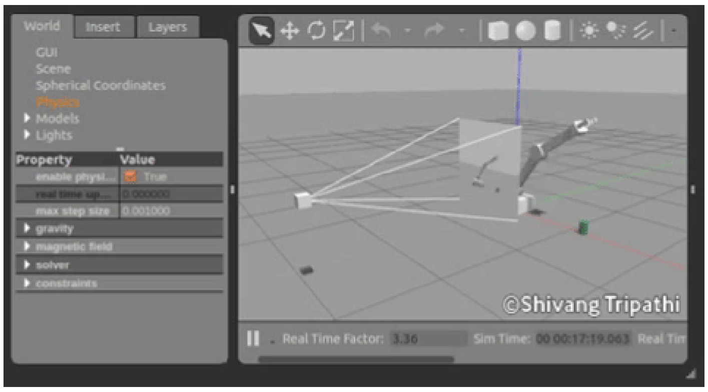
Report WriteUp

This is the writeup for the Deep RL project based on the Nvidia open source project "jetson-reinforcement" developed by Dustin Franklin. **The goal of the project is to create a DQN agent and define reward functions to teach a robotic arm to carry out two primary objectives:**

> 1. Have any part of the robot arm touch the object of interest, with at least a 90% accuracy.
> 2. Have only the gripper base of the robot arm touch the object, with at least a 80% accuracy.

Hyperparameters tuning has the most challenging and time taking part of the project.



Property	Value
enable physi...	True
real time up...	0.000000
max step size	0.001000

Real Time Factor: 3.36 Sim Time: 00 00:17:19.063

Reward Functions

The design of the reward function is the single most important task to accomplish any objective using reinforcement learning. Spending time on designing a good reward function saves a lot of time in later training and tuning the hyperparameters. The agent always tries to optimize the cumulative reward which can sometimes lead to undesired unplanned behavior. This is also called the Cobra effect.

Rewards for the interim states

The distance from the gripper to the goal prop was measured using the `distGoal(gripper, prop)` method. Change of distance to goal was called `delta = distGoal - lastGoal`. As evident, negative `delta` implied the gripper was moving towards the goal pose.

Exponential moving average of the delta `avgGoalDelta` was used in the reward function.

```
alpha = 0.7;
avgGoalDelta = avgGoalDelta * (1 - alpha) + delta * alpha;
```

Higher `alpha` value meant lesser weightage to the previous values of distribution compared to the later values.

```
reward = -C1 * avgGoalDelta
// -C1 is a negative constant
```

Its important to note that this is a positive reward function, i.e., as the gripper moves towards the goal prop, the rewards will be positive.

The important point to note in the case of positive rewards is if the agent can accumulate excessive positive reward without termination *by simply staying in its place*. This does not happen because if the delta stays **0** (position not changing), the rewards will (*smoothly*) become **0**. Thus, training with this reward function will not lead to the robot getting stuck to a position and still accumulate large rewards.

The faster the robotic gripper moves towards the prop, the more the rewards will be. This is because the value of the `delta` will be higher. Thus, this reward function incentivizes the fast robot gripper and prop collision.

Rewards for the terminal states

For the task 1, there were 2 terminal states:

1. Ground contact: The episode was terminated when the gripper touched the ground(*or was within 0.05 meters from the ground*). This was a case of dangerous outcome as the collision with ground will lead to wear and tear of the arm. This was given a penalty of `REWARD_LOSS = -0.10` .
2. Collision with prop: The episode terminated when any part of the arm touched the goal prop. This was an intended outcome and was, thus, given a reward of `REWARD_WIN = 0.20` .

For the task 2, there were 3 terminal states:

1. **Ground contact:** Similar to the previous case, this was given a penalty of `REWARD_LOSS = -0.10` .
2. **Non-gripper collision with prop:** When a non-gripper part of the arm collides with the goal prop, the agent is partly reaching the near the goal. Thus, this was given a reward of `REWARD_WIN/2 = 0.10` .
3. **Gripper Prop collision:** When the gripper touches the goal prop, this was exact intended action and the agent was rewarded with `REWARD_WIN = 0.20` .

Also, if any episode reached 100 frames, it was terminated with a penalty of `REWARD_LOSS = -0.10` .

Type of Robot Joint Control Used

Both velocity as well as the position control was experimented with. Position control performed well for both the task whereas the velocity control did not work well with the second task as it required more precise positioning of the gripper near the prop.

