UNIVERSITY OF TARTU
Institute of Computer Science
Neural Networks

2D Racing game using reinforcement learning and supervised learning

Henry Teigar
University of Tartu
henry.teigar@gmail.com

Miron Storožev
University of Tartu
miron.storozhev96@gmail.com

Janar Saks
University of Tartu
janarsaks@hotmail.com

Abstract

Last year (2017) Tesla introduced their brand new electric car "Roadster", which can achieve maximum speed of 402 km/h. That is faster than highest speed ever recorded in Formula 1 race. Do you think humans can handle this speed and use this car at its full potential? Probably not. However computers can. Using modern sensors and processors with high processing power and speed, self driving cars can make decisions (turning wheel, pressing gas or brake pedal) much faster than humans, preventing numerous accidents and producing better overall driving experience. Greg Votolato, a design history expert and tutor at the Royal College of Art Vehicle Design program says, that 200 mph (321 km/h) isn't much to expect from self driving cars.[1]

Entire concept of self driving cars evolves around machine learning and most commonly around neural networks. It is essential for self driving cars manufacturers to enhance the utilization of machine learning to teach machines tasks like avoiding obstacles, staying on track and driving in general.

[1] Dyani Sabin, "How Fast Will Autonomous Cars Go? 200 MPH," (2017, February). [Online]. https://www.inverse.com/article/28110-self-driving-car-speed-future

Tartu 2017

1. Introduction

The problem with reinforcement learning is that it is not reasonable to train the network using the real world environment with physical car, as the learning process involves failures (crashes). So it is essential to have a virtual simulator. And our goal is to do exactly that!

Our initial idea was to create a simple 2D racing simulator using Pygame that uses just the distance sensors as inputs to our reinforcement network with the aim to train a model that is capable of completing the circuit multiple times without errors. We also decided to train another model using a premade OpenAI Gym environment and this time by using the pixel values as inputs. To make things more interesting we try to train the same model also using the supervised learning. In this paper we focus on the different approaches we took, the problems and successes we encountered.

2. Background/Related Work

Related work

Reinforcement Learning has been applied to a variety of problems, such as robotic obstacle avoidance and visual navigation. Deep Reinforcement Learning (DRL), a combination of reinforcement learning with deep learning has shown unprecedented capabilities at solving tasks such as playing Atari games or the game of Go.[2] The most common articles we found about reinforcement were also about using reinforcement learning on games. All works and articles that are talking about usage of Neural Networks in games are related to our work. Most popular articles that we found were "Deep Reinforcement Learning: Pong from Pixels" by Andrej Karpathy[3] and "Write an AI to win at Pong from scratch with Reinforcement Learning" by Dhruv Parthasarathy[4]. In these articles it is described how to apply Reinforcement Learning on simple Atari game Pong, which is similar to what we were trying to do.
We also took inspiration from DeepMind network architecture as described in "Demystifying deep reinforcement learning" by Tambet Matiisen

2.1 Reinforcement Learning

The basic idea behind the reinforcement learning is that the computer learns on its own by trial and error and therefore it is not necessary to have a huge dataset before the start of the training. That is one of the biggest advantages of reinforcement learning.
Although there are many different approaches to implement the network, there are still some core characteristics of reinforcement learning. It always consists of an agent and an environment. An agent takes into account the current state of environment and based on that, performs an action that yields a reward and mutates the state of the environment. This cycle repeats again and again. An agent chooses its actions with a goal to maximize

[2] M. Pandey, D. Shen, A. Pancholi , "Deep Reinforcement Learning using Memory-based Approaches" , http://cs231n.stanford.edu/reports/2017/pdfs/618.pdf

[3] Andrej Karpathy, "Deep Reinforcement Learning: Pong from Pixels", http://karpathy.github.io/2016/05/31/rl/

