Deep RL Arm Manipulation

This project is based on the Nvidia open source project "jetson-reinforcement" developed by [Dustin Franklin](#) and train on Nvidia Jeston TX2 platform. The goal of the project is to create a DQN agent and define reward functions to teach a robotic arm to realize two following 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.

Background

C++ API (application programming interface)

C++ API provides an interface to the Python code written with PyTorch, but the wrappers use Python's low-level C to pass memory objects between the user's application and Torch without extra copies. By using a compiled language (C/C++) instead of an interpreted one, performance is improved, and speeded up even more when GPU acceleration is leveraged.

Gazebo Arm Plugin

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`.

You can refer to the documentation for more details on the above：

- [Subscribers in Gazebo](#)
- [Gazebo API](#)

Parameters Tuning

There are several sections need to be tuned in `ArmPlugin.cpp`, I used different parameter settings for two tasks.

Task 1

- **Hyperparameters**

I used velocity control in this task, i.e., `VELOCITY_CONTROL` is set to be `true`.

`INPUT_WIDTH` and `INPUT_HEIGHT` are reduced into 64 from 512 to save memory.

`USE_LSTM` is true with lstm size of 256.

The initial reward win and loss are set as +-10.

```
// Turn on velocity based control
#define VELOCITY_CONTROL true
#define VELOCITY_MIN -0.2f
#define VELOCITY_MAX  0.2f

// TODO - Tune the following hyperparameters
#define INPUT_WIDTH   64
#define INPUT_HEIGHT  64
#define OPTIMIZER "RMSprop"
#define LEARNING_RATE 0.05f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 32
#define USE_LSTM true
#define LSTM_SIZE 256

// TODO - Define Reward Parameters
#define REWARD_WIN  10.0f
#define REWARD_LOSS -10.0f
```

- Issue a reward based on collision between the desired arm part and the object.

```
if((strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0))
{
    rewardHistory = REWARD_WIN;
    newReward  = true;
    endEpisode = true;
    return;
}
else {
    // Give penalty for non correct collisions
    rewardHistory = REWARD_LOSS;
    newReward  = true;
    endEpisode = true;
}
```

- Set a penalty for robot wasting time.

```
if( maxEpisodeLength > 0 && episodeFrames > maxEpisodeLength )
{
    printf("ArmPlugin - triggering EOE, episode has exceeded %i frames\n", maxEpisodeLength);
    rewardHistory = REWARD_LOSS;
    newReward     = true;
    endEpisode    = true;
}
```

- Set a penalty for robot hitting the ground.

```
if( gripBBox.min.z < groundContact )
{
    printf("GROUND CONTACT, EOE\n");
    rewardHistory = REWARD_LOSS * 20.0f;
    newReward      = true;
    endEpisode     = true;
}
```

- A `distPenalty` reward is added to encourage arm approach the object quickly for the intermediary reward.

```
const float distDelta  = lastGoalDistance - distGoal;
const float alpha = 0.2f;

// compute the smoothed moving average of the delta of the distance to the goal
avgGoalDelta  = (avgGoalDelta * alpha) + (distDelta * (1.0f - alpha));
float distPenalty = (1.0f - exp(distGoal)) * 0.1f;

if(avgGoalDelta > 0)
{rewardHistory = REWARD_WIN * 0.5f - distGoal * 0.5f + distPenalty ;}
else
{rewardHistory = avgGoalDelta + distPenalty;}
newReward      = true;
```

Task 2

- Hyperparameters

  This task is more challenge, I failed to realize velocity control, hence position control is used here, i.e., `VELOCITY_CONTROL` is false.

  I don't find obvious improvement using LSTM, but still use it with a smaller `LSTM_SIZE` here.

  `EPS_END` is reduced to 0.01 so that the robot arm is more stable when it finish learning.

  `Adam` optimizer is adopted due to its advantage in other deep learning applications.

  `REWARD_WIN` and ·REWARD_LOSS` are set as +-1 to have a better understanding.

  Note: `EPS_DECAY` can be reduced if your reward function is appropriate. The premises is your robot should find the right way within this epochs. This can save lots of time when you try to save your final result.

  ```
  // Turn on velocity based control
  #define VELOCITY_CONTROL false
  #define VELOCITY_MIN -0.2f
  #define VELOCITY_MAX  0.2f

  // Define DQN API Settings
  #define INPUT_CHANNELS 3
  #define ALLOW_RANDOM true
  #define DEBUG_DQN false
  ```

```
#define GAMMA 0.9f
#define EPS_START 0.9f
#define EPS_END 0.01f
#define EPS_DECAY 200

// TODO - Tune the following hyperparameters
#define INPUT_WIDTH   64
#define INPUT_HEIGHT  64
#define OPTIMIZER "Adam"
#define LEARNING_RATE 0.01f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 128
#define USE_LSTM true
#define LSTM_SIZE 16

// TODO - Define Reward Parameters
#define REWARD_WIN  1.0f
#define REWARD_LOSS -1.0f
```

