Deep Neural Network Reinforcement Learning of an Robot arm

Jung, Myoungki

Abstract—Self teaching Robots have been always a dream for robotisists and scifi fans in the world. Reinforcement learning algorithms with deep neural network techonologys opens up new horizon to these ideas and enables many robots able to learn from the environment. Apperance of Alpha GO [1] indeed shed new fresh lights on this old research field and now this topic is one of the most active field of study in artificial intellegence since then. The purpose of this report is to show an application of an end to end reinforcment learning with neural newtork to the robotics field.

Index Terms—Robot Arm, IEEEtran, Udacity, RL, 3DOF, DQN.

1 Introduction

Reinforcement learning always has been in the scholars hot topic well before Deepminds' atari game mastering [2], even before the TD Gammon game shipped in a desktop. With advancement of deep neaural networks, its application to reinforcment learning enables researchers to overcome its previous theoretical limits and chalanges, high dimensional problems, speed of convergence and many more, which are the reasons why RL has been in research feild for a long time but not much progress like other topics until recently. Alphago Zero [3], an agent learning the game of Go without any human game play data unlike its predecessor, shows how eficient reinforcement learning can be compared to humans' learning by mastering the game within months and play more intellegent than any human gamer in the world in that short time.

2 Implementation

2.1 Set up

Reward functions were coded in the cpp plugin file as shown in the listing 2 and 3. The contact were determined by the `ArmPlugin::onCollisionMsg`, a callback function of contact subscription shown in listing 1.

```
void ArmPlugin::Load(physics::ModelPtr _parent, sdf
    ::ElementPtr /*_sdf*/)
{
      printf("ArmPlugin::Load('%s')\n", _parent->
    GetName().c_str());

      //pointer to the model
      this->model = _parent;
      this->j2_controller = new physics::
    JointController(model);

      // camera communication node
      cameraNode->Init();
      cameraSub = cameraNode->Subscribe("/gazebo/"
    WORLD_NAME "/camera/link/camera/image", &
    ArmPlugin::onCameraMsg, this);

      // collision detection node
      collisionNode->Init();
      collisionSub = collisionNode->Subscribe("/
    gazebo/" WORLD_NAME "/" PROP_NAME "/tube_link/
    my_contact", &ArmPlugin::onCollisionMsg, this);

      // event update
      this->updateConnection = event::Events::
    ConnectWorldUpdateBegin(boost::bind(&ArmPlugin::
    OnUpdate, this, _1));
}
```

Listing 1. ArmPlugin::Load

2.2 Reward function

The message from contacts topic was decoded and compared by `strcmp` function and determined the reward or penalty. If contacted items are tube and `gripper_link`, then the agent is rewared 12 points and other cases the agents penalised for -8 points.

```
// Define Reward Parameters
#define REWARD_WIN    12.0f
#define REWARD_LOSS  -8.0f
#define REWARD_MULTIPLIER 10.0f

#define COLLISION_FILTER "ground_plane::link::
    collision"
#define COLLISION_ITEM   "tube::tube_link::
    tube_collision"
#define COLLISION_POINT  "arm::gripperbase::
    gripper_link"

// onCollisionMsg
void ArmPlugin::onCollisionMsg(ConstContactsPtr &
    contacts)
{
      if( testAnimation ) return;

      for (int i = 0; i < contacts->contact_size();
    ++i)
      {
            if( strcmp(contacts->contact(i).
    collision2().c_str(), COLLISION_FILTER) == 0 )
    continue;

            std::cout << "Collision between[" <<
    contacts->contact(i).collision1() << "] and ["
    << contacts->contact(i).collision2() << "]\n";
```

```
            if((strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0) && (strcmp(contacts->contact(i).collision2().c_str(), COLLISION_POINT) == 0))
            {
                rewardHistory = REWARD_WIN;
                newReward  = true;
                endEpisode = true;
                return;
            }
            else {
                rewardHistory = REWARD_LOSS;
                newReward  = true;
                endEpisode = true;
            }
        }
}
```

Listing 2. Reward Function

The agent was rewared only once in the episode when its body part contacts the object. The penalty condition was when the agent touches the ground. On both contacts the episode was reset and restarts another one immediately. The Reward function did not differenciate the task into touching with body and touching with gripper. Because the gripper is the end node frequently touch the object as robot's physical build, Robot tends to master on how to touch the object with gripper every time traning it. In addition, it masters, more than 80 percent for 100 episoides, the task within 400 iterations. If the reward function sets up with cases contacting body and the tube for less rewards, less than 12 points, it could learn to touch the tube with other parts than the gripper link. However, this experiment was kept to simple to see the application of DQN to RL for Robotic arm.