[4] Dhruv Parthasarathy, "Write an AI to win at Pong from scratch with Reinforcement Learning", https://medium.com/@dhruvp/how-to-write-a-neural-network-to-play-pong-from-scratch-956b57d4f6e0

the long-term reward. However there are multiple challenges with this idea. One being the credit assignment problem. It is very likely that the action that lead us to the positive or negative reward was done many steps ago. This makes it difficult to encourage/discourage actions that were responsible for the success or failure. Another major difficulty in implementing the reinforcement learning is the explore-exploit-dilemma. When our model suffers under this problem, it gets stuck at lower score (local minimum) and is happy about it without knowing that it could perform better.[5]

2.2 Markov decision process

All the different approaches of reinforcement learning use a common mathematical formulation of how the agent interacts with the environment. The environment is modelled as a Markov Decision Process (MDP), which works as follows: There is a set of actions $\boldsymbol{A}$ and a set of states $\boldsymbol{S}$. By performing some action $a \in A$, the agent can move from state to state.

At each time step, the process is in some state s, and the decision maker may choose any action $a \in A$that is available in state s. This gives us a probability distribution of transitions to next states and also a probability distribution of rewards. So the probability that the process moves into its new state $\boldsymbol{s'}$ is influenced by the chosen action. Specifically, it is given by the state transition function $\boldsymbol{Pa}(\boldsymbol{s}, \boldsymbol{s'})$. Thus, the next state $\boldsymbol{s'}$ depends on the current state $\boldsymbol{s}$ and the decision maker's action $\boldsymbol{a}$. But given $\boldsymbol{s}$ and $\boldsymbol{a}$, it is conditionally independent of all previous states and actions.[6]

2.4 Policy Gradients

Policy gradients is one of the most popular approaches, besides the deep Q network, to implement a reinforcement network.

The main difference between Q-learning and Policy Gradients is that instead of parameterizing the value function and doing policy improvement we parameterize the policy and do gradient descent into a direction that improves it. [7]
So policy gradients works in a way that it takes in a state and yields probabilities for every action. Then an action is chosen based on the probabilities and after series of chosen actions, we will receive a positive or negative reward. We can find the gradient that points to the parameter space, where we change the parameters so that, when we are on the same state again, the probability of the chosen action is also changed. This means that actions with negative reward or positive reward in certain states will be discouraged or encouraged respectively by the network.

2.5 Supervised learning

Supervised learning is a type of machine learning algorithm that uses a known dataset (called the training dataset) to make predictions. The training data consist of a set of training examples where each example is a pair consisting of an input object and a desired output value. A supervised learning algorithm analyses the training data and

[5] T Matiisen. "Demystifying deep reinforcement learning", http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/

[6] Wikipedia, "Markov decision process", https://en.wikipedia.org/wiki/Markov_decision_process

[7] "Policy Gradient Methods", https://github.com/dennybritz/reinforcement-learning/tree/master/PolicyGradient

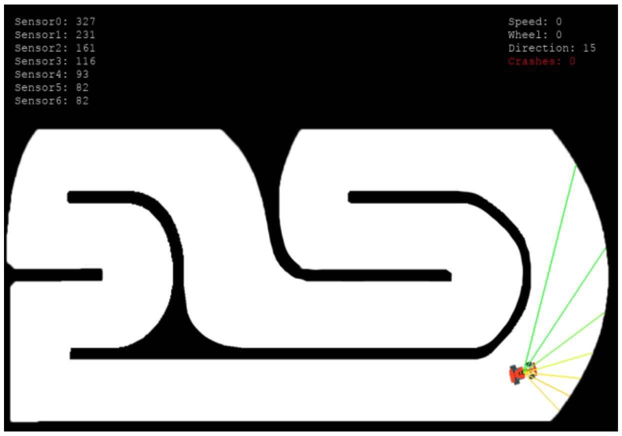
produces an inferred function, which can be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way. [8]

3. Simulation environments

Under this section we cover all the environments that we used throughout this project. These involve also environments that were not directly part of the project, but helped us to better understand different models and networks.

