Robot arm manipulation using Deep Reinforcement Learning

Trim Bresilla

Introduction

Reinforcement Learning is a Machine Learning subfield where an agent learns by interacting with its environment, observing the results of these interactions and receiving a reward accordingly be that positive or negative. This way of learning mimics the fundamental way in which humans and animals alike learn. It is one of the most promising machine learning paradigms for the future development of autonomous robots. It allows a robot to learn from trial-and error interactions with its environment. By observing the results of its actions, a robot can determine the optimal sequence of actions to take in order to reach some goal. Reinforcement learning offers to robotics a framework and set of tools for the design of sophisticated and hard-to-engineer behaviors [Kober et al]. The goal of reinforcement learning is to find a mapping from states to actions, called policy, that picks actions in given states and maximizing the cumulative expected reward.

Recently the combination of deep learning and reinforcement learning was proposed. In reinforcement learning, convolutional networks can be used to recognize an agent's state. This combination allows a learning agent to control a system based only on visual inputs, using a deep neural network to extract relevant features from the images. In reinforcement learning, given an image that represents a state, a convolutional network can rank the actions possible to perform in that state. At the beginning of reinforcement learning, the neural network coefficients may be initialized stochastically, or randomly. Using feedback from the environment, the neural net can use the difference between its expected reward and the ground-truth reward to adjust its weights and improve its interpretation of state-action pairs.

The task of our project was to control a three joint arm simulated in Gazebo environment and controlled by a Deep Q Network. To increase the communication speed and to reach more down to hardware code, an C++ API was used.

Reward Function

Reward is a function (or constant) that is given to the agent upon performing an action. In our case we have defined different kind of rewards (in form of gradient) so the agent gets faster to the goal and learns the best performance.

Episode end reward

At the beginning, a constant reward was given to the robot when an episode ends and the robot does not either reach the goal or collide with another object (ground in our case). This proved to be okay, but it would take the robot quiet some time to know in which direction to move. In order to decrease this time, an alternative reward function was created. Everytime the episode ends, the distance form the griper and the goal was calculated, then multiplied by a constant to generate a reward. This proven to be very efficient as the robot would reach the very close to the goal in less than 6 episodes. The reward would reach -9000 if the arm goes completely in the other direction and the episode ends.

Interim reward

The interim reward was given based on the gripper distance to the object. This reward was much smaller than the episode end reward, but this is issued for every frame until the episode ends. It was observed that this is the most crucial reward function of all, as it serves as direct guide to where the gripper is and how far is from the goal. This was calculated as a smoothed moving average:

```
avgGoalDelta = (avgGoalDelta * ALPHA) + (distDelta * (1.0f - ALPHA));
```

Collision reward

There are two collision rewards. When the arm hits the ground or when the gripper hits the goal.

1. Ground collision is detected through the Gazebo's physics engine and served to us as API. The reward function was designed so that when the arm hits the ground it immediately calculates the distance from the goal and issued a negative reward based on how much far was from the goal. This proved to be not as efficient as initially thought.
2. Goal reached collision was the only reward from al the above described that was given to the agent as positive. When the gripper (or the arm for task 1) reaches the goal a constant and very large reward reward is issued. Later on, other functions were created so the reward would not be only constant but linear or exonential: depending how many strikes without loosing it gets, or based on time it passed, the sooner it starts reaching the goal the higher the reward is and later very slowly decreasing. All those proven to be useful when the robot needs more than few hundreds or thousand episodes.

Joint control

To us was given two ways to control the arm joints of the robot. One by controlling the velocity and the other by controlling the position. Because of the higher performance, and faster accuracy reach, control of the position was used.

