dropout and Proximal Policy Optimization. We added a small negative reward as a metric to penalize the number of green pixels (aka green reward) in the frame. It helped us converge faster than the case when we were not using the green reward. Link for the presentation video: https://youtu.be/iv19T5s-oHc

## 1 Introduction

The Car Racing-vo environment task learns from pixels. Each state contains 9216 = 96x96 pixels. For each frame there is a fixed reward of -0.1 and for every track tile visit there is reward of +1000/N where N is the total number of tiles in the track in that particular frame. For example, if you have finished in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points. [3] An episode finishes when the car visits all the tiles. We choose OpenAI Gym platform since it provides somewhat closer to the real-world RL applications where actions are continuous and state spaces are generally high-dimensional.



```
Track generation: 1086..1365 -> 279-tiles track
retry to generate track (normal if there are not many of this messages)
Track generation: 1029..1295 -> 266-tiles track
```

Fig. 1: Sample frame from the environment

## 2 Literature Review

[2] explores various versions of DQN methods such as Fully connected networks, vanilla CNNs and pre-trained VGG-16 based Deep Q-networks. Findings show that using smaller neural networks for RL based tasks is usually favored since larger networks become a bottleneck during creidt assignment hence making the training slower. Apart from using CNNs [4] explores building generative deep learning models of popular reinforcement learning environments. [6] harnesses the power of Variational Autoencoders to train RL agents. A specific RL algorithm called Deep Deterministic Policy Gradient (DDPG) learns the policy using VAE features as input. There are some advancements reported in double and dueling deep Q-networks for solving similar RL based tasks. [1].

## 3 Experiments and Results

Problem Formulation: The action space consist of 3-tuples: (Steer, Gas, Break).
*Value Ranges:* Steer â [â1, 1], Gas â [0, 1], Break â [0, 1].
**Environment reaction for these ranges:**
Steer: From hard left turn to hard right turn
Gas: None to Full Power (Full forward)
Break: None to Completely stop the motion

### 3.1 Attempted Approaches:

**1. DQN:** We started with a vanilla Deep Q Network to train the agent. **Our Action space**: Consists of four specific actions: left, right, forward, do nothing.
possible_actions = [[1.0, 0.3, 0.0], [-1.0, 0.3, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 0.8]]

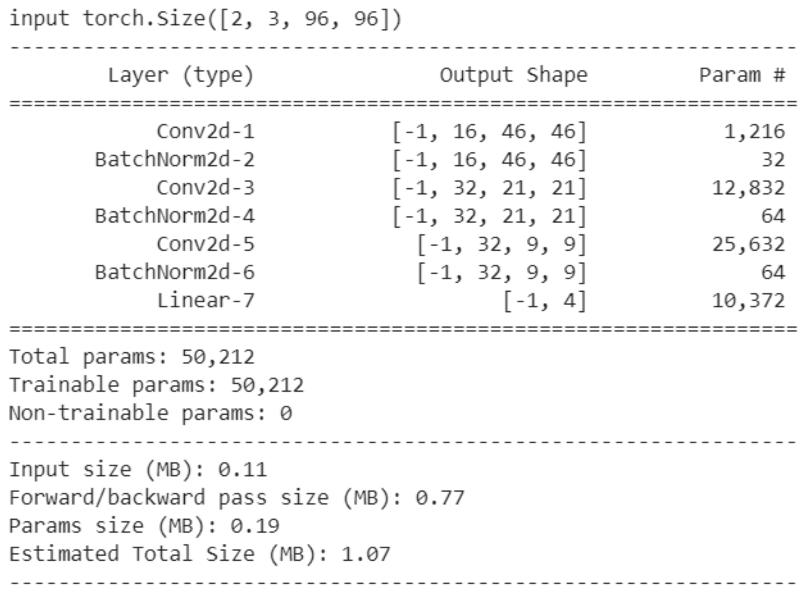
We take inspiration from [5] and design a small neural network which is slightly different than the one mentioned in the paper. The idea was to fit the network on GPU and learn faster credit assignment mechanism. The agent learns to drive the car from raw pixels (rendered screen by Open-AI gym environment). We use a CNN to approximate the optimal Value-Action function as represented below.

$$Q^*(s,a) = \max_\pi \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi\right]$$

**Techniques used: 1. Replay memory:** [5] mentions an excellent technique to address instability which is presented by training a neural network for action-value updates. In our case, Replay memory is a buffer of (States, Rewards, Actions and Next State). We use a moving buffer of size 10000 and do the first network parameter optimization only after we occupy the first 10,000 steps and then use a batch-size of 128 (or 64 different experiment) to randomly sample from the memory for a training step. Having a replay memory helps randomizing over data hence removes correlations in the stream of observations. This also smoothens out the changes in the data distribution.