2.3 Intermediary Reward

Listing 3 shows how the Intermediary Reward was calculated after timeout of an episode. This intermediary reward is important to provide feedbacks to the agent how well it went although it could not finish the task in a given time. The balance between time penalty and reward in relation to the distance to the tube is the key to provide a firm guideline for learning what is a policy the user wants to teach to the agent.

```
// if an EOE reward hasn't already been issued, compute an intermediary reward
if( hadNewState && !newReward )
{
    PropPlugin* prop = GetPropByName(PROP_NAME);

    if( !prop )
    {
        printf("ArmPlugin -- failed to find Prop '%s'\n", PROP_NAME);
        return;
    }

    // get the bounding box for the prop object
    const math::Box& propBBox = prop->model->GetBoundingBox();
    physics::LinkPtr gripper  = model->GetLink(GRIP_NAME);

    if( !gripper )
    {
        printf("ArmPlugin -- failed to find Gripper '%s'\n", GRIP_NAME);
        return;
    }

    // get the bounding box for the gripper
    const math::Box& gripBBox = gripper->GetBoundingBox();
    const float groundContact = 0.05f;

    if( gripBBox.min.z <= groundContact || gripBBox.max.z <= groundContact )
    {
        // set appropriate Reward for robot hitting the ground.
        printf("GROUND CONTACT, EOE\n");
        rewardHistory = REWARD_LOSS;
        newReward     = true;
        endEpisode    = true;
    }
    else
    {
        // Issue an interim reward based on the distance to the object
        const float distGoal = BoxDistance(gripBBox, propBBox); // compute the reward from distance to the goal

        if(DEBUG){printf("distance('%s', '%s') = %f\n", gripper->GetName().c_str(), prop->model->GetName().c_str(), distGoal);}

        // issue an interim reward based on the delta of the distance to the object
        if( episodeFrames > 1 )
        {
            const float distDelta  = lastGoalDistance - distGoal;
            const float movingAvg  = 0.95f;
            const float timePenalty  =  0.25f;

            // compute the smoothed moving average of the delta of the distance to the goal
            avgGoalDelta  = (avgGoalDelta * movingAvg) + (distDelta * (1.0f - movingAvg));
            rewardHistory = (avgGoalDelta - timePenalty) * REWARD_MULTIPLIER;
            newReward     = true;
        }

        lastGoalDistance = distGoal;
    }
}
```

Listing 3. Reward Function

Bounding box of the gripper determines whether it is contacted to the gound or not. The distance to the tube was calcaulated into reward with a moving average of 20 fraction. The time penalty was set to 0.25 to penalise timeout status. With low or no penalty or interim reward, agent may enjoy time lifting arms for no feedback or bending backwards and going far from the tube. Without this intrim reward, the agent might discard all the effort, bending joint towards the tube little by little, and would not learn quickly. The deep network will not converge soon.

3 Hyperparameters

3.1 Default Hyperparameters

The default parameters are summerised on table 1. These parameters are not changed from the default settings.

3.2 Custom Hyperparameters

The customisation of parameters were inevitable to reflect the environmental factors for the training. The gist of the