3.1 Self-made environment

As mentioned previously our initial goal was only to build a custom 2d driving simulator using Pygame. Instead of the regular approach to use all the pixels as an input, we extracted distance information from walls with seven sensors which all point into different directions (*image 1*).



```
Sensor0: 327                                Speed: 0
Sensor1: 231                                Wheel: 0
Sensor2: 161                                Direction: 15
Sensor3: 116                                Crashes: 0
Sensor4: 93
Sensor5: 82
Sensor6: 82
```

Image 1 - Gameplay screenshot from our environment

To make the communication between the neural network and the game easier and similar to the OpenAI Gym environments system, we decided to build an outer layer around the Pygame with PyGame-Learning-Environment (PLE)[9]. After the implementation of the methods we could control the game with functions like step (OpenAi Gym equivalent to act), getScore, getGameState etc. There are two possible options: turn left and turn right. PLE also allowed us to run the simulation without the graphical output, so we could train the network much faster.

3.2 OpenAI Gym - CarRacing-v0

It is a top down racing environment (*Image 2*) where the state consists of 96x96 pixels. Environment gives a negative reward of 0.1 for every frame and a positive reward 1000/N for every track tile visited, where N is the total number of tiles in track. Episode finishes when all tiles are visited.[10] In this paper under Approach section we will discuss more about this reward system and how we altered it.

[8] Wikipedia, "Supervised learning", https://en.wikipedia.org/wiki/Supervised_learning

[9] "PyGame Learning Environment (PLE) -- Reinforcement Learning Environment in Python." https://github.com/ntasfi/PyGame-Learning-Environment

[10] CarRacing-v0 (experimental), https://gym.openai.com/envs/CarRacing-v0/

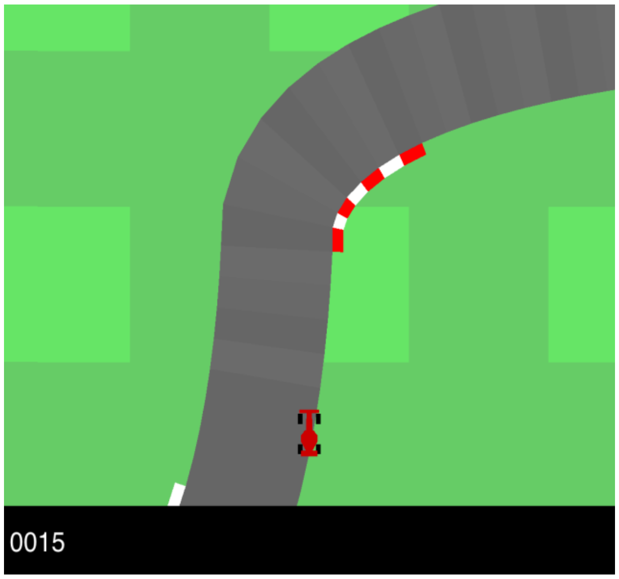


Image 2 - Gameplay screenshot from CarRacing-v0 environment

3.3 OpenAI Gym - CartPole-v0 and Breakout-v0

We will not start to describe these environments very detailed, as they were the side projects we used just for learning the main concepts behind the networks.
But the main idea behind these environments was that they were in some view quite similar to our main environments. For example CartPole-v0 gives us quite similar output about the environment, as our self-made racing simulator - only limited amount of data (in CartPole case, for example the angle of the pole, position of the cart etc.) [11]. So when we will manage to train the CartPole environment, we most probably will be able to train our environment also with similar model. The same goes with Breakout-v0 [12], which in this case is more similar to CarRacing-v0 where the whole pixel image is used as the input.

[11] CartPole-v0. https://gym.openai.com/envs/CartPole-v0/
[12] Breakout-v0. https://gym.openai.com/envs/Breakout-v0/

4. Approach

As mentioned previously we took quite a different approach for the two main environments that we used - self-made environment and CarRacing-v0. Under this section we are going to cover these in more detail.