Hyperparameters

DQN algorithms have been extensively used for various robotics artificial intelligence related problems in recent years. However, their adoption is severely hampered by the many hyperparameter choices one must make. When working with neural networks and machine learning pipelines, there are dozens of free configuration parameters (hyperparameters) need to configure before fitting a model. The choice of hyperparameters can make the difference between poor and superior predictive performance. There is many research being done on trying to make this process more streamlined and easier. However in our case, unfortunately we tweaked parameters by trial-and-error and past experience and intuition (which sometimes is your best friend, and sometimes your worst enemy). Here are the parameters changed:

parameter:	*old value*	*new value*
1. Input image height and width	512x512	64x64
2. Optimiser	Adam	RMSprop
3. Replay memory	10000	25000
4. LSTM	false	true (256)
5. Learning rate	0.01	0.1
6. Batch size	512	256
7. Randomize	false	true

Each parameter affects the performance in different ways. Eg. Learning rate, switching from 0.01 to 0.1 made the arm learn faster reach the goal, from 200+ episodes decreased to 60 episodes.

Results

The objective of the project was divided into two tasks:

1. Reach an accuracy of more or equal to 0.9 when **any part** of the arm reached the goal.

2. Reach an accuracy of more or equal to 0.8 when **gripper** of the arm reached the goal.

Both asks were reached with the parameters explained above, and in the code, an easy switch was coded to change the task on the fly.

TASK 1

After making the changes described on the reward section, it took very few iteration. In our case it took 10-15 iteration to start getting more wins than losses and by 50 iteration there was only win strikes.

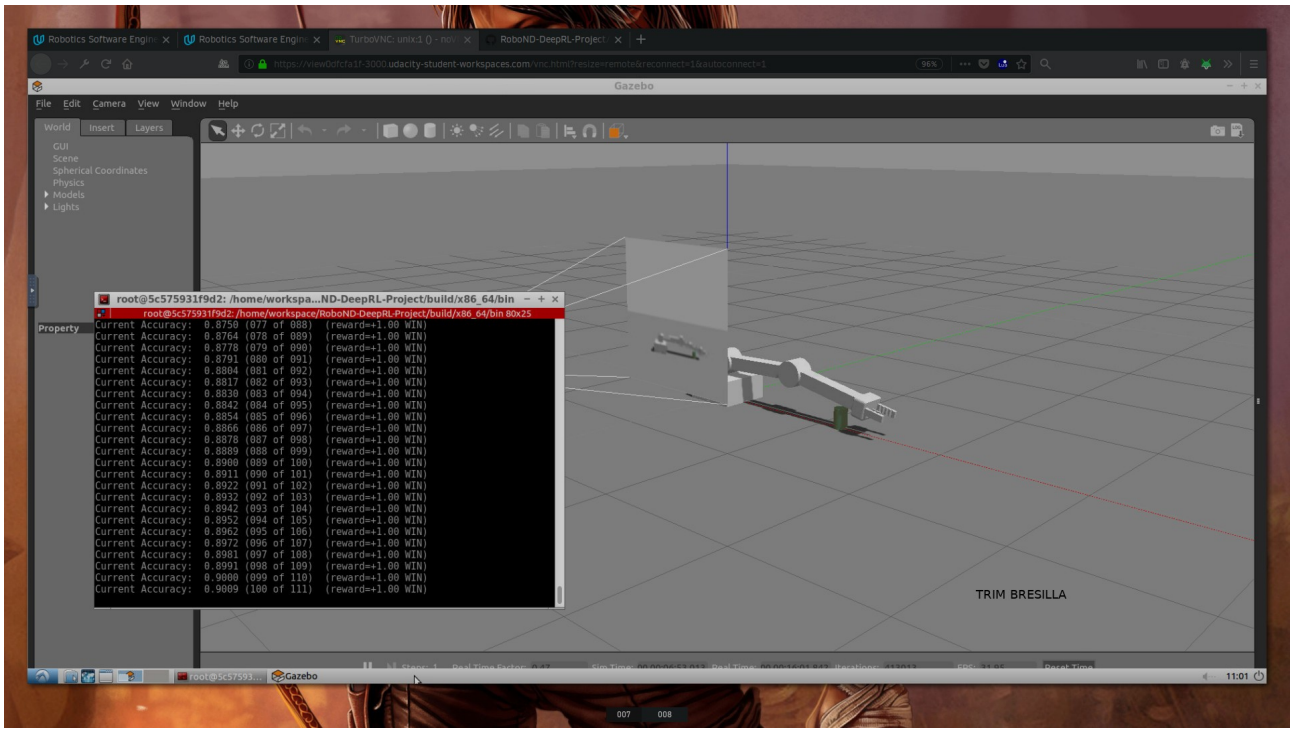

Fig 1. Task 1 goal reached with 0.9

TASK 2

Task 2 was more challenging. After modifications of reward functions and hyper parameters the accuracy was reached.