The advantage of the velocity control was that the arm had less chances of getting stuck at any one position because some **non-zero constant output** would almost always be given by the agent whereas in the case of position control, some **non-zero constant output** would mean that the arm will stay stuck in a particular position.

Joint position control was a good compromise because this was a direct way of controlling the arm. **The important point to note was that the agent was rewarded not on its current distance from the goal but on the difference of successive distances from the goal(*deltas*).** This reduced the chances of the robotic arm getting stuck because that would make the `reward -> 0` (smoothly) and agent always trying to maximize cumulative rewards would not get stuck in a single position.

The joint position control was used which performed well for both the tasks of the project.

Hyperparameters

The `INPUT_WIDTH` and `INPUT_HEIGHT` was kept equal to the size of the input image, i.e., `64 x 64` .

`Adam` was tried with various parameters but `RMSprop` turned out to be better and was used to achieve the required accuracy for both the tasks.

Learning rate was set high enough to learn fast enough to reduce the number of episodes required to achieve the required accuracy. With higher rates, the accuracy was not reached and with lower rates, the number of episodes needed increased by order of hundreds.

The degree of randomness though was set to `true` , was done with a low `EPS_END` of `0.005` to prevent the agent the agent from trying random trajectories every so often, taking more time and episodes to reach required accuracy.

Larger batch size leads to convergence to narrow goal cases. This means that the agent will work only for cases in which the prop is in the narrow neighborhood of the prop training region. The agent will fail to generalize for the wide range of goal positions that it may encounter. Thus, a smaller batch size was used to generalize for various distance goal instances.

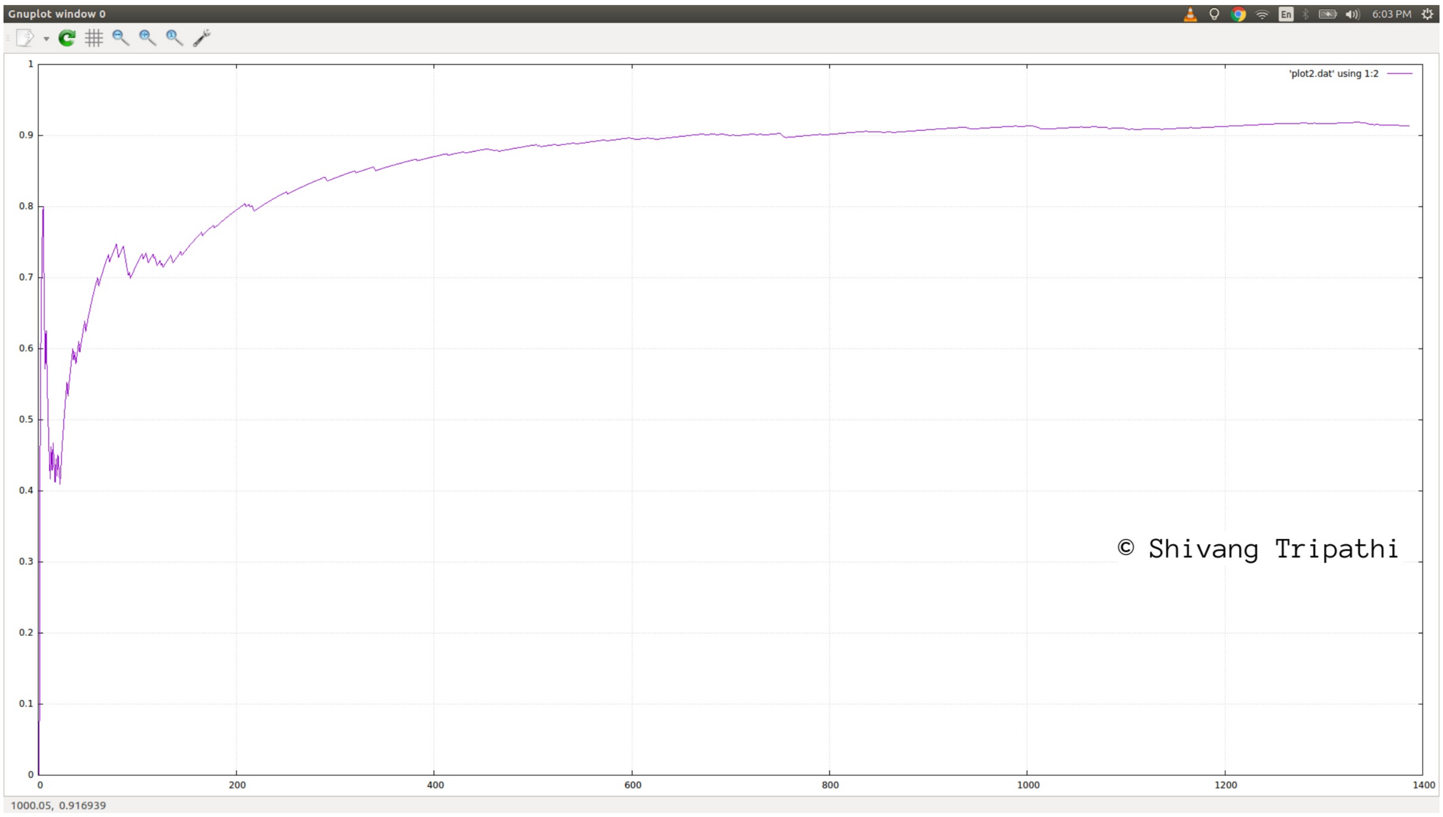
The LSTM size was chosen depending upon the length of sequence that will need to be learnt. Episodes terminated at 100 episode frames but the agent was able to come to a conclusive result(ground or prop collision) by the 30 - 40 episode frames. Also, making size a multiple of 32 may actually speed up the training by a small amount. Thus, a value of 64 was used for this task.