4.1. Self-made environment approach

4.1.1 Reward system

For this environment we used a simple approach, where every frame, the car hasn't crashed, gives a positive reward +1. And the goal is to earn as much reward as possible by the end of the episode. The episode ends with the car crashing into the wall.

4.1.1 Reinforcement learning model

We used Policy Gradient method to train our model. Our model looked like this:

1. INPUT: Sensor data (shape = (7,))
2. DENSE: 100 features (activation=tanh)
3. DENSE: 25 features (activation=tanh)
4. DENSE: 2 features (activation=softmax)

So the output of the model is 2 - probability distribution for turning left or right (the actions just turn the wheel and therefore it is possible that for example with the turning left action the car is still turning right, just a little bit less)

4.1.2. Overview of the algorithm (simplified version)

```
rewards, actions, input = [], [], []
while True:
 input_sensors = getGameState()
```

```
action_probs =model.predict(input_sensors)
probs = action_probs /np.sum(action_probs)
action = getActionByProb(probs)
reward, crashed = act(action)

rewards.append(reward)
actions.append(action)
input.append(input_sensors)
if crashed:
  discounted_rewards = discount(rewards)
  advantage = rewards - np.mean(rewards)
  model.train_on_batch(inputs, actions,
                  sample_weight=advantage)
  inputs, actions, rewards = [], [], []
```

4.2 CarRacing-v0

For CarRacing-v0 environment our initial idea was to only implement the reinforcement learning algorithm. Later we also performed a supervised learning process. Under this section we are going to briefly talk about both of these approaches.

4.2.1. Reinforcement learning on CarRacing-v0

We tested with lots of different models and ideas that we cover more thoroughly in this paper under the experiments and results section.

4.2.1.1. Reward system and actions

By default the environment ends the episode only when the track is completed. As in most cases we tried the car drove off the circuit at the very beginning of the episode, we decided to modify the default reward system and made it so, that when the car has gathered negative reward (doesn't pass any track tiles) long enough, we force to restart the episode. In our opinion it makes the training more likely and faster.

Also as the environment accepts 3 different action values, each with a scale from -1 to 1, we decided to create just four different possible actions: [[1.0, 0.3, 0.0], [0.0, 1.0, 0.0], [-1.0, 0.3, 0.0], [0.0, 0.0, 0.8]] where the first value in each list is steering value, the second one is for throttle and the third represents the breaking.

4.2.1.2. Image preprocessing

As our final model uses convolutional layers, we didn't put much emphasis on image preprocessing, other than simple cropping as many pixels as we could without losing any potentially important information. We also took away RED and BLUE color channels, because with removing these, we didn't lose any valuable information because the image is mostly green and gray.

4.2.1.3. Reinforcement learning model

Our final model was with the following structure:

1. INPUT: Image with shape=(80,64,1)
2. CONV2D(32, 8, strides=4, activation=relu)
3. CONV2D(64, 4, strides=2, activation=relu)
4. CONV2D(64, 3, strides=1, activation=relu)
5. DENSE(512, activation=relu)
6. DENSE(4, activation=softmax)
   Baseline value
7. DENSE(100, activation=relu)
8. DENSE(1)

We also used entropy calculation in our custom loss function in a hope to make the car more likely to turn in the corners.

The overall algorithm idea is quite similar with what we created during the training process of our own environment. One of the main differences is that in case of this game, before we feed the input to our network we subtract the previous frame image from the current one in order to express also the movement of objects.

4.2.2. Supervised learning on CarRacing-v0

As we weren’t so pleased with the results of our reinforcement learning with CarRacing-v0 (more on that later), we also tried to train the same environment with supervised learning.