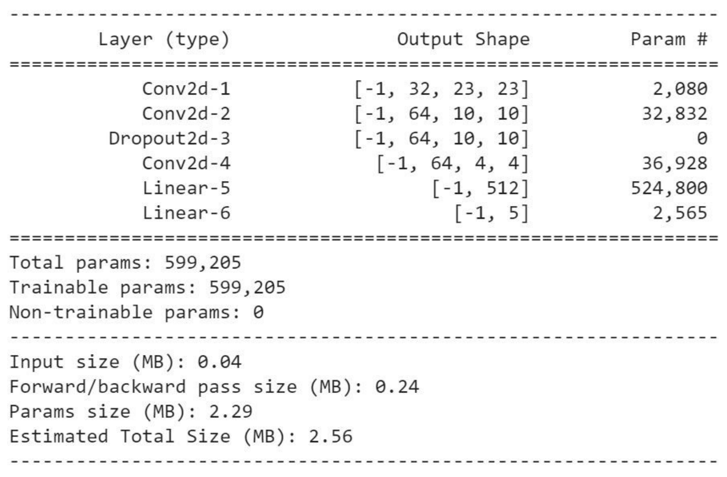
**2. Iterative update:** An iterative update was used to adjust the Q values towards the target values (updated periodically after every 10 episodes). This approach also reduces the correlations with the target.

The network architecure for the DQN is displayed below:



```
input torch.Size([2, 3, 96, 96])
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 46, 46]           1,216
       BatchNorm2d-2           [-1, 16, 46, 46]              32
            Conv2d-3           [-1, 32, 21, 21]          12,832
       BatchNorm2d-4           [-1, 32, 21, 21]              64
            Conv2d-5             [-1, 32, 9, 9]          25,632
       BatchNorm2d-6             [-1, 32, 9, 9]              64
            Linear-7                    [-1, 4]          10,372
================================================================
Total params: 50,212
Trainable params: 50,212
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 0.77
Params size (MB): 0.19
Estimated Total Size (MB): 1.07
----------------------------------------------------------------
```

The network architecture (Slightly updated version as compared to vanilla DQN, more deeper and regularized by Dropout) for DQN is displayed below:

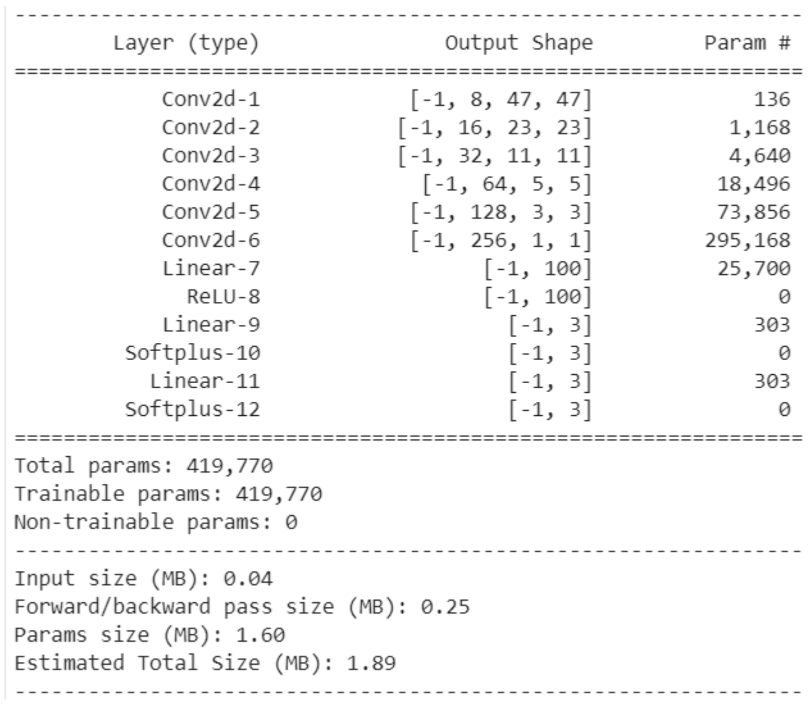


```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 23, 23]           2,080
            Conv2d-2           [-1, 64, 10, 10]          32,832
         Dropout2d-3           [-1, 64, 10, 10]               0
            Conv2d-4             [-1, 64, 4, 4]          36,928
            Linear-5                  [-1, 512]         524,800
            Linear-6                    [-1, 5]           2,565
================================================================
Total params: 599,205
Trainable params: 599,205
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.04
Forward/backward pass size (MB): 0.24
Params size (MB): 2.29
Estimated Total Size (MB): 2.56
----------------------------------------------------------------
```