TABLE 1
Summary of Default Hyperparameters

Parameter	Value
VELOCITY_CONTROL	false
VELOCITY_MIN	-0.2f
VELOCITY_MAX	0.2f
INPUT_CHANNELS	3
ALLOW_RANDOM	true
DEBUG_DQN	true
GAMMA	0.9f
EPS_START	0.9f
EPS_END	0.05f
EPS_DECAY	200

defined parameters are shown on Table 2. `INPUT_WIDTH` and `INPUT_HEIGHT` are a downsized from the subscribed camera image feed. Usual RL Hyperparameters were defined, RMSProp optimise, 0.1 of rather high learning rate for simple tasks, smaller batch size for less memory space and affordable size. Replay memory is a new concept for DQN and is set to 20000, which is big but this samples with low coherency between samples and converges faster and more stable during the training. Because of the training is timely expensive and sometimes does not finish for a long time. Replay memory acts like shadow training for boxers and treats experience data from the training more valable. If the number of experience replay is small, the training can be stuck in a local minima, with a large number of experience sets, this does not makes any issues. LSTM was used to track the movement and effectively makes the sequence labled and turn the training with single image into a learning in consideration of sequences and occurences together. In other words, the agents can see the casuality between time frame and choose the best action instead choosing the action from a snapshot of the time frame. LSTM size more than 256 was ineffective for a short period of training like this experiment. Reward for winning was set higher than losing. Higher

TABLE 2
Summary of Custom Hyperparameters

Parameter	Value
INPUT_WIDTH	64
INPUT_HEIGHT	64
OPTIMIZER	RMSprop
LEARNING_RATE	0.1f
BATCH_SIZE	32
REPLAY_MEMORY	20000
USE_LSTM	true
LSTM_SIZE	256

winning ratio tends to converge the training faster and does not let the agent perform unexpected actions avoiding not to get punished. Robot arm tends to touch the object rather than trying not approach any part of the robot arm to the ground. The Hyperparameters were used to initialise a DQN agent in the function `ArmPlugin::createAgent` as shown in the listing 4.

```
bool ArmPlugin::createAgent()
{
	if( agent != NULL )
		return true;

	// Create DQN Agent
	agent = dqnAgent::Create(INPUT_WIDTH,
    INPUT_HEIGHT, INPUT_CHANNELS, DOF*3, OPTIMIZER,
    LEARNING_RATE, REPLAY_MEMORY, BATCH_SIZE, GAMMA,
     EPS_START, EPS_END, EPS_DECAY,USE_LSTM,
    LSTM_SIZE, ALLOW_RANDOM, DEBUG_DQN);

	if( !agent )
	{
		printf("ArmPlugin - failed to create DQN
     agent\n");
		return false;
	}

	inputState = Tensor::Alloc(INPUT_WIDTH,
    INPUT_HEIGHT, INPUT_CHANNELS);

	if( !inputState )
	{
		printf("ArmPlugin - failed to allocate %
    ux%ux%u Tensor\n", INPUT_WIDTH, INPUT_HEIGHT,
    INPUT_CHANNELS);
		return false;
	}

	return true;
}
```

Listing 4. ArmPlugin::createAgent function

`DOF*2` was used for `number of actions` parameter of the `ArmPlugin::createAgent` function as there are three possible actions, +, -.

4 Results

Result of the experiment shows the agent is learning the task, touching the gripper to the tube, were successfully performed.

4.1 Acheivements

The first overall 80 percent of touching the tube with gripper is shown on Figure 1.

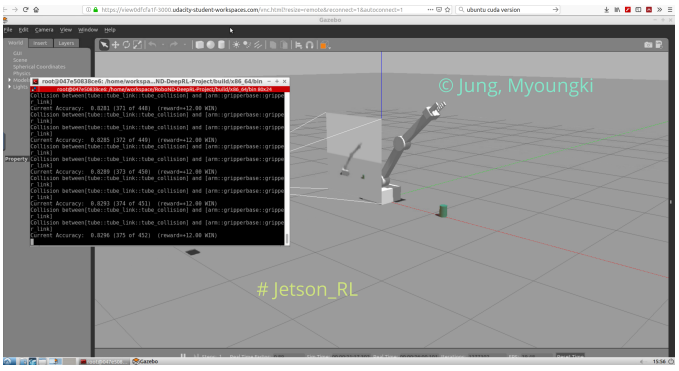

Fig. 1. Agent reaching overall 80 percent accuracy

200 iterations more the average 90 percent of contact to the tube could be realised, Figure 2, this is above the project requirement. After this point agent's `EPS_END` is 0.05 and rarely starts new exploration in control. Therefore, more than 95 percent of gripper touching the tube could be realised.

