Innovation 2020
Organised by
Computer Engineering Department, Dr. D. Y. Patil School of Engineering, Lohegaon, Pune, Maharashtra, India in association with
International Journal of Scientific Research in Science, Engineering and Technology

Autonomous vehicle controls using Reinforcement Learning

Mukesh Iyer, Jai Baheti, Rajmohan Bajaj, Nilesh Nanda, Dr. Sunil Rathod
Department of Computer Engineering, Dr. D. Y. Patil School of Engineering, Lohegaon Savitribai Phule Pune University, Pune, Maharashtra, India

ABSTRACT

In this paper, we explore a reinforcement learning algorithm to train an agent to drive a vehicle in the OpenAI Gym environment called CarRacing-v0. Gym is an open-source repository created by OpenAI which provides a toolkit for developing and comparing various Reinforcement learning algorithms. The gym library is a collection of environments that can be used to work out reinforcement learning algorithms. Learning to drive in the CarRacing-v0 environment is challenging since it requires the agent to finish the continuous control task by learning from pixels. To tackle this challenging problem, we explored an approach called Deep Q Learning. In this paper we further demonstrate a method to train the agent which learns from raw pixels without providing any hand-crafted features. Some minor environment specific changes were made but the base agent was not provided any knowledge regarding car racing.

Keywords : OpenAI Gym, Reinforcement Learning, CarRacing-v0

I. INTRODUCTION

Now-a-days self driving cars are more and more popular for quick transportation, safety and economic advantages but these cars would only follow orders about destination and route, and may only adopt some lane-tracking or car-following guidance whereas in order to make autonomous driving a truly ubiquitous technology, paper advocates for agents that can learn the ability to drive and navigate in absence of maps and explicit rules, relying just like humans, on a comprehensive understanding of the immediate environment and the various objects in the environment, predict their possible future behaviors and interactions, and then plan how to control it in order to safely move closer to their desired destination while obeying the rules of the environment. This is a difficult challenge for machines that humans solve well, contributing to knowledge. Thus making reinforcement learning a promising approach. Reinforcement Learning is an area of machine learning, aiming at learning the optimal behaviour in an environment by maximizing the cumulative reward. The concept of deep reinforcement learning was introduced recently and was tested with success in games like Go [1] or Atari 2600 [2], proving the capability to learn and understand a good representation of the environment. Reinforcement Learning allows the agent to learn its behaviour based on feedback that is received from the environment. This behaviour can be learnt at the beginning once and for all, or keep on adapting as time goes by. If the problem is modelled properly, some Reinforcement Learning algorithms can perform remarkably well and converge to the global optimum; this is the ideal behaviour that maximises the reward. This automated learning scheme implies that there is no or little need for a human. The time spent on designing a solution will be less, since there is no requirement for hand crafting complex sets of rules as with Expert Systems. The motive of this paper is to train an agent using the Deep Q Learning algorithm, which can drive in the OpenAI Gym CarRacing-v0 environment. The environment consists of a randomly generated two-dimensional world of racetrack with grass and boundaries. The goal here is to reach the end of the track in as little time as possible.

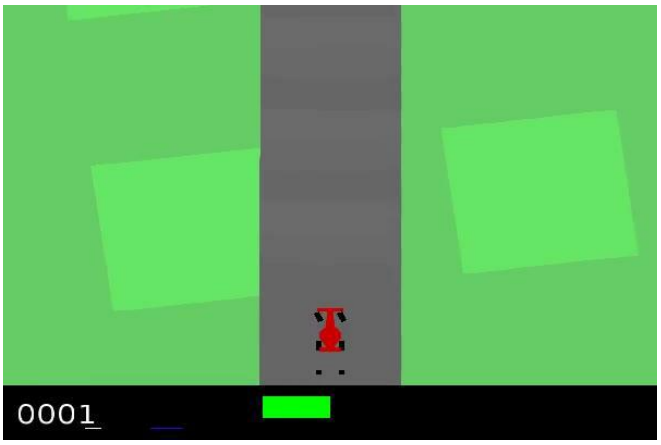


Figure 1 : OpenAI Gym CarRacing-v0 Environment

II. METHODOLOGY

Papers like Playing Atari with Deep Reinforcement Learning on NIPS in 2013 [2] and Human-level control through deep reinforcement learning on Nature in 2015 [4] introduced the concept of Deep Q Networks. DQN is inspired from Q-Learning, where Q-Learning is a model free reinforcement learning algorithm. The main task of a Q-Learning is to learn a policy, which guides an agent to take the best action under any circumstances. Model of the environment isn't required. Thus, the connotation of "model-free". Q-Learning finds an optimal action-selection policy, it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. Before learning begins, Q-table is initialized, Q-Table is just a simple lookup table where we calculate the maximum expected future rewards for action at each state. Basically, the agents will be guided to the best action at each state by the table.