We trained these DQN variants for 1000 and 800 epochs respectively which took approx 9.5 and 8 hours respectively. We use RMSProp optimizer for training both of the models with a learning rate of 1e-3. We also downsize the screen by a factor of 40 via torchvision resize function. We train an epsilon greedy DQN with starting EPS = 0.9 and decay it by a factor of 200. Although there were many positive scores in the intermediate episodes, none of the models showed a continuous increasing trend of the scores. Some of the sample scores from various iterations are showed in the figures Figure 4 and Figure 5 in Appendix. We observed that even after prolonged training, the average reward over episodes was oscillating between high negative value and 50.

*Possibles Reasons for failure:*
1. Action space: We tried a different combinations of discrete action space.
2. We also noticed that just like GANs there are certain tricks of training RL models. Hyperparameters like replay memory size, Batch size, No of episodes, target network update duration, action space are play key role in success of training RL model. We also observed some minor bugs in our training loop which was the main reason for slow training.

**2. Proximal Policy Optimization or PPO**
Here instead of using a discretized action space, we let the model learn the alpha and beta parameters of beta distribution [7]. We then use the torch.distributions.beta to generate action samples. We implemented a slightly modified version of PPO without the surrogate loss. The motivation was to simply diversify the action space keeping the training loop same.
**Experimental details:**
We tried using both normalized RGB and gray scale images to adjust the complexity of the model. We added a small negative reward as a metric to penalize the number of green pixels (aka green reward) in the frame. It helped us converge faster than the case when we were not using the green reward. We use RMSProp optimizer for training the PPO model with a learning rate of 1e-3. This model was relatively faster than our previously trained DQNs. We did not track results for PPO after certain iterations because Fully Connected DQNs (as explained the section below) worked faster and much better as compared to modified PPO. The Architecture used for this experiment (PPO) is displayed below:

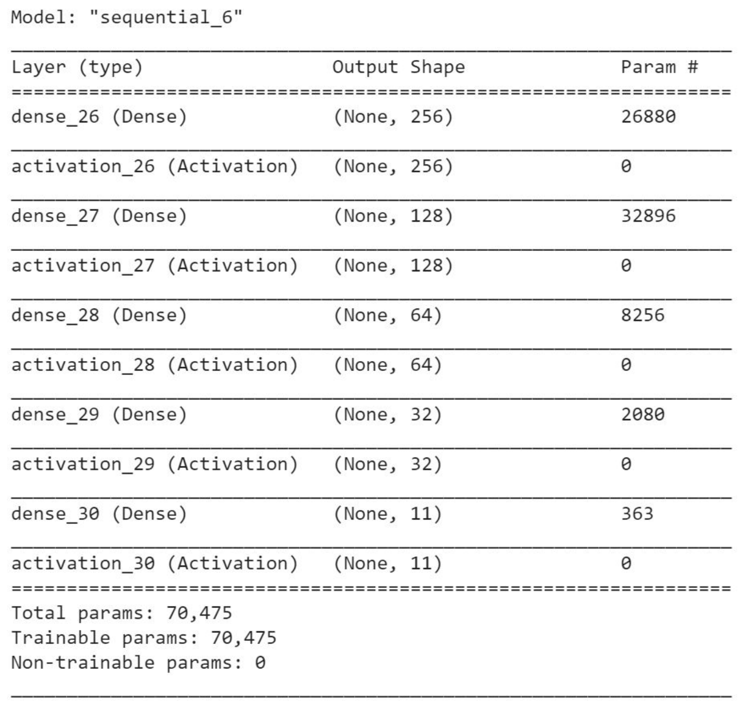


```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 8, 47, 47]             136
            Conv2d-2           [-1, 16, 23, 23]           1,168
            Conv2d-3           [-1, 32, 11, 11]           4,640
            Conv2d-4             [-1, 64, 5, 5]          18,496
            Conv2d-5            [-1, 128, 3, 3]          73,856
            Conv2d-6            [-1, 256, 1, 1]         295,168
            Linear-7                  [-1, 100]          25,700
              ReLU-8                  [-1, 100]               0
            Linear-9                    [-1, 3]             303
        Softplus-10                     [-1, 3]               0
          Linear-11                     [-1, 3]             303
        Softplus-12                     [-1, 3]               0
================================================================
Total params: 419,770
Trainable params: 419,770
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.04
Forward/backward pass size (MB): 0.25
Params size (MB): 1.60
Estimated Total Size (MB): 1.89
----------------------------------------------------------------
```

