Model for Car Racing in OpenAI Gym using Reinforcement Learning

Harsh Agrawal
Department of Computer Science)
North Carolina State University
Raleigh, USA
hagrawa@ncsu.edu

Isha Pandya
Department of Computer Science)
North Carolina State University
Raleigh, USA
ipandya@ncsu.edu

Sanjana Kacholia
Department of Computer Science)
North Carolina State University
Raleigh, USA
skachol@ncsu.edu

I. Introduction

Reinforcement Learning(RL) is a type of learning technique that enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and experiences. Deep Q-Networks can learn successful policies directly from high dimensional sensory input. Our Idea was to use this algorithm to train an agent who can successfully complete laps in 2D Car Racing environment. We want the agent to learn the best course of actions to complete the laps (maximize the reward) in the minimum amount of time possible. This can help us understand how these techniques can be applied to real-world problems such as autonomous cars.

II. Methodology

A. Environment

OpenAI's Gym is a toolkit for developing and comparing reinforcement learning algorithms. We are using CarRacing-v0 environment of the gym which is a top down car racing game. A track is made up a random number of tiles and the track is randomly generated for every run of the game. The goal of the car is to complete the lap, i.e visit every tile in the track and gain maximum reward. Reward is the points the car gains when it visits a tile successfully. Some of the components of the enviroment are explained in detail below:

1) State: At any point the state of the game can be extracted from the environment in the form of a 96 x 96 RGB Image (Fig 1). There are some indicators at the bottom of the image such as true speed, four ABS Sensors, steering wheel position and gyroscope.

2) Reward Function: Reward is calculated on the basis of how many tiles the car has visited since the start of the current episode.

$$R = 1000/N - 0.1F$$

N = Number of tiles visited
F = Number of frames rendered
Additional modifications:

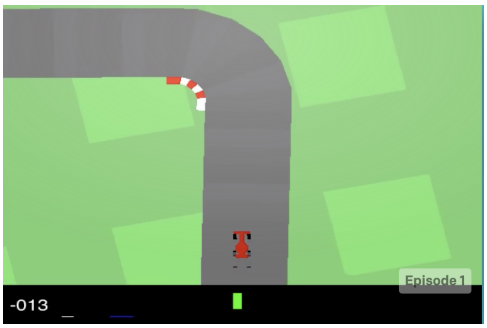

Fig. 1. Frame from CarRacing-v0

- Penalty of -20 points if the car gets 'n' consecutive rewards, where 'n' is decided based on the hyper-parameters in the game
- Bonus of 100 points if the car gains a total reward of 500 or more
- Penalty of -0.01 for each green pixel on the screen

3) Controls: There are 3 possible action present in the environment.:

- Steer: [left(-1), right(1)]
- Accelerate
- Brake

These values are given to the environment in the form of an array of length 3, where the elements denote the actions in respective order of steering, acceleration and braking. The combination of all the values specified above gives us a total of 12 possible actions that can be taken.

B. Network Architecture

We are using a Simple CNN network with 2 convolutional layer and 2 full-connected dense layers. (Fig 2)

III. Model Training and Hyperparameter selection

A. Algorithm

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model of the environment, and it can handle

TABLE I
NETWORK LAYERS

Layer(type	*Output Shape*	*Param #*
Conv2D	(8, 90, 90)	1184
BatchNormalization	(8, 90, 90)	360
MaxPooling2D	(8, 45, 45)	0
Conv2D	(6, 43, 16)	6496
BatchNormalization	(6, 43, 16)	64
MaxPooling2D	(6, 21, 8)	0
Flatten	(1008)	0
Dense	256	258304
Dense	num_actions(12)	3084

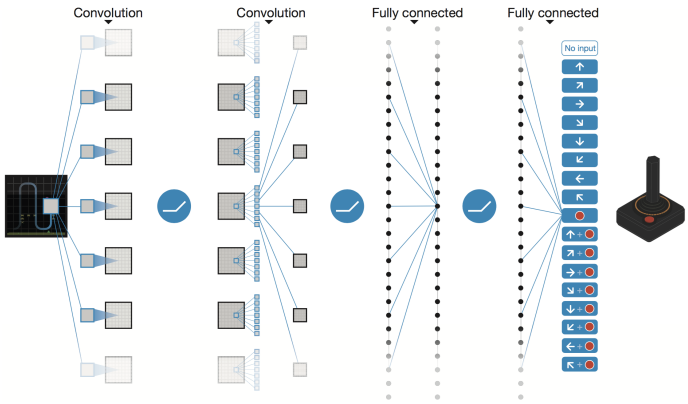

Fig. 2. Network Structure

problems with stochastic transitions and rewards without require adaptations. Model is trained using Deep Q Learning Algorithm with Experience replay as mentioned in the the paper 'Playing Atari with Deep Reinforcement Learning'. Our aim is to train a policy that tries to maximize the reward gained by the agent. The main idea behind Q-learning is that we have a function,

