Project : Deep RL Arm Manipulation

Goutam Gupta

Abstract—A DQN agent is created to teach a robotic arm to touch an object of interest.The goal is to create an RL agent for the robotic arm that will learn to manipulate its joints to reach and touch the object placed in its vicinity. The simulation is performed in a Gazebo environment. Hyper parameters are tuned for velocity and position control of the robotic arm to get optimum performance.

Index Terms—Robot, IEEEtran, Udacity, Deep reinforcement learning, Deep Q learning

1 Introduction

THE goal of this project is create a DQN agent and define reward function to teach a robotic arm to carry out two primary objectives:

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

Both of these objectives, have associated tasks that shall be completed while working on this project. A robotic arm with a camera sensor, a collision detect sensor and an object is created in a Gazebo environment.There are three main components to this gazebo file, which define the environment:

- The robotic arm with a gripper attached to it.
- A camera sensor, to capture images to feed into the DQN.
- A cylindrical object or prop.

The robotic arm has three non-fixed joints and hence it has three degree of freedom.

2 Arm Plugin

The robotic arm model, found in the gazebo-arm.world file, calls upon a gazebo plugin called the **ArmPlugin**. This plugin is responsible for creating the DQN agent and training it to learn to touch the prop. The gazebo plugin shared object file, libgazeboArmPlugin.so, attached to the robot model in gazebo-arm.world, is responsible for integrating the simulation environment with the RL agent. The plugin is defined in the ArmPlugin.cpp file, also located in the gazebo folder.

The ArmPlugin.cpp file takes advantage of the C++ API. This plugin creates specific constructor and member functions for the class ArmPlugin defined in ArmPlugin.h. Some of the important methods, that will be required for the project, are discussed below:

- ArmPlugin::**Load()**. This function is responsible for creating and initializing nodes that subscribe to two specific topics - one for the camera, and one for the contact sensor for the object which detects collision.

  In gazebo, subscribing to a topic has the following structure:
  gazebo::transport::SubscriberPtr sub = node-Subscribe(”topic_name”,callback_function, class_instance);
  Where,
  - callback_function is the method thats called when a new message is received.
  - class_instance is the instance used when a new message is received.

  For each of the two subscribers, there is a callback function defined in the file and is discussed below.
- ArmPlugin::**onCameraMsg()**: This is the callback function for the camera subscriber. It takes the message from the camera topic, extracts the image, and saves it. This is then passed to the DQN.
- ArmPlugin::**onCollisionMsg()**: This is the callback function for the objects contact sensor. This function is used to test whether the contact sensor, called my_contact, defined for the object in gazebo-arm.world, observes a collision with another element/model or not.
  Furthermore, this callback function can also be used to define a reward function based on whether there has been a collision or not.
- ArmPlugin::**createAgent()**: The createAgent() class function serves the purpose wherein a DQN agent can be created and initialized. In ArmPlugin.cpp, the various parameters that are passed to the Create() function for the agent are defined at the top of the file.
- ArmPlugin::**updateAgent()**: For every frame that the camera receives, the agent needs to take an appropriate action.Since the DQN agent is discrete, the network selects one output for every frame. This output (action value) can then be mapped to a specific action - controlling the arm joints. The updateAgent() method, receives the action value from the DQN, and decides to take that action.
  There are two possible ways to control the arm joints:
  - Velocity Control.
  - Position Control.

For both of these types of control,the joint velocity or the joint position can be either decreased or increased, by a small delta value.Several different approaches to define on what basis the joint velocity or position changes can be experimented.For example, one such action can be based on whether the action value is odd or even.
ArmPlugin.cpp includes the variable VELOCITY_CONTROL, set to false by default, which can be used to define whether you want to control joint velocities or positions.