**3. Q-Learning with Fully Connected Network**
We were also working concurrently on another hypothesis which inspired us to think beyond raw

pixel data. We used features taken from patches around the car and patches around the track and flatten them into a 1D vector which is fed to a fully connected neural net. This hypothesis revolves around learning more accurate representation of the screen in lesser number of parameters.[2]

**Experimental details:**
For this implementation, after reading work by Luc Prieur, we decided to downsize our model complexity and increase the importance of the features we fed into the model. Below, you can see our model which consists of 5 fully-connected layers.



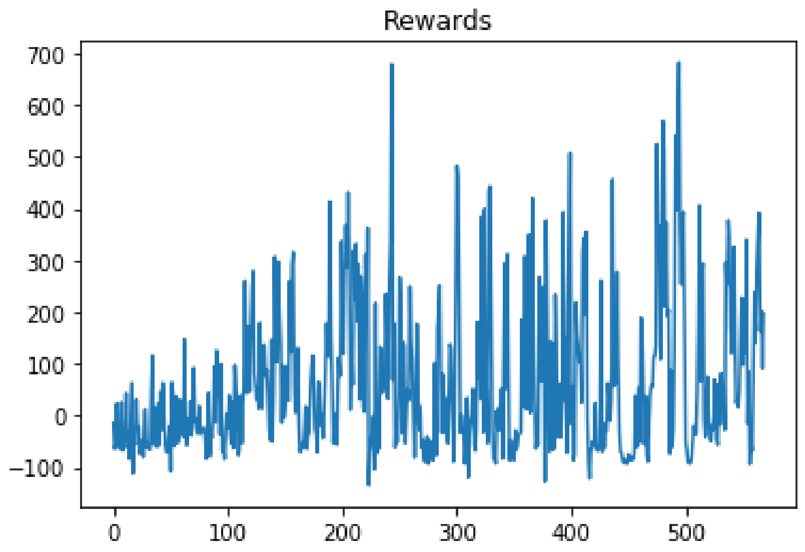
```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_26 (Dense)             (None, 256)               26880     
_________________________________________________________________
activation_26 (Activation)   (None, 256)               0         
_________________________________________________________________
dense_27 (Dense)             (None, 128)               32896     
_________________________________________________________________
activation_27 (Activation)   (None, 128)               0         
_________________________________________________________________
dense_28 (Dense)             (None, 64)                8256      
_________________________________________________________________
activation_28 (Activation)   (None, 64)                0         
_________________________________________________________________
dense_29 (Dense)             (None, 32)                2080      
_________________________________________________________________
activation_29 (Activation)   (None, 32)                0         
_________________________________________________________________
dense_30 (Dense)             (None, 11)                363       
_________________________________________________________________
activation_30 (Activation)   (None, 11)                0         
=================================================================
Total params: 70,475
Trainable params: 70,475
Non-trainable params: 0
_________________________________________________________________
```