$$Q^* : State \times Action \rightarrow \mathbb{R}$$

that will tell us the reward given a state and the corresponding action to be performed. Out policy will be to maximize the reward

$$\pi^*(s) = argmax\ Q^*(s, a)$$

However since the environment is unknown, we don't have access to Q^*. But since neural networks are universal function approximators, we can simply create one and train it to resemble Q^*.
For our training update rule, well use a fact that every Q function for some policy obeys the Bellman equation:

$$Q^\pi(s, a) = r + \gamma Q^\pi(s', \pi(s'))$$

The difference between the two sides of the equality is known as the Q-value δ:

$$\delta = Q(s, a) - (r + \gamma maxQ(s', a))$$

To minimize the error we use Mean-squared error. We calculate this over a batch B of transitions sampled from the replay memory.
The algorithm makes use of the following component explained briefly:

- Replay Memory: A queue of length 'N' used to store the state changes in the environment and the corresponding rewards. Sample mini-batches are extracted from this memory and used to train the network. In the beginning, the replay buffer is filled with random experiences.
- Policy Network: The network that learns the policy
- Target Network: We use a separate network to estimate the target values. This improves the stability of the model. Every 'network_update_frequency' steps, this network is updated with the parameter values of the policy network.

Figure 3 shows a single iteration of the algorithm.

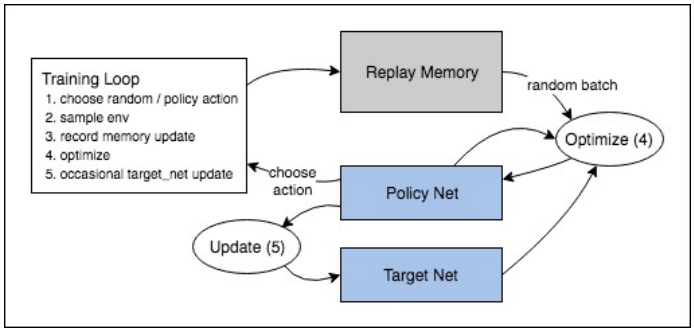

Fig. 3. Deep Q-learning Algorithm

B. Training

The state of the environment is represented by a stack of 3 frames from the game. Each frame is pre-processed by converting them to gray-scale before feeding it to the replay memory.

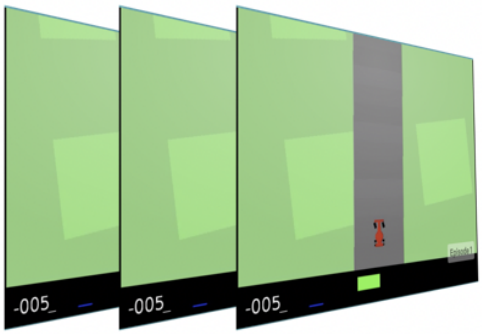

Fig. 4. Representation of State

A transition is a tuple of (State, Action, Reward, Next State, Done) where:

- State: Current state of the environment

- Action: Action taken by the agent
- Reward: Reward gained by the agent
- Next State: The change in state when the action is performed
- Done: Boolean values representing if the current episode of the game is over

Since to train the model we continually update the transitions stored in the replay memory, we have to perform action on the environment to generate a transition. A transition is generated by the following steps:

- Choosing a random action or an action predicted by the network. This is determined using epsilon value, which starts at a value of one and decays as the number of game steps increases, forcing the network to choose an action and learn from its experiences.
- Same action is performed for 'frame_skip' times. We found that skipping 3 frames from when a certain action is performed, to record the next state, yields the best result.
- Reward gain/loss is calculated and recorded for each of these steps.
- Check if the episode ended by using the 'Done' flag. An episode ends under 2 condtions
  - If the car has visited all of the tiles in the track
  - If the car has gained negative record consecutively for 'max_neg_rewards' steps.
- Next state is extracted by recording the 3 new frames rendered by the game.
- Tuple is created and pushed to the memory.