Results

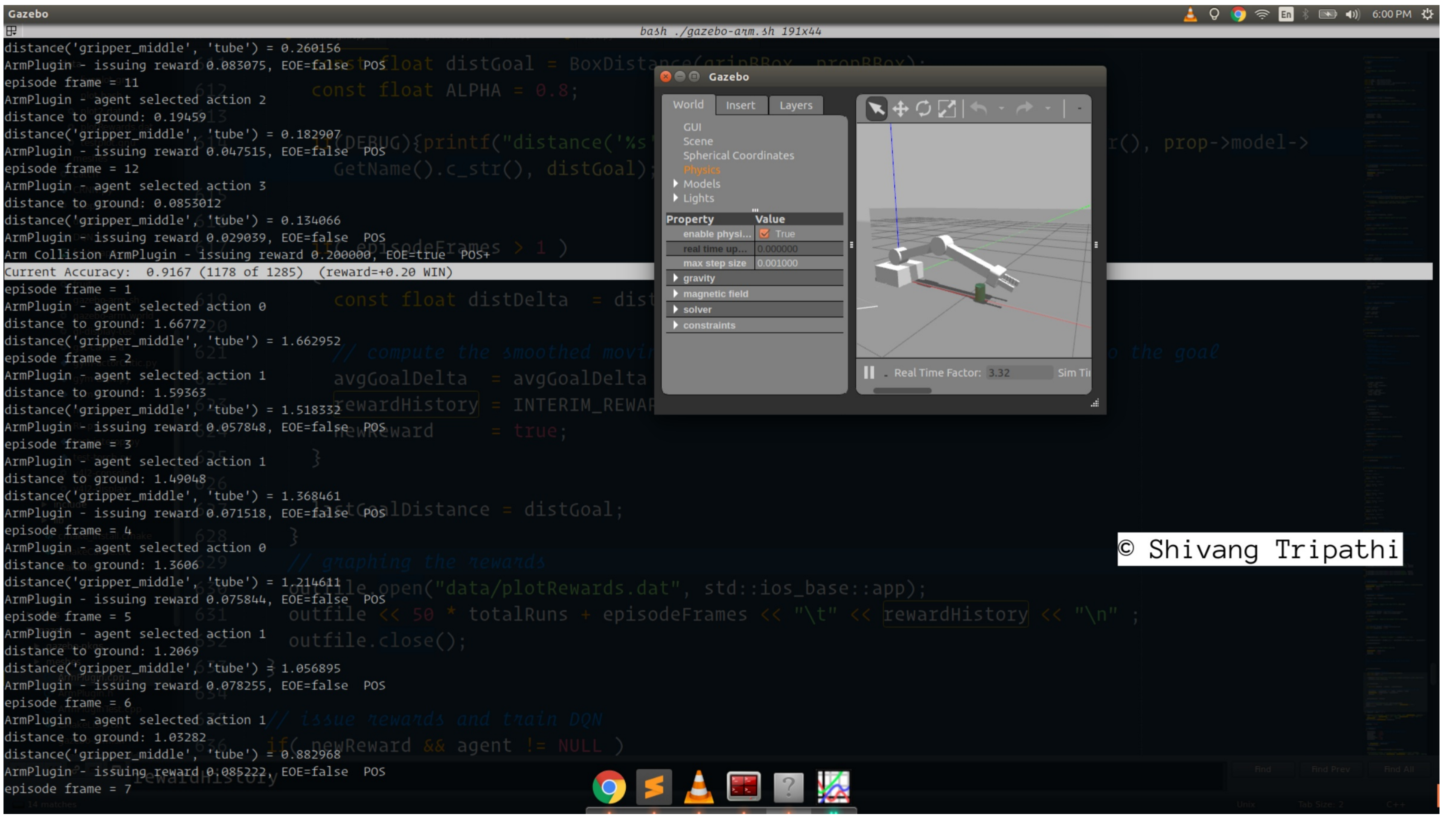
The first task to achieve was:

> Getting the robotic arm to touch the goal prop with more than 90% accuracy.

The watermarked screenshot of the **task 1 accuracy graph**:



It took close to `700 iterations` to reach the `90% accuracy` mark. It reached to `91.67%` by the `1285th iteration` .

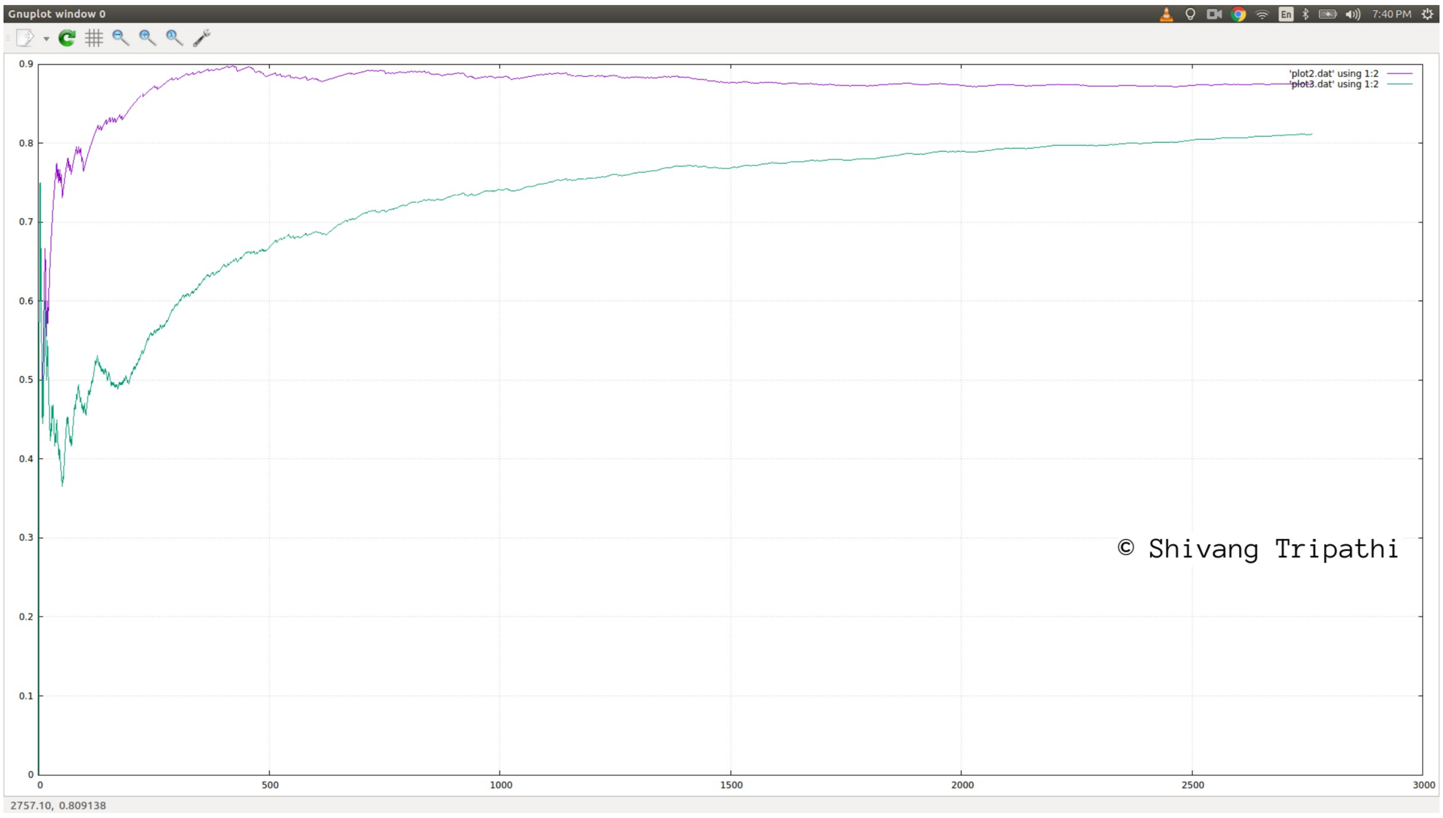


```
distance('gripper_middle', 'tube') = 0.260156
ArmPlugin - issuing reward 0.083075, EOE=false  POS
episode frame = 11
ArmPlugin - agent selected action 2
distance to ground: 0.19459
distance('gripper_middle', 'tube') = 0.182907
ArmPlugin - issuing reward 0.047515, EOE=false  POS
episode frame = 12
ArmPlugin - agent selected action 3
distance to ground: 0.0853012
distance('gripper_middle', 'tube') = 0.134066
ArmPlugin - issuing reward 0.029039, EOE=false  POS
Arm Collision ArmPlugin - issuing reward 0.200000, EOE=true  POS+
Current Accuracy:  0.9167 (1178 of 1285)  (reward=+0.20 WIN)
episode frame = 1
ArmPlugin - agent selected action 0
distance to ground: 1.66772
distance('gripper_middle', 'tube') = 1.662952
episode frame = 2
ArmPlugin - agent selected action 1
distance to ground: 1.59363
distance('gripper_middle', 'tube') = 1.518332
ArmPlugin - issuing reward 0.057848, EOE=false  POS
episode frame = 3
ArmPlugin - agent selected action 1
distance to ground: 1.49048
distance('gripper_middle', 'tube') = 1.368461
ArmPlugin - issuing reward 0.071518, EOE=false  POS
episode frame = 4
ArmPlugin - agent selected action 0
distance to ground: 1.3606
distance('gripper_middle', 'tube') = 1.214611
ArmPlugin - issuing reward 0.075844, EOE=false  POS
episode frame = 5
ArmPlugin - agent selected action 1
distance to ground: 1.2069
distance('gripper_middle', 'tube') = 1.056895
ArmPlugin - issuing reward 0.078255, EOE=false  POS
episode frame = 6
ArmPlugin - agent selected action 1
distance to ground: 1.03282
distance('gripper_middle', 'tube') = 0.882968
ArmPlugin - issuing reward 0.085222, EOE=false  POS
episode frame = 7
```