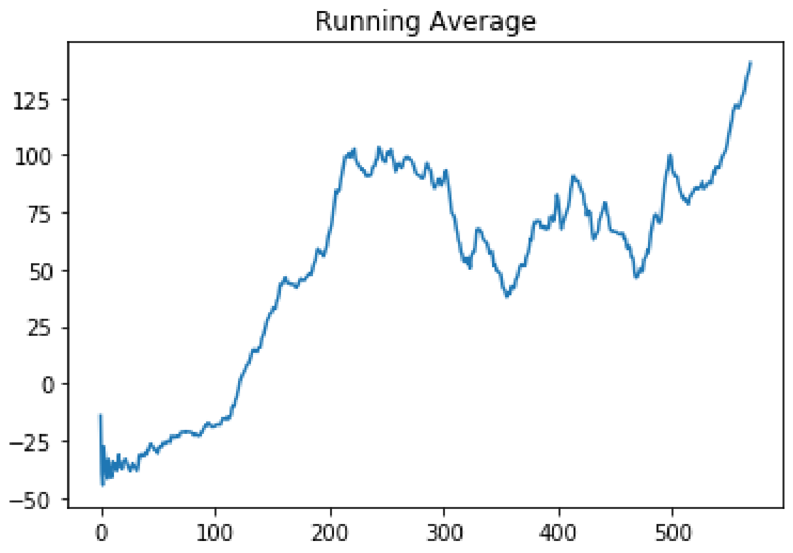
From here, we decided to focus on featurization of each screen to be compatible with an FCN while maintaining crucial pieces of information from the image. We decided to take the patch of image around the car and around the track (refer Figure 3), transformed them to a grayscale image and flattened them into a 1D feature vector to be fed into the FCN. This was in order to learn the surroundings of the car and the track surroundings when a good reward was achieved.

Then, we also decided to segment the action space differently; with this method, we took the index of the highest Q-value and translated it to either a steering direction, forward thrust or stop action. This turned out to give 11 actions (8 different steering, 1 forward, 1 brake) (refer Figure 2). This was done in order to have a much more diverse action space with various steering angles (didn't seem to need other gas and brake rates). Each action was also randomly sampled during certain steps using the parameter epsilon which was decayed just as it was in the DQN step.

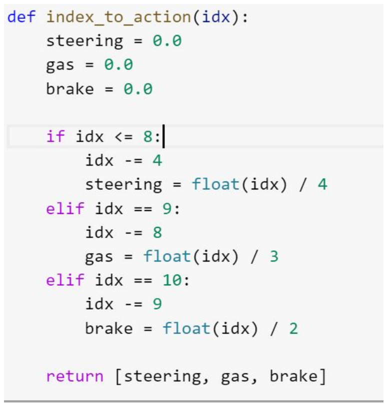
From there, we wrote a normal training loop (which we ended after 570 episodes) with Q-updates every environment step using the target net lagging behind by one step (as is custom in DQN). We used the Adamax optimizer provided with Tensorflow because we saw that it works better than Adam in certain cases especially in models with latent representations of images.

Below, you can see our network's rewards over training iterations and running average over the previous 100 runs.



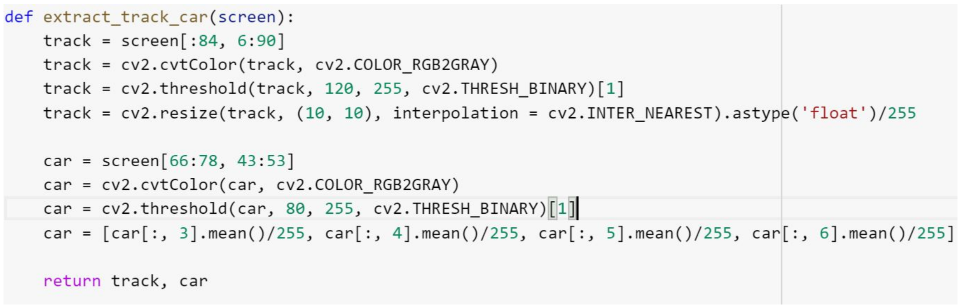


Some critical functions for selecting action based on index and pre-processing of images are given as follows:



```python
def index_to_action(idx):
    steering = 0.0
    gas = 0.0
    brake = 0.0

    if idx <= 8:
        idx -= 4
        steering = float(idx) / 4
    elif idx == 9:
        idx -= 8
        gas = float(idx) / 3
    elif idx == 10:
        idx -= 9
        brake = float(idx) / 2

    return [steering, gas, brake]
```