Model is trained using batches of 'N' transitions extracted from the replay memory and the loss of Q-value is minimized.

C. Hyperparameters

Parameters used for the training of the network and their significance:

TABLE II
HYPER PARAMETERS

Parameter	*Value*
Experience Capacity	40000
Batch Size	64
Num. of Frames in stack	3
Gamma	0.95
Tau	0
Initial Epsilon	1
Minimum Epsilon	0.1
Epsilon Decay Steps	40000
Learning Rate	0.0004
Train Frequency	4
Frame Skip	3
Network update freq	1000
Maximum negative rewards	12

IV. EVALUATION

Our model was able to learn how to cover the whole track by playing around 2 hundred thousand steps which is around 4000 episodes of the game. We plotted the curve for reward gained for each episode of the game played by the agent.

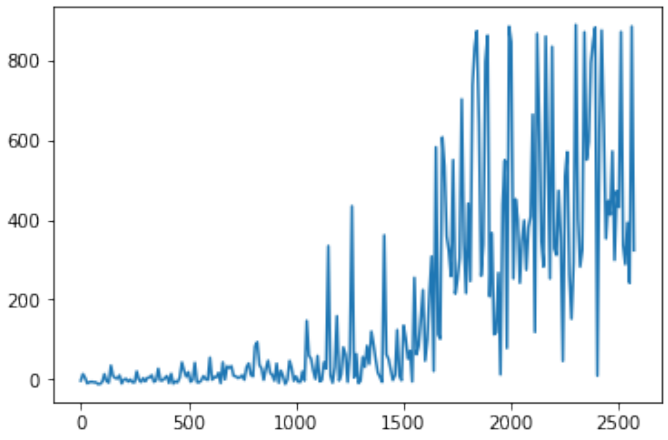

Fig. 5. Reward for each episode

Smoothing applied on the same curve for every 10 episodes show us clearly how the agent starts to learn how to maximize reward around episode 150

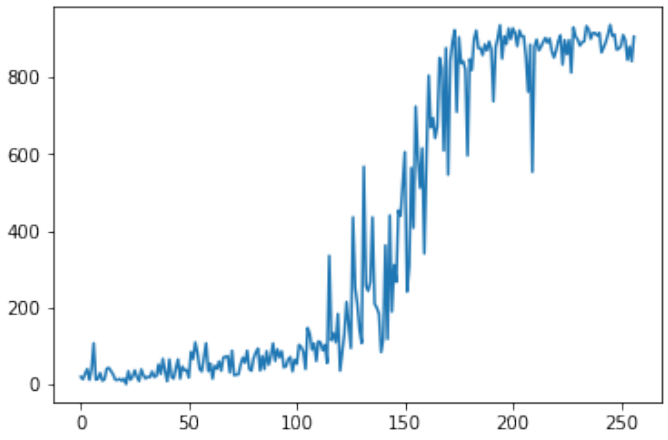

Fig. 6. Avg Reward for every 10 episode

There is no Training/Validation loss curve since there is no training/validation data involved in reinforcement learning technique. Learning rate was constant over all the epochs and Q-value curve follows the same curve as reward. From this we can determine that the model was able to successfully accomplish its goal in learning the environment and maximizing the reward.

V. CONCLUSION

The agent performed relatively well and played the game better than what a human can possibly learn in the same amount of time. It can be optimized further by changing the network structure and tuning the hyper-parameters hence reducing the cost of training. Apart from that we also learned that the algorithm is pretty generic and the code can

be generalized even more such that it be applied to any environment.

We would like to compare this algorithm with more complex algorithms such as Double Deep Q Network, Duelling Q Networks and A3C algorithm.

In conclusion, this was an interesting project and we learned a great deal about various reinforcement learning techniques and how they can be used in various other fields which can solve some common problem that normal ML techniques suffer such as lack of data and dynamic environment.

VI. Code

Code can be found at
https://github.com/hrshagrwl/autonomous-car-racing

References

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville, "Deep Learning"

[2] Playing Atari with Deep Reinforement Learning - https://arxiv.org/abs/1312.5602

[3] Human-level control through Deep Reinforcement Learning - http://files.davidqiu.com/research/nature14236.pdf

[4] Duelling network architectures for Deep Reinforcement Learning - https://arxiv.org/abs/1511.06581

[5] Implementing Deep Reinforcement Learning models with tensorflow + openAI Gym - https://lilianweng.github.io/lil-log/2018/05/05/implementing-deep-reinforcement-learning-models.htmldeep-q-network