4.2.2.1. Dataset

As we did not have any pre-recorded data (to be honest, we even did not search, because the idea to record the data by ourselves, sounded very tempting), so we programmed a simple Pygame program, that interacts with the gym environment. This allowed us to play the game simply by arrow keys and also constantly record the data in the background. We used Pygame, because of the great key-listener capabilities.

We recorded 30 frames per second for about 25 minutes and gathered about 40 000 frames of data. Each row of data included the image pixels and the selected action.

4.2.2.2. Image preprocessing

The input image is initially exactly the same, as mentioned under the reinforcement learning part. But mainly because we had to save a huge amount of data, we tried to reduce the size of the input image, as much as possible. The initial image has a shape (96, 96, 3), which results in 27 648 parameters. We again cropped the image, took away two color channels, but this time also took away every 3rd pixel from height and every 4th pixel in width. Also we reduced the number of different color shades (for example the grass contains multiple colors). That all resulted in a final parameter count of 2544 (*Image 3 and Image 4*)

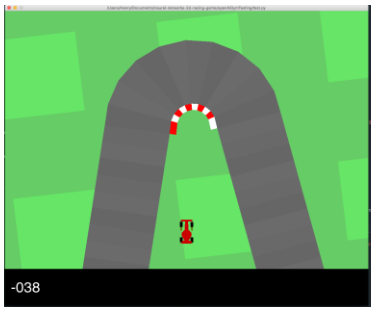


Image 3 - Without preprocessing

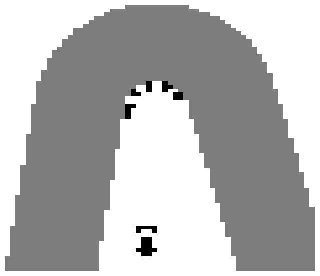


Image 4 - With preprocessing

4.2.2.2. Supervised learning model

INPUT - with shape: (53,48,1) - the image was reshaped back to matrix
CONV2D(32, 16)
BatchNormalization (BN)
Activation(“relu”)
CONV2D(64, 8)
BN
Activation(“relu”)
Dropout(0.25)
CONV2D(64,8)
DENSE(512)
BN
Activation(“relu”)
Dropout(0.5)
Dense(4) - as we had 4 possible actions

For our loss function, we used categorical_crossentropy.

5. Experiments and results

5.1. Self-made environment training results and remarks

The model learned surprisingly quickly that in right corners, it is wise to turn right and in left corners, it is wise to turn left. However it took some time to reach to the point where we could say that the model learned the environment and could drive almost error free. If we look at *Image 5*, which represents the average score and episode count, we can see the exploding raise in the score around episode 140. It happened just because the model finally learned to take each corner almost perfectly and as the maximum score can be virtually infinite, then it took hundreds of laps for our trained model to finally make a mistake. You can observe the final result here: Youtube

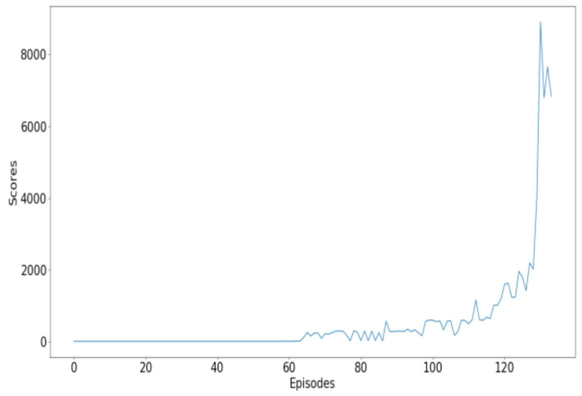


Image 5 - Episodes and Scores when training with our environment

5.2. CarRacing-v0 environment with reinforcement learning results and remarks

As we mentioned previously in our paper, that before starting to train with the CarRacing-v0 environment, we tried a very similar model and algorithm on the Breakout-v0 environment. We trained about 10 hours on our own laptops (just for testing) and we could see that the model trains decently *(Image 6)*.