To solve a real-world problem, Q-table is not a feasible solution, owing to continuous state and action spaces. Moreover, a Q-table is environment-specific and not generalized. Therefore, there is a requirement for a model which can map the state information provided as input to Q-values of the possible set of actions.

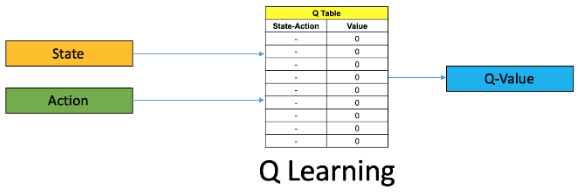


Figure 2 : Q-Learning

This is where a neural network comes to play the role of a function approximator, which can take state information input in the form of a vector, and learn to map them to Q-values for all possible actions.

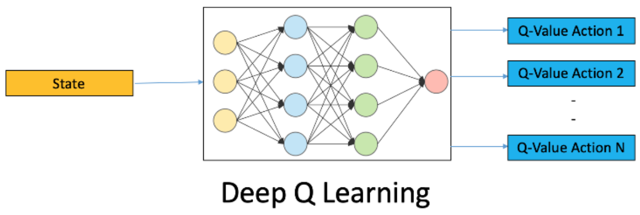


Figure 3 : Deep Q-Learning

Well, all the environments like CarRacing-v0 have one thing in common and that is, all are made of pixels. If the pixels can be provided to a model that can be mapped to actions then it can be generalized across all games [5]. DeepMind's implementation of convolutional neural networks had game image frames, where the inputs and the outputs were the Q-values for each possible action in that environment. Therefore, for our environment or any other gaming environment, a deep Q-network (DQN) consists of consecutive frames as the input to capture the motion and outputs Q-values for all possible actions in the game. Since a deep neural network is being used as a function approximator of the Q-function, this process is called deep-Q learning.

Originally, the observations obtained from the environment were colored RGB images with a black bar at the bottom to display score number. Since these observation images are the training inputs for the neural network, they were pre-processed to remove unwanted information and improve the training results.

The three channel image was converted to a single channel grayscale.

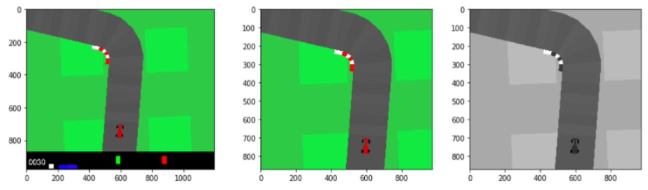
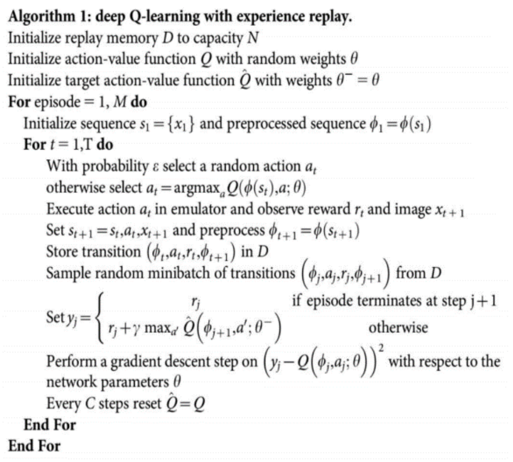


Figure 4 : Image Transformations

The essence of deep Q-learning is the estimation of Q∗ (s, a) using the neural network parametrized by a vector θ.

In RL, the policy or value functions play a vital role in sampling actions. However, this changes frequently as we know better what to explore. As we play out the game, we know better about the ground truth values of states and actions. So our target outputs are changing also. Now, we try to learn a mapping *f* for a constantly changing input and output. Luckily, both input and output can converge. So if in both input and output, we slow down the changes enough, we may have a chance to model *f* while allowing it to evolve.

Experience replay: For instance, last million transitions are put (or video frames) into a buffer and sample a mini-batch of samples of size 32 from this buffer to train the deep network. An input dataset is formed which is stable enough for training. As we randomly sample from the replay buffer, the data is more independent of one another. RL training is sensitive to optimization methods. Changes in input during the training cannot be handled as the Simple learning rate schedule is not dynamic enough. Many RL training uses RMSProp or Adam optimizer. This DQN is trained with Adam.



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t$ = 1,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

Figure 5 : DQN Pseudocode

where ϕ preprocess lasts 4 image frames to represent the state. To capture motion, we use four frames to represent a state. At training iteration i we write the network parameters as θi . The loss at this step is given by the temporal difference error.