- ArmPlugin::**OnUpdate()** : This method is primarily utilized to issue rewards and train the DQN. It is called upon at every simulation iteration and can be used to update the robot joints, issue end of episode (EOE) rewards, or issue interim rewards based on the desired goal.
  At EOE, various parameters for the API and the plugin are reset, and the current accuracy of the agent performing the appropriate task is displayed on the terminal.

2.1 Tasks

We have to perform following tasks

- Subscribe to camera and collision topics. The nodes corresponding to each of the subscribers have already been defined and initialized. We will have to create the subscribers in the ArmPlugin::Load() function, based on the following specifications -
  For the camera node :
  - Node - CameraNode
  - Topic Name - /gazebo/arm_world/camera/link/camera/image
  - Callback Function - ArmPlugin::onCameraMsg (this should be a reference parameter)
  - Class Instance - refer to the same class, using the this pointer or keyword.

  For the contact/collision node -
  - Node - collisionNode
  - Topic Name - /gazebo/arm_world/tube/tube_link/my_contact
  - Callback Function - ArmPlugin::onCollisionMsg (this should be a reference parameter)
  - Class Instance - refer to the same class, using the this pointer or keyword.

- Create the DQN Agent : Refer to the API instructions to create the agent using the Create() function from the dqnAgent Class, in ArmPlugin::createAgent(). Pass the variable names defined at the top of the file to this function call.
  Add LSTM parameters when initializing DQN agent! agent = dqnAgent::Create(..., USE_LSTM, LSTM_SIZE, ALLOW_RANDOM, DEBUG_DQN);
- Velocity or position based control of arm joints :
  DQN output is mapped to a particular action, which, for this project, is the control of each joint for the robotic arm. In ArmPlugin::updateAgent(), there are two existing approaches to control the joint movements. We can either use one of these approaches ;
  - Velocity control :
    The current value for a joints velocity is stored in the array vel with the array lengths based on the number of degrees of freedom for the arm. If we choose to select this control strategy, modify the following line of code to assign a new value to the variable velocity based on the current joint velocity and the associated delta value, actionVelDelta
    float velocity = 0.0;
  - Position control :
    The current value for a joints position is stored in the array ref with the array lengths based on the number of degrees of freedom for the arm. If we choose to select this control strategy, modify the following line of code to assign a new value to the variable joint based on the current joint velocity and the associated delta value, actionJointDelta.
    float joint = 0.0;
    To find out which joints value to change (corresponding index to either of the arrays, ref or vel) we can define the index as a function of the variable action. This is helpful if we want to have a more complicated (more degrees of freedom) arm without having to define new set of conditions. The default number of actions, defined in DQN.py, is 3; and there are two outputs for every action - increase or decrease in the joint angles.
    The next set of tasks are based on creating and assigning reward functions based on the required goals. There are a few important variables in relation to rewards -
    - rewardHistory : Value of the previous reward, we can set this to either a positive or a negative value.
    - REWARD_WIN or REWARD_LOSS : The values for positive or negative rewards, respectively.
    - newReward : If a reward has been issued or not.
    - endEpisode : If the episode is over or not.

    For each of the following, we have to decide what rewards to use, and whether or not its a new reward and whether or not to trigger an end of episode.
- Reward for robot gripper hitting the ground:
  In Gazebos API, there is a function called GetBoundingBox() which returns the minimum and maximum values of a box that defines that particular object/model corresponding to the x, y, and z axes.
  For example, for the bounding box defined for the robots base, we will have the attributes box.max.x and box.min.x to obtain the minimum and maximum values of the box in reference to the x-axis.

Using the above, we can check if the gripper is hitting the ground or not, and assign an appropriate reward. The bounding box for the gripper, and a threshold value have already been defined for us in the ArmPlugin::OnUpdate() method.