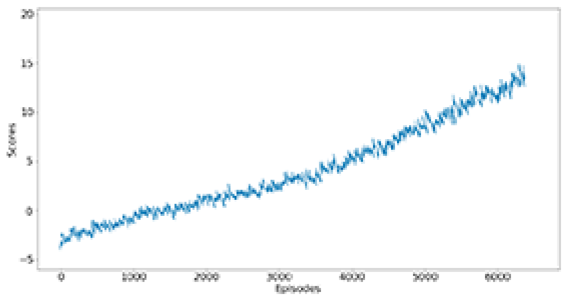


Image 6 - Episodes and Scores when training the breakout environment

The things were a little bit different with CarRacing-v0 environment. It seemed, that the initial training (also about 10 hours), didn't actually train almost at all (*Image 7*). When we looked at the actual gameplay, we could see that the car drove only straight and in corners didn't even try to turn. We investigated things further and in several cases, when we restarted the whole learning process, it got stuck by always turning left. So we most probably experienced the explore-exploit dilemma. In order to reduce that, we tried to increase the batch size (so it could see more possible

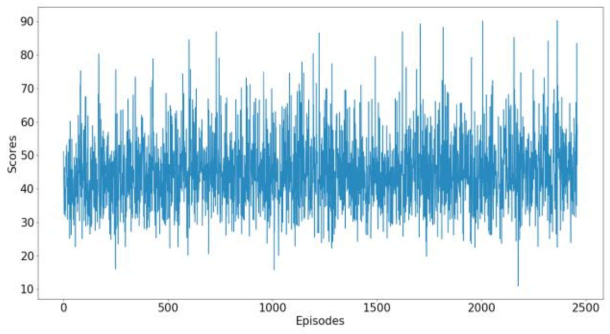


Image 7 - Episodes and Scores when training the CarRacing-v0 environment with initial model options before updating the weights) and we also started to use entropy equation in our custom loss function. We tried different values for the entropy multiplier and it seemed, that we got the best results when using 0.1 multiplier before the entropy addition. We also introduced a better baseline calculation. With all that and lots and lots of testing with different parameters, we got a result like in *Image 8*.

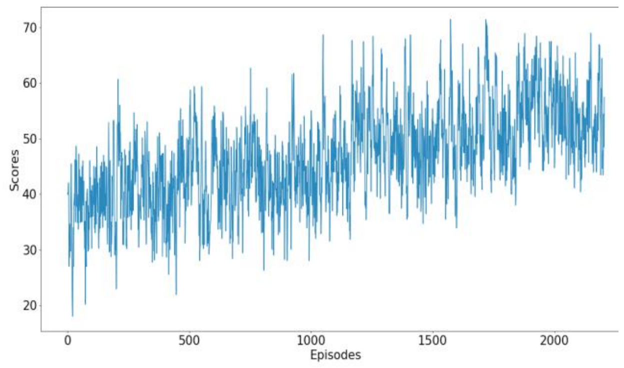


Image 8 - Episodes and Scores when training the CarRacing-v0 environment with new model

It seemed to be a little bit better, but not by a big factor. And the scores went up and down always very rapidly (a little bit less than in our initial model though). The actual gameplay was improved a little bit - now before the left turns, the car slightly turned to left and the same thing with right corners (only to the right), but after that the car went straight to the grass and the episode restarted.

What makes CarRacing-v0 much more challenging than for example the Breakout, is that the track has a lot of long straight roads (also in the beginning) and this can easily lead to agent learning, that going straight gives always the biggest reward.

One of the possible solutions which we thought about is to give an extra positive reward when the agent is turning. This could make it more likely, that the car wants to turn more, but in order to still get the big reward which will be given by crossing the next track tile, the car should still want to stay in track.

5.3. CarRacing-v0 environment with supervised learning results and remarks

When training the model, we experimented quite a lot with different parameters (batch_size, epoch count etc.). In the beginning we constantly trained a model which either turned always to the left or drove always straight. We realised that most probably we overtraining our model. One of the biggest improvement made when we reduced the number of epochs to train.