- More sophisticated reward function is adopted when arm achieve its goal.

```
rewardHistory = REWARD_WIN*10.0f + (1.0f - (float(episodeFrames) /
float(maxEpisodeLength))) * REWARD_WIN * 100.0f;
```

- An improved intermediary reward for task 2

```
if( episodeFrames > 1 )
{
    const float distDelta  = lastGoalDistance - distGoal;
    const float alpha = 0.9f;
    // compute the smoothed moving average of the delta of the distance to the goal
    avgGoalDelta  = (avgGoalDelta * alpha) + (distDelta * (1.0f - alpha));
    float distPenalty = (1.0f - exp(distGoal));
    if(avgGoalDelta > 0.01)
    {rewardHistory = (REWARD_WIN + distPenalty*0.1f)*0.1f;}
    else
    {rewardHistory = - distGoal*2.0f;}
    newReward      = true;
}
```
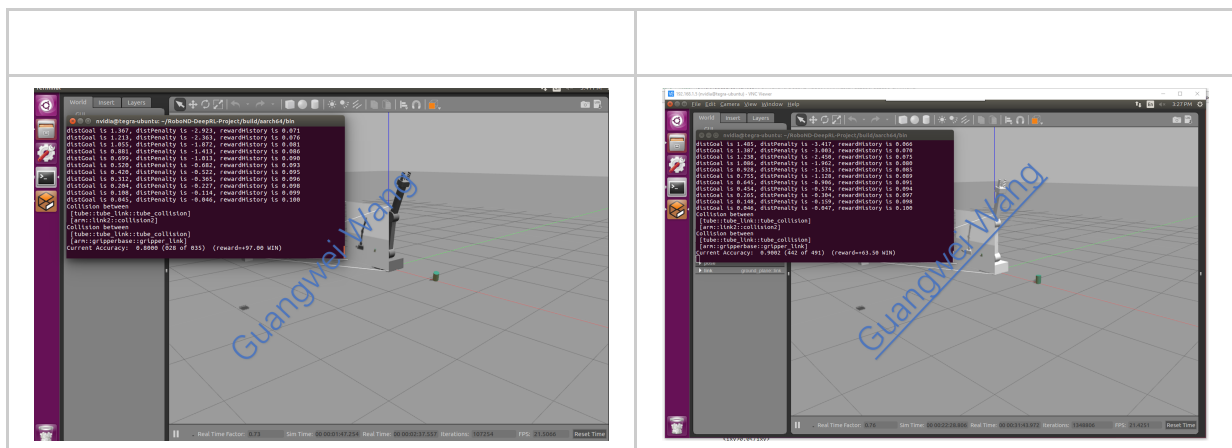
Results

Task	Total Runs	Accuracy
1: Arm	17	94.12%
1: Arm	124	90.32%
2: Gripper base	35	80.00%
2: Gripper base	491	90.02%

- Task 1: Any parts of arm touching the prop
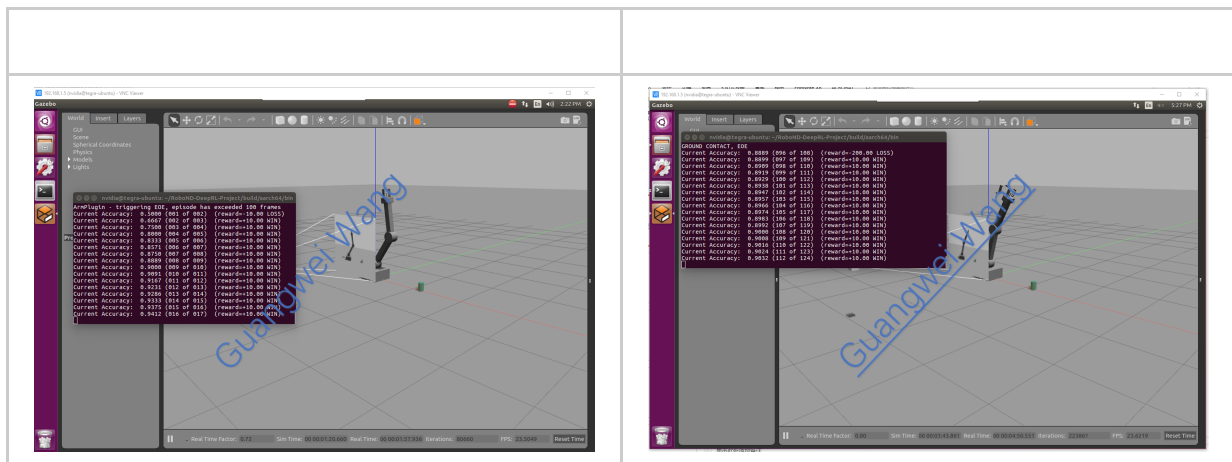
- Task 2: Gripper base touching the prop

  https://youtu.be/pjYxtJ0pTRY

Further Work

- Investigate how to use LSTM appropriately
- Realize velocity control for task 2

- Add more constraints to make arm touching prop softly.
- Realize project challenges, such as random prop position, high arm DOF, etc.