Fig. 2: Action function for selecting action based on Q-index

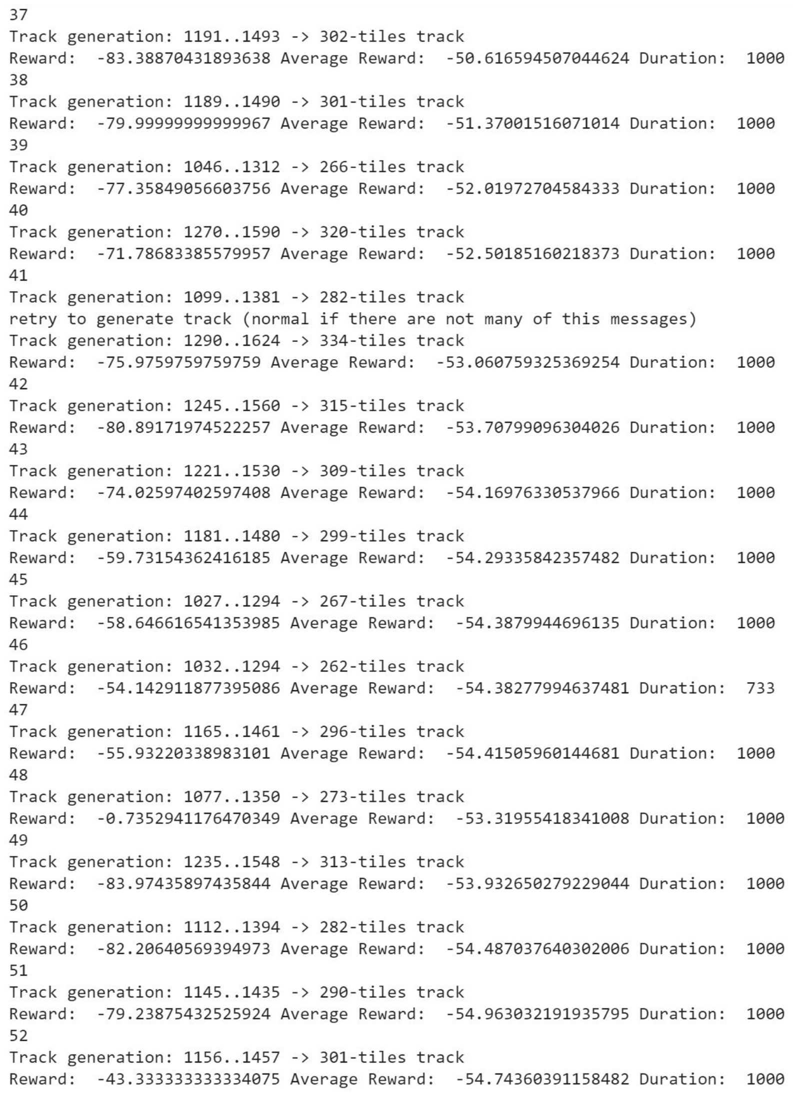


```python
def extract_track_car(screen):
    track = screen[:84, 6:90]
    track = cv2.cvtColor(track, cv2.COLOR_RGB2GRAY)
    track = cv2.threshold(track, 120, 255, cv2.THRESH_BINARY)[1]
    track = cv2.resize(track, (10, 10), interpolation = cv2.INTER_NEAREST).astype('float')/255

    car = screen[66:78, 43:53]
    car = cv2.cvtColor(car, cv2.COLOR_RGB2GRAY)
    car = cv2.threshold(car, 80, 255, cv2.THRESH_BINARY)[1]
    car = [car[:, 3].mean()/255, car[:, 4].mean()/255, car[:, 5].mean()/255, car[:, 6].mean()/255]

    return track, car
```