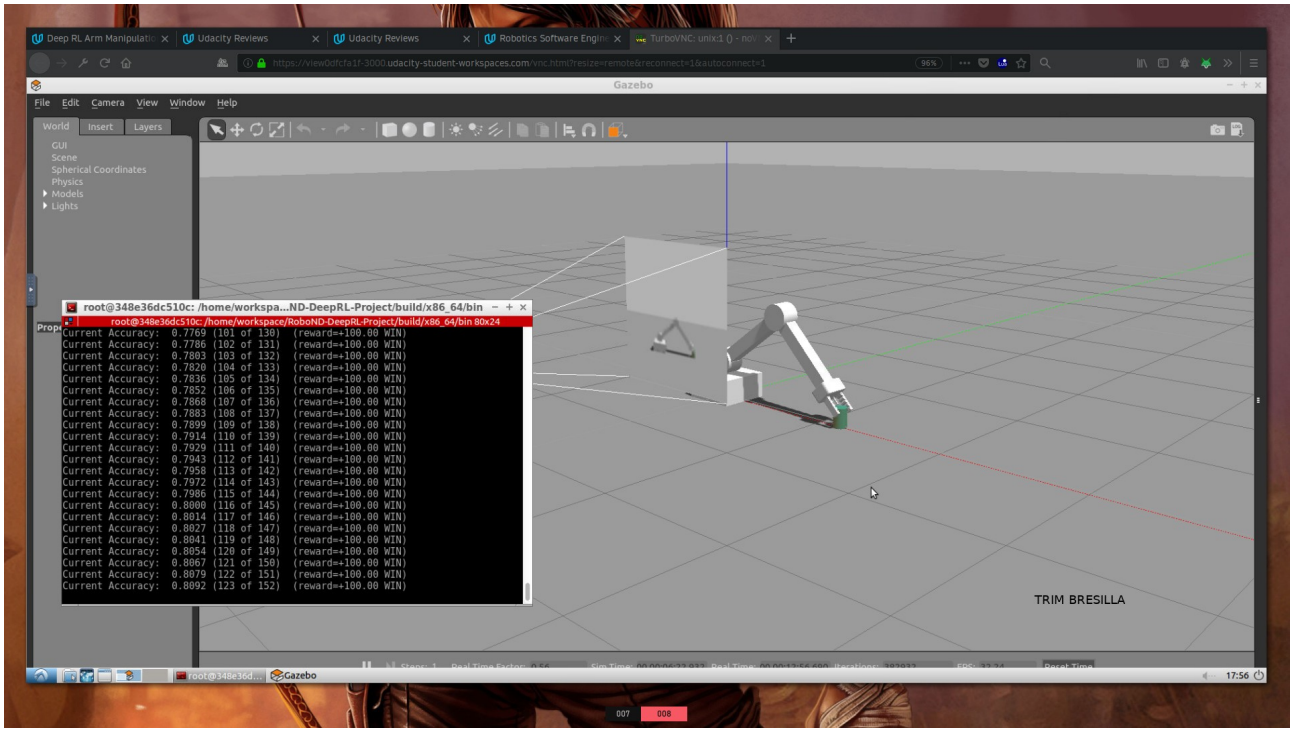

Fig 2. Task 2 goal reached with 0.8

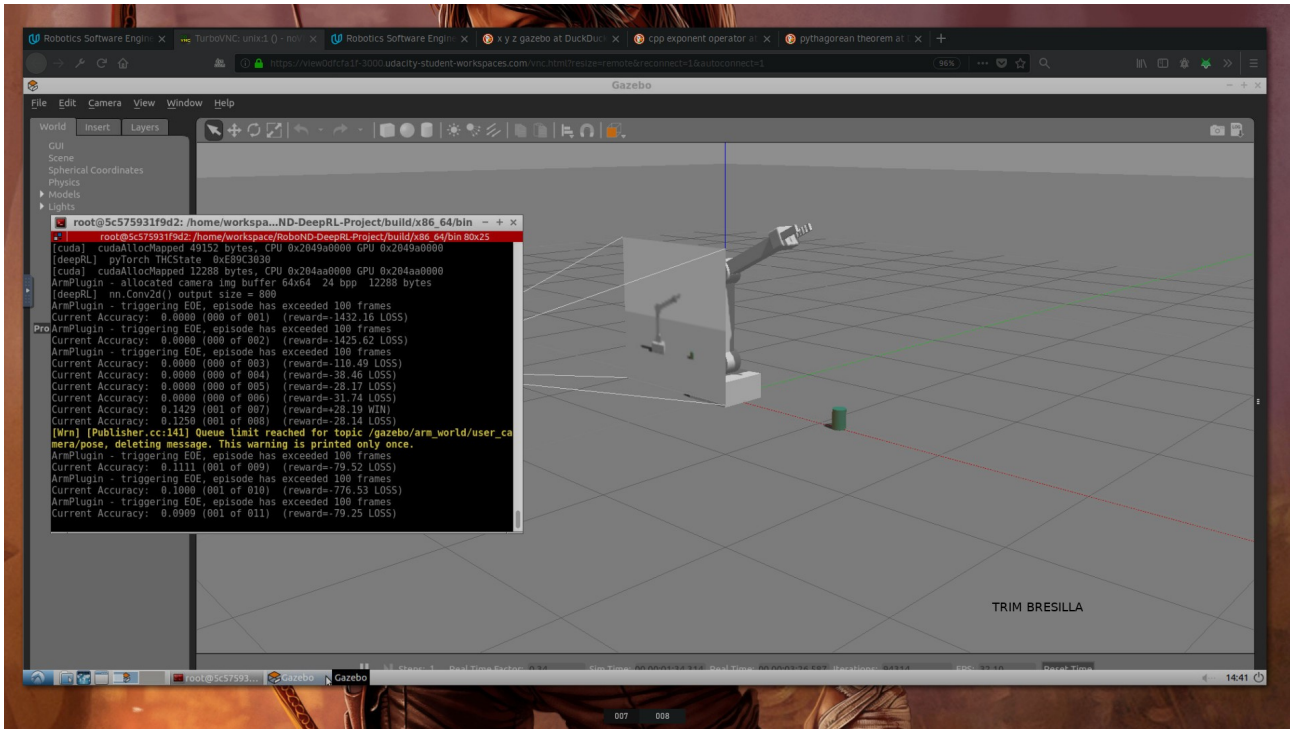

Fig 3. Negative reward function significant decrease.

There were different reward functions designed (as described above), and the one showed in Fig 3 was the most efficient. As seen from the picture the negative reward decreases from the rage of -3000 to -40 in just four iteration. This was after the episode end reward function based on distance was designed.

Future improvements

There are many improvements that can be implemented to increase the accuracy and efficiency. It was observed that sometimes (still for some unknown reasons, needs more debugging) the arm would take many iterations until it finds the goal (even after implementation of those different reward functions). When the arm hits the goal (randomly or by following the smoothed moving average) it was easier after to get on track and start learining.

Another improvement would be the gripper collision. Sometimes it was observed that the griper would reach the goal (the can) and would put in between the fingers, but because of the size of the object was small, grip would hit the ground when reaching the can and still be considered a loss, even though that would count for perfect reaching of goal.

Other improvements would be to make the gripper rotate and randomise the can position. In this was the gripper needs to adjust the rotation and learn to grip the object. This in fact would open a load of other complex problems to be considered.

About DQN and Hyperparameters, there are many improvements that can be done. Using some new tools like: hypersearch https://github.com/kevinzakka/hypersearch that would try to find automatically the best hyperparametrs would be the logical choice.