Actual moment of touching the gripper to the tube is snapshoted and shown on Figure 3. The videos realated

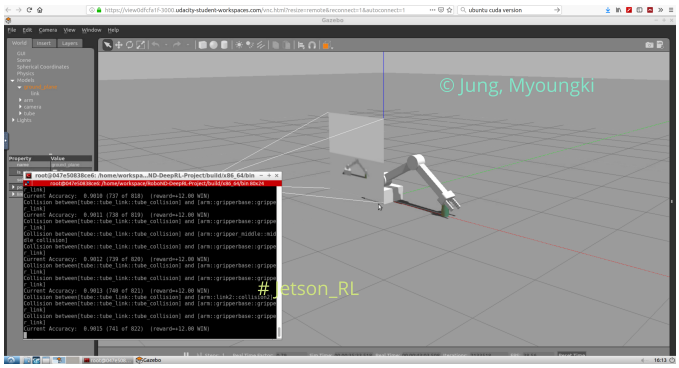

Fig. 2. Agent reaching overall 90 percent accuracy

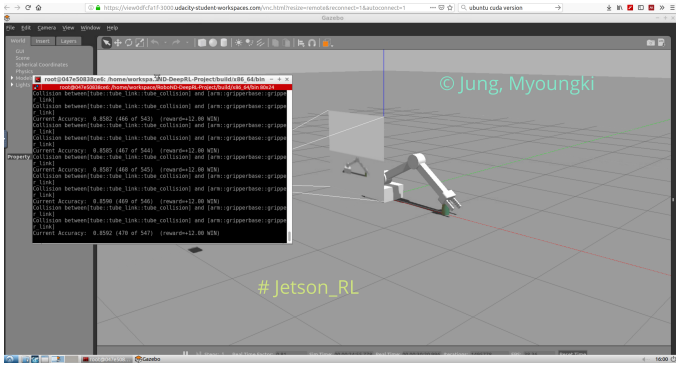

Fig. 3. Agent reaching reaching with gripper

to those images are present in `./Videos` folder of the project. `./Videos/BeginToLearn.avi` shows struggling agent not knowing what is an expected action. Other videos shows an experienced agens touching the tube confidiidently, knowing what are actions with rewards over many episodes of learning.

5 Discussion

This experiment shows usage of premade example of jetson reinforcement repository. The building of the libraries of torch was too hard on jetson Xavier, it had to be tested on the udacity workspace, however, some build errors lurks up and had make the environment for the workspace using `sudo apt-get install libignition-math2-dev` to build the `ArmPlugin.cpp`. This repository should be updated for new architectures and prefer not to use a library not readily accessible, pytorch. The backend of the DQN RL could be with tensorflow RT instead of importing python object processed by pytorch, which can be a issue with realtime robot cases by injecting performance issues. If saving DQN weights function was implemented it could be better to continue the training after pausing the simulation. Abandonning trained agents' weight every time disconnecting from the workspace seem to be a waste of computing resources and processing time.

6 Conclusion / Future work

The more advanced reinforcement agent algirithms are available. Proximal Policy Optimization Algorithms(PPO) [4] is excelent with continous state space and control space like drone flight controller or rocket propulsion control with regard to the continous environmental states. OoenAI foundation suggests Deep deterministic policy gradient (DDPG) [5] and Hindsight Experience Replay [6],a new experience replay that can learn from failure, as the top of the edge reinforcement learning algorithms and proves their algorithms are outperforming conventional algorithms easily. It is a rapidly developing area of research field. The technology used in this project can be substituted with new algorithms on cutting edge trends.

References

[1] M. C. Fu, "AlphaGo and Monte Carlo tree search: the simulation optimization perspective," in *Proceedings of the 2016 Winter Simulation Conference*, pp. 659–670, IEEE Press, 2016.
[2] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, D. Kumaran, and R. Hadsell, "Learning to Navigate in Complex Environments," 2016.
[3] D. Hassabis and D. Silver, "AlphaGo Zero: Learning from scratch," 2017.
[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," pp. 1–12, 2017.
[5] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015.
[6] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. Mcgrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight Experience Replay," no. Nips, 2017.