- Issue an interim reward based on the distance to the object:
  In ArmPlugin.cpp a function called BoxDistance() calculates the distance between two bounding boxes. Using this function, calculate the distance between the arm and the object. Then, use this distance to calculate an appropriate reward as well.
- Issue a reward based on collision between the arm and the object:
  In the callback function onCollisionMsg, we can check for certain collisions. Specifically, we will define a check condition to compare if particular links of the arm with their defined collision elements are colliding with the COLLISION_ITEM or not. The function already contains some code that can help us with this task.We can use below recommend reward which is a smoothed moving average of the delta of the distance to the goal.
  average_delta = (average_delta * alpha) + (dist * (1 - alpha));
- Tuning the hyperparameters :
  The list of hyperparameters that we can tune is provided in ArmPlugin.cpp file, at the top.we will tune these hyperparameters to obtain the required accuracy.We might also have to revisit our reward functions and tune or modify them as needed.
- Issue a reward based on collision between the arms gripper base and the object:
  For this task of the project, modify the collision check defined in task 6 above, to check for collision between the gripper base and the object. And then, assign the appropriate reward. Run your project again, and observe how the agent performs. Tune the rewards or the hyperparameters to obtain the required accuracy, if required.

3 Reward Functions

The reward function used for Intermediate reward by both the task is selected based on the recommendation in the lesson, which is a moving average of the delta of the distance to the goal.
averageGoal_delta = (averageGoal_delta * alpha) + (distDelta * (1 - alpha)); alpha was changed from 0.1 to 0.7 and it was observed that alpha value of 0.5 works best for both the task. Moving average output of delta distance to goal is multiplied by reward multiplier and then updated in reward history. This is done to converge towards goal rapidly if the distance of the arm decreases from the object. Similarly same multiplier concept helps in learning rapidly when the distance from goal increases. Various values of REWARD_MULTIPLIER were tried and value of 4 works out to be best for both the tasks.
REWARD_WIN and REWARD_LOSS parameters that works out best for both the task is +20 and -20 respectively.

All the parameters associated with reward function are summarised in the below table.

TABLE 1
Reward Parameters

Reward Parameters	Values
REWARD_WIN	20.0
REWARD_LOSS	-20.0
REWARD_MULTIPLIER	4.0
alpha	0.5

3.1 Win Reward

Reward is awarded whenever the robot arm achieves following conditions:

- The robot arm makes a collision with the object
- The robot arm as well as the gripper base makes a collision with the object.
- Whenever the robot arm moves towards the object. This is described in the section Interim Reward.

Whenever above two conditions are met, collision flag is set to true and the reward is awarded.

3.2 Loss Reward

Apart from awarding Interim reward, a reward loss is also awarded for task 2. As the goal in task 2 is that the gripper base should touch the object, a reward loss of -20 is awarded whenever the arm touches the object but the gripper is not able to reach to the object. This is done for the agent to quickly learn about the goal i.e gripper base touching the object.

3.3 Interim Reward

As discussed above, Interim reward is awarded based on the moving averaged distance between the arm and the object. A reward multiplier is used to increase the impact of reward when the arm is closer to the object.The delta of the distance computed is a small value, reward multiplier is used to make the reward value in the range of REWARD_WIN parameter. A penalty of -0.2 is added to learn about each move that prevents the agent to hold and stop moving. The value of 0.2 is achieved by trial and error.

4 Hyperparameters

All the hyperparametrs are decided after a series of trial and error steps. All the hyperparametr values are summarised in the following table

- INPUT_WIDTH and INPUT_HEIGHT : Agent makes prediction by looking at each camera frame by mapping pixels to Q map. Bigger image will require higher memory as well compute capability. Image resolution of 64x64 works good for both the task as it it carries optimum information while consuming appropriate compute resources.
- OPTIMIZER : Many variations of gradient descent are mention in the lesson resources. RMSprop optimizer was tried and it worked well for both the task.