Our final model was able to follow more or less the road curvage, even though sometimes it made a wrong decision and in some cases the model wanted to drive beside the road, not on the road, but it still followed the curvage.

We uploaded two of the best models to youtube: Link1 and Link2

When analysing the performance of the model, we have to take into consideration that the data, which the model trained on, is not perfect. When we were manually driving for about 25 minutes to gather data, the PyGame key-listener, after about every 30 seconds did not register the key release (we are not sure what caused the glitch), and that resulted us to drive out of the rode every now and then. Also as CarRacing-v0 environment seems to be built in a way that with constant throttle and no breaking the speed of the car always increases (of course to the certain point), then it was almost impossible for us to always drive with the same speed when we were at process of creating the dataset. And going into a corner with different throttle values, results in very different turning speeds. Also the circuit has very long straights and this means that the majority of the data is probably just about driving straight.

All that said, we were actually really impressed that with the final model. The car was actually able to make most of the decisions correctly and it really was able to train with this kind of training data.

We think that maybe we would have gotten even better results, if we would have preprocessed the training data. Currently as we said, most of the actions are going straight actions. Maybe if we would have sampled the data in such way that we get almost all actions equal amounts, then maybe the model would have trained better.

6. Conclusion

In this paper, we tried to implement two reinforcement learning algorithm and one supervised learning algorithm.

Neural networks can definitely be quite challenging to get to work, but overall we are quite satisfied with the results. We managed to achieve our primary goal: make the car in our self-made environment drive almost perfectly around the track. We also managed to train the model with supervised learning to play CarRacing-v0 by itself reasonably well. The reinforcement learning approach for CarRacing-v0 was not so successful but as we suggested, changing the reward system should improve the performance.

Author's personal note:
As we all three are bachelor's students, this was the first course ever that we took that focuses on machine learning, let alone neural networks. Therefore it was extremely exciting to learn this new world and this project definitely helped us to get a much better understanding about the main concepts of neural networks and about machine learning in general.

7. References

[1] Dyani Sabin, "How Fast Will Autonomous Cars Go? 200 MPH", https://www.inverse.com/article/28110-self-driving-car-speed-future
[2] M. Pandey, D. Shen, A. Pancholi , "Deep Reinforcement Learning using Memory-based Approaches",
http://cs231n.stanford.edu/reports/2017/pdfs/618.pdf
[3] Andrej Karpathy, "Deep Reinforcement Learning: Pong from Pixels",
http://karpathy.github.io/2016/05/31/rl/
[4] Dhruv Parthasarathy, "Write an AI to win at Pong from scratch with Reinforcement Learning",
https://medium.com/@dhruvp/how-to-write-a-neural-network-to-play-pong-from-scratch-956b57d4f6e0
[5] Ted Li, Sean Rafferty, "Playing Geometry Dash with Convolutional Neural Networks",
http://cs231n.stanford.edu/reports/2017/pdfs/605.pdf
[6] T Matiisen. "Demystifying deep reinforcement learning",
http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/
[7] Wikipedia, "Markov decision process",
https://en.wikipedia.org/wiki/Markov_decision_process
[8] "Policy Gradient Methods",
https://github.com/dennybritz/reinforcement-learning/tree/master/PolicyGradient
[9] Wikipedia, "Supervised learning",
https://en.wikipedia.org/wiki/Supervised_learning
[10] "PyGame Learning Environment (PLE) -- Reinforcement Learning Environment in Python."
https://github.com/ntasfi/PyGame-Learning-Environment
[11] CarRacing-v0 (experimental).
https://gym.openai.com/envs/CarRacing-v0/
[12] CartPole-v0.
https://gym.openai.com/envs/CartPole-v0/
[13] Breakout-v0.
https://gym.openai.com/envs/Breakout-v0/