Fig. 3: Function for extracting patch around the car

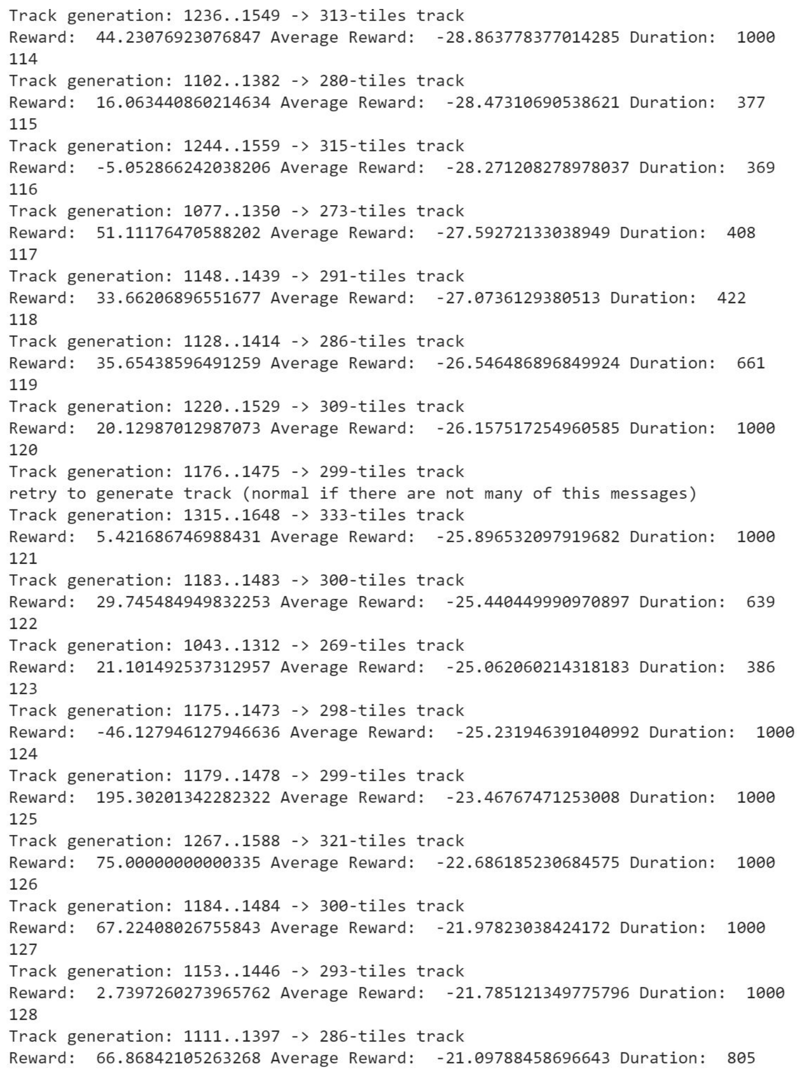
## 4 Appendix

### 4.1 Failure cases



```
37
Track generation: 1191..1493 -> 302-tiles track
Reward:  -83.38870431893638 Average Reward:  -50.616594507044624 Duration:  1000
38
Track generation: 1189..1490 -> 301-tiles track
Reward:  -79.99999999999967 Average Reward:  -51.37001516071014 Duration:  1000
39
Track generation: 1046..1312 -> 266-tiles track
Reward:  -77.35849056603756 Average Reward:  -52.01972704584333 Duration:  1000
40
Track generation: 1270..1590 -> 320-tiles track
Reward:  -71.78683385579957 Average Reward:  -52.50185160218373 Duration:  1000
41
Track generation: 1099..1381 -> 282-tiles track
retry to generate track (normal if there are not many of this messages)
Track generation: 1290..1624 -> 334-tiles track
Reward:  -75.9759759759759 Average Reward:  -53.060759325369254 Duration:  1000
42
Track generation: 1245..1560 -> 315-tiles track
Reward:  -80.89171974522257 Average Reward:  -53.70799096304026 Duration:  1000
43
Track generation: 1221..1530 -> 309-tiles track
Reward:  -74.02597402597408 Average Reward:  -54.16976330537966 Duration:  1000
44
Track generation: 1181..1480 -> 299-tiles track
Reward:  -59.73154362416185 Average Reward:  -54.29335842357482 Duration:  1000
45
Track generation: 1027..1294 -> 267-tiles track
Reward:  -58.646616541353985 Average Reward:  -54.3879944696135 Duration:  1000
46
Track generation: 1032..1294 -> 262-tiles track
Reward:  -54.142911877395086 Average Reward:  -54.38277994637481 Duration:  733
47
Track generation: 1165..1461 -> 296-tiles track
Reward:  -55.93220338983101 Average Reward:  -54.41505960144681 Duration:  1000
48
Track generation: 1077..1350 -> 273-tiles track
Reward:  -0.7352941176470349 Average Reward:  -53.31955418341008 Duration:  1000
49
Track generation: 1235..1548 -> 313-tiles track
Reward:  -83.97435897435844 Average Reward:  -53.932650279229044 Duration:  1000
50
Track generation: 1112..1394 -> 282-tiles track
Reward:  -82.20640569394973 Average Reward:  -54.487037640302006 Duration:  1000
51
Track generation: 1145..1435 -> 290-tiles track
Reward:  -79.23875432525924 Average Reward:  -54.963032191935795 Duration:  1000
52
Track generation: 1156..1457 -> 301-tiles track
Reward:  -43.333333333334075 Average Reward:  -54.74360391158482 Duration:  1000
```