The task 2 was more refined:

> Getting only the robotic gripper to touch the goal prop with more than 80% accuracy.

This took more time and episodes but was a more meaningful task to accomplish for the agent. It touched the `80% accuracy` mark after around `2400 iterations` . The watermarked screenshot of the **task 2 accuracy graph**:



By the `2808th iteration`, the accuracy was at `81.23%`

Future Work

To improve the current results,

- Find a better reward function.

In the current reward function, until the very end of episode, the agent is blind as to where it is heading. At the very end, it may encounter a collision with a gripper(high reward) or a ground(high penalty). The reward function should also consider the current distance between the gripper and the ground.

- Bright coloring the gripper and the prop.

This will make it easier for the CNN in DQN algorithm to find the relation of gripper and prop position with the reward received.

DQN is a novel end-to-end reinforcement learning agent that can perform diverse range of tasks with superhuman level of mastery without human intervention. This is a major technical step in the quest of general AI. Paradoxically, the algorithms that power this machine learning are manually designed. Instead, if this learning could to automated, this could open up new opportunities. Learning to optimize and Learning to Learn by Thrun & Pratt, 2012) was a step in this direction which I would like to implement as they used RL to learn the optimizer.

Building from Source (Nvidia Jetson TX2)

Run the following commands from terminal to build the project from source:

```
$ sudo apt-get install cmake
$ git clone http://github.com/udacity/RoboND-DeepRL-Project
$ cd RoboND-DeepRL-Project
$ git submodule update --init
$ mkdir build
$ cd build
$ cmake ../
$ make
```

During the `cmake` step, Torch will be installed so it can take awhile. It will download packages and ask you for your `sudo` password during the install.