TABLE 2
HYPER Parameters

Hyper parameters	Values
INPUT_CHANNELS	3
GAMMA	0.9
EPS_START	0.9
EPS_END	0.05
EPS_DECAY	200
INPUT_WIDTH	64
INPUT_HEIGHT	64
OPTIMIZER	RMSprop
LEARNING_RTAE	0.01
REPLAY_MEMORY	10000
BATCH_SIZE	32
LSTM_SIZE	256
NUM_ACTIONS	6

- LEARNING_RATE : A slower learning rate takes more time but archives controlled learning. Based on the learning rate, the optimizer adjusts the weights in the direction of steepest gradient. Higher learning rate makes the optimizer to learn faster but it possess the risk of divergence, The local minima may get missed with higher learning rate.With a lower learning rate, training was more reliable, though it consumed more time. A learning rate of 0.01 worked well for both the tasks.
- BATCH_SIZE : Batch size is a hyperparameter of gradient descent that control the number of training samples to work through before the model's internal parameters are updated. At the end of the batch, the predictions are compared to the expected output variables and an error is calculated. From this error, the update algorithm is used to improve the model, e.g. move down along the error gradient.In the case of mini-batch gradient descent, popular batch sizes include 32, 64, and 128 samples. Higher batch size uses more memory and it takes more time for model update. In our case, Batch size of 32 worked well for both the tasks.
- NUM_ACTIONS : There are two actions per robot's joint. The actions for each joint includes either increasing the joint position/velocity or decreasing the joint's position/velocity. Since we have chosen position control for both the tasks, the actions comprises of increasing or decreasing the joint's position.
- LSTM_SIZE : LSTM is used for recurrent neural network.The agent uses old actions to learn quickly from the present samples. Feed-forward networks, had the limitation that they did not have a memory element to them. RNNs are a type of neural network which utilize memory, i.e. the previous state, to predict the current output. RNNs are more effective when they are only trying to learn from the most recent information.The LSTM architecture keeps track of both, the long-term memory and the short-term memory, where the short-term memory is the output or the prediction. Every camera frame, at every simulation iteration, is fed into the DQN and the agent then makes a prediction and carries out an appropriate action. Using LSTMs as part of that network, we can train our network by taking into consideration multiple past frames from the camera sensor instead of a single frame.
  Bigger LSTM size uses more computing power but makes efficient RNNs. A LSTM size of 256 worked optimum for our case.

5 Result

After setting the reward parameter and other hyperparameters as discussed above,a 92% accuracy was achieved for task 1 within 280 cycles. Similarly 86% accuracy was achieved for task 2 i.e when the gripper base touches the object within 340 cycles.The results are shown in following figure.

Fig. 1. Deep RL Arm results : Robot arm touches the object.

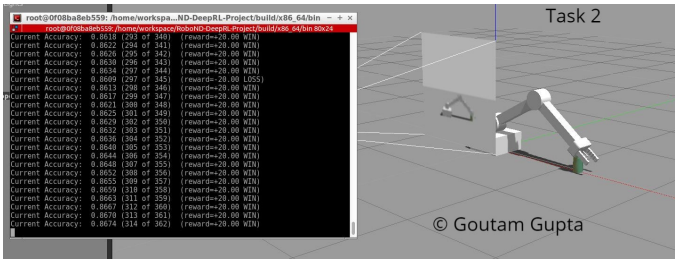

Fig. 2. Deep RL Arm results : Gripper base touches the object.

6 Conclusion / Future work

Both the tasks were performed with position control. Velocity control was also tried for both the tasks but it wasn't able to yield required accuracy. Velocity control can be explored further after tuning the hyperparametrs again. Higher frame sizes for the images can be used on more compute intensive platform which would help in achieving accuracy within stipulate time frame. Google DeepMind has devised a solid algorithm for tackling the continuous action space problem. They have produced a policy-gradient actor-critic algorithm called Deep Deterministic Policy Gradients (DDPG) which is off-policy and model-free, and that uses some of the same methods from Deep Q-Networks (DDPG).These methods can be further tried in the case robotic arm manipulator.