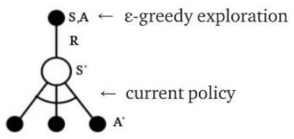
$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{s'}\left[\left(R(s,a) + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)^2\right].$$

Differentiating with respect to θi , we find that loss is minimized when the Bellman equation is satisfied, i.e.

$$Q(s,a;\theta_i) = \mathbb{E}_{s'}\left[R(s,a) + \gamma \max_{a'} Q(s',a';\theta_{i-1})\right]$$

This in conjunction with the fact that neural networks are universal function approximators implies that, given sufficient training data, a DQN will learn the optimal values of Q.

DQN uses ϵ-greedy to select the first action.



In our implementation of DQN we use a simple fully connected architecture where the first layer has 512 nodes. The first layer was fully connected to the second layer which consisted of 11 nodes . We used the ReLU and Linear activation function. We used an exploration rate of γ = 0.1 and a learning rate of λ = 0.01.

III. MODEL EVALUATION

We evaluated our model using conventional CPUs and training 200 episodes took about 12 hours.

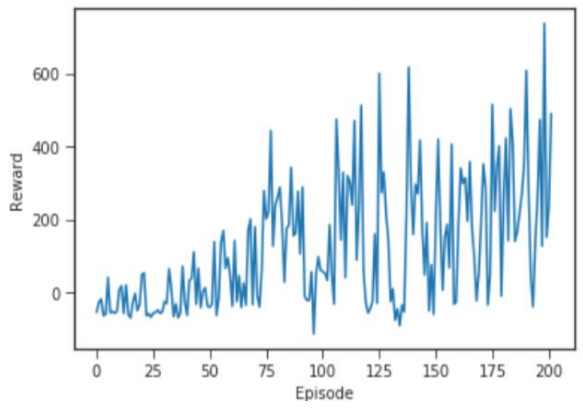


Figure 6 : Episode and Reward Graph - I

It seemed that the initial training didn't actually train at all. 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.

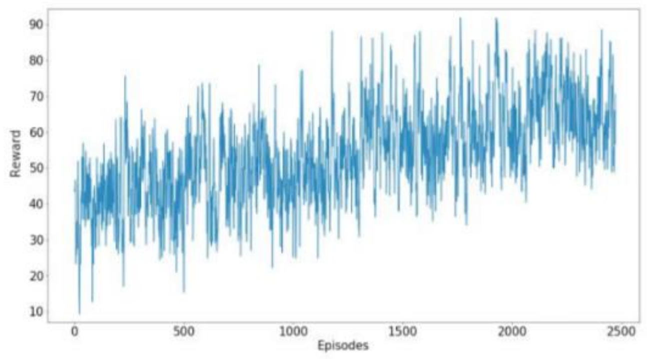


Figure 7 : Episode and Rewards Graph - II

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. The performance of our models are not appreciable. However, it must be recognized that reinforcement learners require many epochs to reach a decent solution, and we believe that we were an order of magnitude off in the number of necessary simulations that we ran on each of our models. What we are happy about is the clear upward trend in our rewards as the epochs rolled by. Knowing this, we are confident that our best model would have reached a reasonable solution given ample time and computational power.

IV. CONCLUSION

None of us had experience in implementing reinforcement learning algorithms prior to this project, and unfortunately much of our research time was spent trying to make up for this. Of course we would have preferred to use this time to improve our methodology to achieve more convincing results. Though the results didn't live up to our expectations, we learned a lot in the process. Our original proposal for this project was to train the model using multiple RL algorithms like DDPG, DDQN and PPO. It wasn't until the status update when we found out our problem was much harder than we first expected. This forced us to do a lot of our own research on DQNs and be able to somewhat successfully.

V. REFERENCES

[1] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou,

Daan Wierstra, Martin Riedmiller. arXiv:1312.5602. https://arxiv.org/abs/1312.5602

[3] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[4] Shashank Kotyan, Danilo Vasconcellos Vargas, Venkanna U. Self Training Autonomous Driving https://arxiv.org/abs/1904.12738

[5] Wal, Douwe van der and Wenling Shang. “Advantage Actor-Critic Methods for CarRacing.” (2018). https://esc.fnwi.uva.nl/thesis/centraal/files/f285129090.pdf

[6] Simon Kardell, Mattias Kuosku, “Autonomous vehicle control via deep reinforcement learning”, 2017

[7] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, “Alex Bewley, Amar Shah “Learning to Drive in a Day”, September 2018

[8] Mayank Bansal, Alex Krizhevsky, Abhijit Ogale, “ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst”, December 2018

[9] Nihal Altunas, Erkan Imal, Nahit Emanet, Ceyda Nur Ozturk, “Reinforcement learning-based mobile robot navigation”, March 2016.