Fig. 4: The training trend for the first few episodes: Vanilla DQN: more negative rewards per episode



```
Track generation: 1236..1549 -> 313-tiles track
Reward:  44.23076923076847 Average Reward:  -28.863778377014285 Duration:  1000
114
Track generation: 1102..1382 -> 280-tiles track
Reward:  16.063440860214634 Average Reward:  -28.47310690538621 Duration:  377
115
Track generation: 1244..1559 -> 315-tiles track
Reward:  -5.052866242038206 Average Reward:  -28.271208278978037 Duration:  369
116
Track generation: 1077..1350 -> 273-tiles track
Reward:  51.11176470588202 Average Reward:  -27.59272133038949 Duration:  408
117
Track generation: 1148..1439 -> 291-tiles track
Reward:  33.66206896551677 Average Reward:  -27.0736129380513 Duration:  422
118
Track generation: 1128..1414 -> 286-tiles track
Reward:  35.65438596491259 Average Reward:  -26.546486896849924 Duration:  661
119
Track generation: 1220..1529 -> 309-tiles track
Reward:  20.12987012987073 Average Reward:  -26.157517254960585 Duration:  1000
120
Track generation: 1176..1475 -> 299-tiles track
retry to generate track (normal if there are not many of this messages)
Track generation: 1315..1648 -> 333-tiles track
Reward:  5.421686746988431 Average Reward:  -25.896532097919682 Duration:  1000
121
Track generation: 1183..1483 -> 300-tiles track
Reward:  29.745484949832253 Average Reward:  -25.440449990970897 Duration:  639
122
Track generation: 1043..1312 -> 269-tiles track
Reward:  21.101492537312957 Average Reward:  -25.062060214318183 Duration:  386
123
Track generation: 1175..1473 -> 298-tiles track
Reward:  -46.127946127946636 Average Reward:  -25.231946391040992 Duration:  1000
124
Track generation: 1179..1478 -> 299-tiles track
Reward:  195.30201342282322 Average Reward:  -23.46767471253008 Duration:  1000
125
Track generation: 1267..1588 -> 321-tiles track
Reward:  75.00000000000335 Average Reward:  -22.686185230684575 Duration:  1000
126
Track generation: 1184..1484 -> 300-tiles track
Reward:  67.22408026755843 Average Reward:  -21.97823038424172 Duration:  1000
127
Track generation: 1153..1446 -> 293-tiles track
Reward:  2.7397260273965762 Average Reward:  -21.785121349775796 Duration:  1000
128
Track generation: 1111..1397 -> 286-tiles track
Reward:  66.86842105263268 Average Reward:  -21.09788458696643 Duration:  805
```

Fig. 5: The training trend for few intermediate episodes: Vanilla DQN: more positive rewards per episode

## References

1. Juliani a., simple reinforcement learning with tensorflow, medium
2. Aldape, P., Sowell, S.: Reinforcement learning for a simple racing game
3. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym (2016)
4. Ha, D., Schmidhuber, J.: World models. arXiv preprint arXiv:1803.10122 (2018)
5. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529 (2015)
6. Raffin, A., Sokolkov, R.: Learning to drive smoothly in minutes. `https://github.com/araffin/learning-to-drive-in-5-minutes/` (2019)
7. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms (2017)